CHAPTER 17

Web Applications with Spring MVC

Web applications have become a very important part of any enterprise system. The key requirement for a web framework is to simplify development of the web tier as much as possible. In this chapter, you will learn how to develop web applications using Spring. We will start with an explanation of Spring MVC architecture and the request cycle of Spring web applications, introducing handler mappings, interceptors, and controllers. Then we will show how we can use different technologies to render HTML in the browser.

Before we dive into a discussion of Spring MVC, we should take a look at what MVC stands for and how it can help develop more flexible web applications.

MVC Architecture

MVC is the acronym for the model view controller architectural pattern. The purpose of this pattern is to simplify the implementation of applications that need to act on user requests and manipulate and display data. There are three distinct components of this pattern:

- The model represents data that the user expects to see. In most cases, the model will consist of JavaBeans.
- The view is responsible for rendering the model. A view component in a text editor will probably display the text in appropriate formatting; in a web application, it will, in most cases, generate HTML output that the client's browser can interpret.
- The controller is a piece of logic that is responsible for processing and acting on user requests: it builds an appropriate model and passes it to the view for rendering. In the case of Java web applications, the controller is usually a servlet. Of course, the controller can be implemented in any language a web container can execute.

Currently, there are two MVC models. In the domain of web applications, model one architecture is illustrated in Figure 17-1.

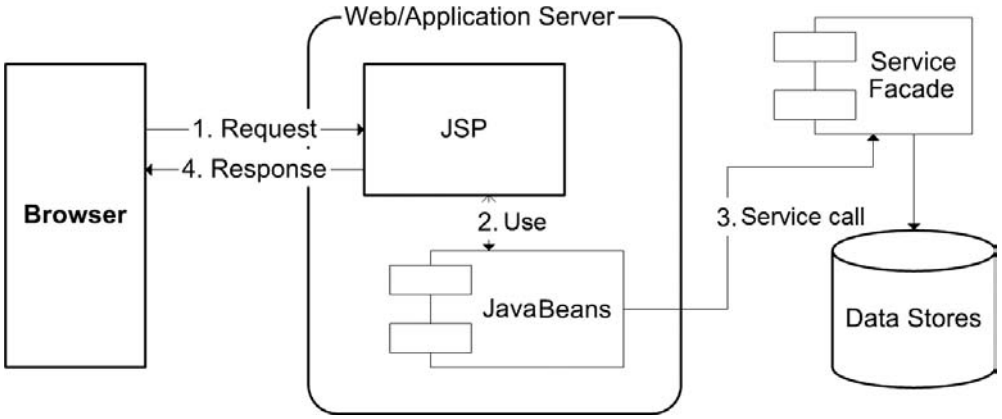

Figure 17-1. *MVC model one architecture*

As Figure 17-1 shows, the JSP pages are at the center of the application. They include both the control logic and presentation logic. The client makes a request to a JSP page, and the logic in the page builds the model (typically using POJOs) and renders the model. The separation of the presentation layer and control layer is not very clear. In fact, with the exception of "Hello, World" applications, model one quickly grows out of control, simply because of the amount of logic that different JSP pages need to perform.

Model two is far more manageable in a larger application. Whereas in model one, a JSP page acted as both view and controller, model two adds a separate controller (see Figure 17-2).

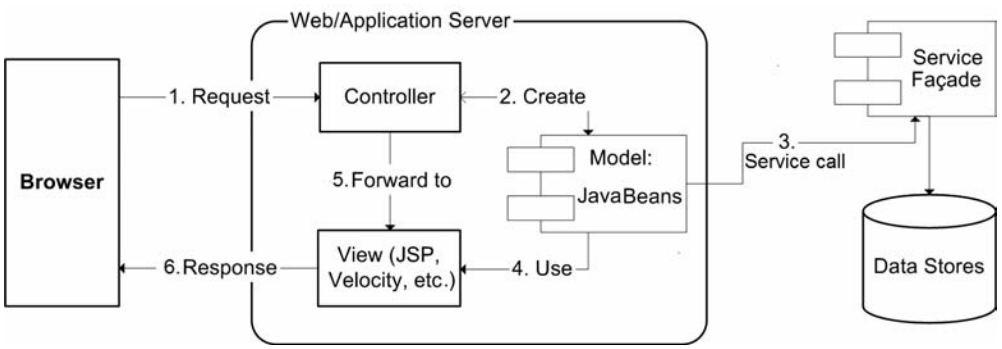

Figure 17-2. *MVC model two architecture*

Now, the controller accepts user requests, prepares the model, and passes it to the view for rendering. The JSP pages no longer contain logic for processing the requests; they simply display the model prepared by the controller.

We have used JSP pages in place of the view and controller in model one's diagram and in place of the view in model two's diagram. Using JSP pages as the view illustrates only one type of MVC implementation. However, as Spring MVC architecture is an implementation of model two MVC, the view can be created using anything that can render the model and return it to the client.

Now that we have covered the basics of the MVC pattern, we can take a more detailed look at how this pattern is implemented in Spring and how you can use it in your web applications.

Spring MVC

Spring MVC support allows us to build flexible applications using MVC model two. The implementation is truly generic. The model is a simple `Map` that holds the data; the view is an interface whose implementations render the data; and the controller is an implementation of the `Controller` interface.

■ **Note** We have addressed Spring MVC support for servlet-based web applications in this chapter. Spring goes beyond that and offers full support for JSR 168 portlet development. The major difference between servlets and portlets is that a portlet can have two distinct phases: the action phase and the render phase. The action phase is executed only once, when some business layer changes are invoked (such as a database update). The render phase is executed whenever a user requests a page. The Spring portlet MVC framework is designed to be a mirror image of Spring web MVC architecture, as much as possible. A detailed explanation of the Spring portlet framework is beyond the scope of this book, but the full reference and documentation can be found on the Spring Framework web site.

Spring's implementation of the MVC architecture for web applications is based around `DispatcherServlet`. This servlet processes requests and invokes appropriate `Controller` elements to handle them.

The `DispatcherServlet` intercepts incoming requests and determines which controller will handle the request. The Spring controllers return a `ModelAndView` class from their handling methods. The `ModelAndView` instance holds a reference to a view and a model. The model is a simple `Map` instance that holds JavaBeans that the `View` interface is going to render. The `View` interface, when implemented, defines the render method. It follows that the `View` implementation can be virtually anything that can be interpreted by the client.

MVC Implementation

If we want to create a web application with Spring, we need to start with the basic `web.xml` file, where we specify the `DispatcherServlet` and set the mapping for the specified `url-pattern`. Listing 17-1 shows a sample `web.xml` file.

Listing 17-1. *A Web.Example*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee ➥
         http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
         version="2.4">
    <servlet>
        <servlet-name>ch17</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>ch17</servlet-name>
        <url-pattern>*.html</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name> ch17</servlet-name>
        <url-pattern>*.tile</url-pattern>
    </servlet-mapping>
</web-app>
```

This `web.xml` file defines the `ch17` servlet of the class `DispatcherServlet` that maps to all requests to `*.html` or `*.tile`.

■ **Note** We usually create a mapping to `*.html` because it is a recognized extension and easily fools search engines into thinking that the page is not dynamically generated.

Using Handler Mappings

How does our web application know which servlet (controller implementation) to invoke on a specific request? This is where Spring handler mappings kick in. In a few easy steps, you can configure URL mappings to Spring controllers. All you need to do is edit the Spring application context file.

Spring uses `HandlerMapping` implementations to identify the controller to invoke and provides three implementations of `HandlerMapping`, as shown in Table 17-1.

Table 17-1. *HandlerMapping Implementations*

HandlerMapping	Description
`BeanNameUrlHandlerMapping`	The bean name is identified by the URL. If the URL were `/product/index.html`, the controller bean ID that handles this mapping would have to be set to `/product/index.html`. This mapping is useful for small applications, as it does not support wildcards in the requests.
`SimpleUrlHandlerMapping`	This handler mapping allows you to specify in the requests (using full names and wildcards) which controller is going to handle the request.
`ControllerClassNameHandlerMapping`	This handler mapping is part of the convenience over configuration approach introduced with Spring 2.5. It automatically generates URL paths from the class names of the controllers. This implementation is covered in more detail later in this chapter.

All three `HandlerMapping` implementations extend the `AbstractHandlerMapping` base class and share the following properties:

- `interceptors`: This property indicates the list of interceptors to use. `HandlerInterceptors` are discussed in the next section.
- `defaultHandler`: This property specifies 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.
- `alwaysUseFullPath`: If this property is set to `true`, Spring will use the full path within the current servlet context to find an appropriate handler. If this property is set to `false` (the default), the path within the current servlet mapping will be used. For example, if a servlet is mapped using `/testing/*` and the `alwaysUseFullPath` property is set to `true`, `/testing/viewPage.html` would be used, whereas if the property is set to `false`, `/viewPage.html` would be used.
- `urlPathHelper`: Using this property, you can tweak the `UrlPathHelper` used when inspecting URLs. Normally, you shouldn't have to change the default value.

- `urlDecode`: The default value for this property is `false`. The `HttpServletRequest` returns request URLs and URIs that are *not* decoded. If you do want them to be decoded before a `HandlerMapping` uses them to find an appropriate handler, you have to set this to `true` (which requires JDK 1.4). The decoding method uses either the encoding specified by the request or the default ISO-8859-1 encoding scheme.
- `lazyInitHandlers`: This allows for lazy initialization of *singleton* handlers (prototype handlers are always lazily initialized). The default value is `false`.

■ **Note** The last four properties are only available to subclasses of `org.springframework.web.servlet.handler.AbstractUrlHandlerMapping`.

We will start with the example of `BeanNameUrlHandlerMapping`. This is the simple `HandlerMapping` implementation that maps controller bean IDs to the servlet URLs. This `HandlerMapping` implementation is used by default if no `HandlerMapping` is defined in the Spring context files. Listing 17-2 shows an example of the `BeanNameUrlHandlerMapping` configuration, without the actual `BeanNameUrlHandlerMapping` bean (`DispatcherServlet` will instantiate it by default, if no other `HandlerMapping` has been configured).

Listing 17-2. *BeanNameUrlHandlerMapping Configuration*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
...>

<beans>
    <bean name="/index.html.form" ➥
class=" com.apress.prospring.ch17.web.IndexController "/>
</beans>
```

`SimpleUrlHandlerMapping` offers more flexibility in the request mappings. You can configure the mapping as key/value properties in the `publicUrlMapping` bean. In Listing 17-3, you can see a simple example of a Spring application context file containing the handler mapping configuration.

Listing 17-3. *SimpleUrlHandlerMapping Definitions*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
...>

<beans>
    <bean id="publicUrlMapping"
        class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
                <property name="mappings">
            <value>
                /index.html=indexController
                /product/index.html=productController
                /product/view.html=productController
                /product/edit.html=productFormController
            </value>
        </property>
    </bean>
</beans>
```

Spring Controllers

Controllers do all the work to process the request, build the model based on the request, and pass the model to the view for rendering. Spring's `DispatcherServlet` intercepts the requests from the client and uses a `HandlerAdapter` implementation that is responsible for delegating the request for further processing. You can implement the `HandlerAdapter` yourself, allowing you to modify the chain of command the request must pass through.

The `DispatcherServlet` has a `List handlerAdapters` property that allows you to specify the `HandlerAdapter` implementations you wish to use. To make sure the `HandlerAdapter` implementations are called in the right order, you can choose to implement the `Ordered` interface in your `HandlerAdapter` to indicate the position among other `HandlerAdapter` implementations.

If the `handlerAdapters` property of `DispatcherServlet` is `null`, the `DispatcherServlet` will use `SimpleControllerHandlerAdapter`. Because we are not going to provide any additional `HandlerAdapter` implementations, our application will use the `SimpleControllerHandlerAdapter`.

`SimpleControllerHandlerAdapter` delegates the request to the implementation of the `Controller` interface, hence, the beans—handlers that are to act as controllers must implement the `Controller` interface. This approach provides you the flexibility to write your own implementation from scratch or to use one of the convenient implementations already provided. The `Controller` interface depends on `HttpServletRequest` and `HttpServletResponse`, which means that you can use it only in web applications.

Let's take a look at the most basic implementation of the `Controller` interface. We are going to create an `IndexController` that simply writes "Hello, World" to the response stream, as shown in Listing 17-4.

Listing 17-4. *IndexController Implementation*

```
public class IndexController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request,
        HttpServletResponse response) throws Exception {

        response.getWriter().println("Hello, world");

        return null;
    }

}
```

The only method we need to implement is `ModelAndView handleRequest(HttpServletRequest, HttpServletResponse)`. We are returning `null` as the return value of `ModelAndView`, which means that no view will be rendered and the result will be written directly to the output of the response, which will be committed and returned to the client.

Implementing the `Controller` interface is, in most cases, too much work, so Spring provides a number of useful superclasses.

AbstractController

At first glance, it might seem that `AbstractController` is simply a wrapper around the interface that will force you to implement the `handleRequestInternal` method to process the request. This is only partially true, as `AbstractController` extends the `WebContentGenerator` class that allows you to set additional properties to control the request and response. Additionally, `WebContentGenerator` extends `WebApplicationObjectSupport`, which, in turn, extends the `ApplicationObjectSupport` class that implements `ApplicationContextAware`. In other words, extending your controller from `AbstractController`

rather than implementing the `Controller` interface directly gives you access to `ServletContext`, `WebApplicationContext`, `ApplicationContext`, `Log`, and `MessageSourceAccessor`.

Table 17-2 provides a closer look at properties you can set that are related to the web application environment.

Table 17-2. *WebContentGenerator and AbstractController Properties*

Property	Description	Default Value
`supportedMethods`	Supported and allowed HTTP methods	`GET`, `POST`
`requiresSession`	Specifies whether an `HttpSession` instance is required to process the request	`false`
`useExpiresHeader`	Specifies whether to use the HTTP 1.0 `expires` header	`true`
`useCacheControlHeader`	Specifies whether to use the HTTP 1.1 cache-control header	`true`
`cacheSeconds`	Instructs the client to cache the generated content for the specified number of seconds	`-1`
`synchronizeOnSession`	Specifies whether the controller should synchronize an instance of `HttpSession` before invoking `handleRequestInternal`. Useful for serializing reentrant request handling from a single client	`false`

As an example, we can set the `cacheSeconds` property to `10` and refresh the page in the client (making sure we are not instructing the client to bypass the cache), and we should receive new content from the server only every 10 seconds (see Listing 17-5).

Listing 17-5. *IndexController Implementation Using AbstractController*

```
public class IndexController extends AbstractController {

    protected ModelAndView handleRequestInternal(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        setCacheSeconds(10);
        response.getWriter().println("Hello, world at " + ➥
System.currentTimeMillis());
        return null;
    }

}
```

If you compare the implementation of `IndexController` from Listings 17-4 and 17-5, you will see that there is very little difference, except for the fact that code in Listing 17-5 now has full access to the context (both `Servlet` and `Application`) and can manipulate the HTTP headers more easily.

ParameterizableViewController

The `ParameterizableViewController` is a very simple subclass of `AbstractController`; it implements the `handleRequestInternal` method to return a new model with a name set in its `viewName` property. No data is inserted into the model, and the only reason you would choose to use this controller is simply to display a view using its name.

Listing 17-6 shows the `ParameterizableIndexController` we have created to demonstrate the functionality of the `ParameterizableViewController`.

Listing 17-6. *ParametrizableIndexController Implementation*

```
public class ParametrizableIndexController extends ParameterizableViewController {

}
```

We will add a `parameterizableIndexController` bean to the application context file, set its `viewName` property to `product-index`, and add a reference to it to the `publicUrlMapping` bean as shown in Listing 17-7.

Listing 17-7. *parametrizableIndexController Bean Declarations*

```
<beans xmlns="http://www.springframework.org/schema/beans"
       ...>

<beans>
    <bean id="publicUrlMapping"
        class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <value>
                /index.html=indexController
                /pindex.html=parametrizableIndexController
                /product/index.html=productController
                /product/view.html=productController
                /product/edit.html=productFormController
                /product/image.html=productImageFormController
            </value>
        </property>
    </bean>

    <bean id="parametrizableIndexController"
        class="com.apress.prospring.ch17.web.ParametrizableIndexController">
        <property name="viewName" value=products-index/ >
    </bean>
</beans>
```

MultiActionController

This `Controller` implementation is far more interesting than the `ParameterizableViewController`. It is also a subclass of `AbstractController`, giving it access to all its properties and methods. Most importantly, it lets you provide as many implementations of `public ModelAndView(HttpServletRequest, HttpServletResponse)` as you need. You can choose to implement the methods in your subclass of `MultiActionController`, or you can specify a delegate object that implements these methods and the `MultiActionController` will invoke the methods on the delegate object. Using this `Controller` implementation, you can map multiple URLs to the same controller and use different methods to process the various URLs.

The two additional properties of `AbstractController`—`delegate` and `methodNameResolver`—are used to tell the `MultiActionController` which method on which object to invoke for each request. If the delegate property is left to its default value of `null`, the controller will look up and invoke the method on the `MultiActionController` subclass itself; if the delegate is not `null`, the method will be invoked on the delegate.

The `methodNameResolver` must be set to an implementation of `MethodNameResolver`. The three implementations of `MethodNameResolver` are shown in Table 17-3.

Table 17-3. *MethodNameResolver Implementations*

Implementation	Description
`InternalPathMethodNameResolver`	The method name will be taken from the last part (the file part) of the path, excluding the extension. When using this resolver the path, `/servlet/foo.html`, will map to the method `public ModelAndView foo(HttpServletRequest, HttpServletResponse)`. This is also the default implementation used in `MultiActionController`.
`ParameterMethodNameResolver`	The method name will be taken from the specified request parameter. The default parameter name is `action`; you can change the parameter name in the context file.
`PropertiesMethodNameResolver`	The method name will be resolved from an external properties file. You can specify exact mapping, such as `/test.html=handleTest`, or you can use wildcards, such as `/*=handleAll`.

■ **Note** You won't be able to map URLs like `/view-product.html` using `InternalPathMethodNameResolver`. You cannot define method `public ModelAndView view-product()` in Java, because the hyphen is an illegal character in Java definitions. If you need to use URLs like this, you'll have to implement your own `MethodNameResolver`. However, for most cases, the resolvers that Spring provides are sufficient.

Let's take a look at the simplest implementation of the `MultiActionController` subclass.

Listing 17-8. *MultiActionController Subclass*

```
public class ProductController extends MultiActionController {

    public ModelAndView view(HttpServletRequest request,
        HttpServletResponse response) throws Exception{

        response.getOutputStream().print("Viewing product " +
            request.getParameter("productId"));

        return null;
    }

}
```

The `ProductController` from Listing 17-8 adds only one method, `view()`. If the path `/product/*` is mapped to this controller and if the request is `/product/view.html?productId=10`, the output displayed in the browser is going to be "Viewing product 10".

The fact that the URL from the previous section invoked the method `public ModelAndView view(HttpServletRequest request, HttpServletResponse response)` of `ProductController` proves that the `MultiActionController` defaults to using `InternalPathMethodNameResolver` as a method name resolver and that the delegate property is `null`.

If you want to use `InternalPathMethodNameResolver` with custom properties, you can always define it as a Spring-managed bean and add any properties you like. In the example shown in Listing 17-9, we add the suffix property, with the value "Handler". Using this configuration, Spring will look for a method whose name matches the last part of the URL without the extension with the string "Handler" appended to it.

Listing 17-9. *MultiActionController Subclass*

```
<bean id="internalPathMethodNameResolverclass="org.springframework.web.➥
          servlet.mvc.multiaction.InternalPathMethodNameResolver">
    <property name="suffix" value="Handler"/>
</bean>

<bean id="productController"
    class="com.apress.prospring2.ch17.web.product.ProductController">
    <property name="methodNameResolver" ref="internalPathMethodNameResolver"/>
</bean>
```

Now, the `view.html` URL will be mapped to the `public ModelAndView viewHandler()` method of `MultiActionController`. Let's take a look at how we can configure other `methodNameResolvers`, starting with the `ParameterMethodNameResolver`.

By default, `ParameterMethodNameResolver` uses the `action` parameter name to derive the method name; we can change that by setting the `paramName` property. We can also specify a method name that will be invoked when the `paramName` parameter is not present in the request by setting the `defaultMethodName` property to the name of the method to be invoked (see Listing 17-10).

Listing 17-10. *The ch17-servlet.xml Definition with ParameterMethodNameResolver*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   ...>
    <!-- other beans -->
    <bean id="productController"
        class="com.apress.prospring2.ch17.web.product.ProductController">
        <property name="methodNameResolver" ref="productMethodNameResolver"/>
    </bean>

    <bean id="productMethodNameResolver"
        class="org.springframework.web.servlet.mvc.multiaction.➥
            ParameterMethodNameResolver">
            <property name="paramName" value="method"/>
            <property name="defaultMethodName" value="view"/>
    </bean>
</beans>
```

If we now make a request to `/product/a.html` and do not specify the method parameter, `ProductController.view` will be invoked, and we will get the same behavior if we make a request to `/product/a.html?method=view`. However, if we make a request to `/product/a.html?method=foo`, we will get an error message, because the method `public ModelAndView foo(HttpServletRequest, HttpServletResponse)` is not implemented in `ProductController`.

The last method name resolver we will discuss is the `PropertiesMethodNameResolver`. This method resolver relies on the request URI, but unlike `InternalPathMethodNameResolver`, we can specify the method names in the Spring context file.

Listing 17-11. *The ch17-servlet.xml Definition with PropertiesMethodNameResolver*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        ...>
    <!-- other beans -->
    <bean id="productController"
        class="com.apress.prospring2.ch17.web.product.ProductController">
        <property name="methodNameResolver" ref="productMethodNameResolver"/>
```

```
....
    <bean id="productMethodNameResolver"
        class="org.springframework.web.servlet.mvc.multiaction.➥
            PropertiesMethodNameResolver">
        <property name="mappings">
            <value>
                /product/view.html=view
                /product/v*.html=view
            </value>
        </property>
    </bean>
</beans>
```

This code listing demonstrates how to use `PropertiesMethodNameResolver`: we need to configure its mappings property and add a list of mappings and their handler methods. The example from Listing 17-11 declares that `/product/view.html` as well as `/product/v*.html` will map to the `public ModelAndView view(HttpServletRequest, HttpServletResponse)` method in `ProductController`. The benefit of this `MethodNameResolver` is that we can use wildcards in the mapping strings.

All these controllers are very useful, but if we had to process input submitted by a user, we would have to write a lot of code to get the submitted values and process error messages. Spring simplifies this process by providing several command `Controllers`. Before we can move ahead to the command controllers, however, we must discuss views and interceptors. These, with handlers (explained previously), will enable us to create pages in which the command controllers process the data that users enter.

Interceptors

Interceptors are closely related to mappings, as you can specify a list of interceptors that will be called for each mapping. `HandlerInterceptor` implementations can process each request before or after it has been processed by the appropriate controller. You can choose to implement the `HandlerInterceptor` interface or extend `HandlerInterceptorAdapter`, which provides default do-nothing implementations for all `HandlerInterceptor` methods. As an example, we are going to implement a `BigBrotherHandlerInterceptor` that will process each request.

Listing 17-12. *BigBrotherHandlerInterceptor Implementation*

```
public class BigBrotherHandlerInterceptor extends HandlerInterceptorAdapter {

    public void postHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler,
        ModelAndView modelAndView) throws Exception {
        // process the request
    }
}
```

The actual implementation of such an interceptor would probably process the request parameters and store them in an audit log. To use the interceptor, we will create a URL mapping and interceptor bean definitions in the Spring application context file as shown in Listing 17-13.

Listing 17-13. *HandlerMapping and HandlerInterceptor Definitions*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       ...>
```

```
    <bean id="bigBrotherHandlerInterceptor"
        class="com.apress.prospring2.ch17.web.BigBrotherHandlerInterceptor"/>

    <bean id="publicUrlMapping"
        class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="interceptors">
            <list>
                <ref local="bigBrotherHandlerInterceptor"/>
            </list>
        </property>
        <property name="mappings">
            <value>
                /index.html=indexController
                /product/index.html=productController
                /product/view.html=productController
                /product/edit.html=productFormController
            </value>
        </property>
    </bean>
</beans>
```

You can specify as many `HandlerMapping` and `HandlerInterceptor` beans as you like, provided that the actual mappings do not collide with each other (one URL mapping can have one corresponding handler).

Now, we have mapped URL requests to invoking controllers. But how can we display the actual web page in the browser? We will now take a look at Spring views.

Views, Locales, and Themes

When we touched on the `View` interface, we merely stated its uses. Now, let's take a look at it in more detail. We'll start with a custom implementation of the `View` interface that will demonstrate how simple it is to create a custom view and what Spring does to look up (and instantiate) an appropriate instance of `View` when we refer to the `View` by its name.

Using Views Programmatically

In this example, we are going to manually implement a view and return the implementation in the `ModelAndView` class, which is the return value of a call to the `AbstractController.handleRequestInternal` method.

A custom view must implement the single method from the `View` interface: `render(Map, HttpServletRequest, HttpServletResponse)`. The `View` implementation we are going to create will output all data from the model to a text file and set the response headers to indicate to the client that the returned content is a text file and should be treated as an attachment (see Listing 17-14).

Listing 17-14. *PlainTextView Implementation*

```
public class PlainTextView implements View {

    public void render(Map model, HttpServletRequest request,
        HttpServletResponse response) throws Exception {

        response.setContentType("text/plain");
        response.addHeader("Content-disposition", ➥
                                "attachment; filename=output.txt");
```

```
        PrintWriter writer = response.getWriter();
        for (Iterator k = model.keySet().iterator(); k.hasNext();) {
            Object key = k.next();
            writer.print(key);
            writer.println(" contains:");
            writer.println(model.get(key));
        }
    }

}
```

We will modify the `IndexController` class from the "Spring Controllers" section to return our custom view (see Listing 17-15).

Listing 17-15. *Modified IndexController Class*

```
public class IndexController extends AbstractController {

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

        setCacheSeconds(10);
        Map model = new HashMap();
        model.put("Greeting", "Hello World");
        model.put("Server time", new Date());

        return new ModelAndView(new PlainTextView(), model);
    }
    public String getContentType() {
        return "text/plain";
    }
}
```

Let's now send a request to the `/index.html` path. The `IndexController.handleRequestInternal` method will be called and will return an instance of `ModelAndView` with `View` set to an instance of `PlainTextView` and a model `Map` containing the keys `Greeting` and `Server time`. The render method of `PlainTextView` will set the header information that will prompt the client to display an Opening window (see Figure 17-3).

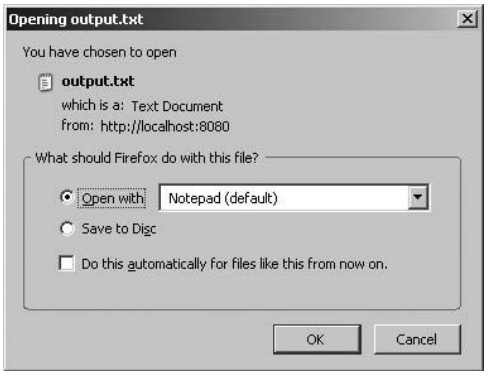

Figure 17-3. *PlainTextView prompts the client to display the Opening window to allow you to save the file.*

The content of the `output.txt` file is simply the model displayed as plain text, as shown in Figure 17-4.

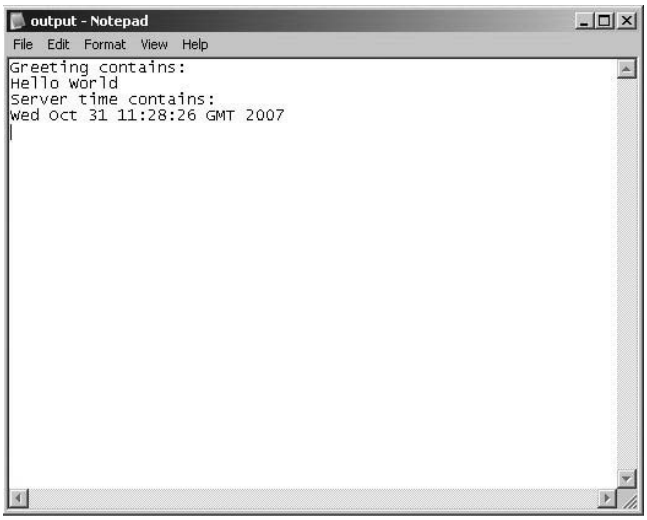

Figure 17-4. *The downloaded output.txt file*

The result is exactly what we expected: there were two entries in the model map, "Greeting" and "Server time" with "Hello World" and current date value.

This example has one disadvantage: the code in `IndexController` will create an instance of `PlainTextView` for each request. This is not necessary as the `PlainTextView` is a stateless object. Let's improve the application and make `PlainTextView` a Spring singleton bean. You can simply insert this bean into the `IndexController` via dependency injection, as Listing 17-16 shows.

Listing 17-16. *PlainTextView As a Spring Bean*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       ...>
    <bean id="plainTextView"
        class="com.apress.prospring2.ch17.web.views.PlainTextView"/>
    <bean id="indexController"
        class="com.apress.prospring2.ch17.web.IndexController">
        <property name="view" ref="plainTextView" />
   </bean>
    <!-- other beans as usual -->
</beans>
```

The `IndexController` will need to be modified to use the `plainTextView` bean instead of instantiating `PlainTextView` for each request. We will add a view property to `IndexController`, and implement a public setter method to inject the `PlainTextView` bean (see Listing 17-17).

Listing 17-17. *Modified IndexController Class*

```
public class IndexController extends AbstractController {
    private View view;
```

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

        setCacheSeconds(10);
        Map model = new HashMap();
        model.put("Greeting", "Hello World");
        model.put("Server time", new Date());

        return new ModelAndView(this.view, model);
    }

    public setView(View view){
         this.view = view;
    }

}
```

This approach is better than the one used in Listing 17-15, because each request gets the same instance of the `PlainTextView` bean. However, it is still far from ideal. A typical web application consists of a rather large number of views, and configuring all views this way would be inconvenient. Moreover, certain views require further configuration. Take a JSP view, for example: it needs a path to the JSP page. If we were to configure all views as Spring beans manually, we would have to configure each JSP page as a separate bean. An easier way to define the views and delegate all the work to Spring would be nice to have. This is where view resolvers come into the picture.

Using View Resolvers

A `ViewResolver` is a strategy interface that Spring MVC uses to look up and instantiate an appropriate view based on its name and locale. There are various view resolvers that all implement the `ViewResolver` interface's single method: `View resolveViewName(String viewName, Locale locale) throws Exception`. This allows your applications to be much easier to maintain. The locale parameter suggests that the `ViewResolver` can return views for different client locales, which is indeed the case.

Table 17-4 shows the implementations of the `ViewResolver` interface that are supplied with the Spring Framework.

Table 17-4. *ViewResolver Implementations*

Implementation	Description
`BeanNameViewResolver`	This simple `ViewResolver` implementation will try to get the `View` as a bean configured in the application context. This resolver may be useful for small applications, where you do not want to create another file that holds the view definitions. However, this resolver has several limitations; the most annoying one is that you have to configure the views as Spring beans in the application context. Also, it does not support internationalization.
`ResourceBundleViewResolver`	This resolver is far more complex. The view definitions are kept in a separate configuration file, so you do not have to configure the `View` beans in the application context file. This resolver supports internationalization.

Continued

Table 17-4. *Continued*

Implementation	Description
`UrlBasedViewResolver`	This resolver instantiates the appropriate view based on the URL, which can configure the URL with prefixes and suffixes. This resolver gives you more control over views than `BeanNameViewResolver` but can become difficult to manage in a large application and does not support internationalization.
`XmlViewResolver`	This view resolver is similar to `ResourceBundleViewResolver`, as the view definitions are kept in a separate file. Unfortunately, this resolver does not support internationalization.

Now that you know the advantages and disadvantages of your `ViewResolver` options in Spring, we can improve the sample application. We are going to discuss the `ResourceBundleViewResolver`, because it offers the most complete functionality.

We will start off by updating the application context file to include the `viewResolver` bean definition, as shown in Listing 17-18.

Listing 17-18. *ResourceBundleViewResolver Definition*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
...>
    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
        <property name="basename" value="views"/>
    </bean>
    <!-- etc -->
</beans>
```

This introduces the `viewResolver` bean that Spring will use to resolve all view names. The class is `ResourceBundleViewResolver`, and its `basename` property value is `views`, which means that the `ViewResolver` is going to look for a `views_<LID>.properties` file on the classpath, where LID is the locale identifier (`EN`, `FR`, `ES`, `CS`, etc.). If the resolver cannot locate a `views_<LID>.properties` file, it will try to open `views.properties`. To demonstrate the internationalization support in this resolver, we are going to create `views_ES.properties`, which is the Spanish language file, and `views.properties`, which will be used for any other language. In general, the syntax of the properties file looks like this: `viewname.class=class-name` and `viewname.url=view-url`. Our example's syntax is shown in Listing 17-19.

Listing 17-19. *views.properties File Syntax*

```
#index
products-index.class=org.springframework.web.servlet.view.JstlView
products-index.url=/WEB-INF/views/en_GB/product/index.jsp
```

Probably the best way to keep this file reasonably easy to maintain is to follow the logical structure of the application, using a dash as a directory separator. To create a user edit view definition, we can use the code in Listing 17-20.

Listing 17-20. *Additions to views.properties*

```
#index
user-edit.class=org.springframework.web.servlet.view.JstlView
user-edit.url=/WEB-INF/views/en_GB/user/edit.jsp
```

In Listing 17-21, you can see that the Spanish versions of the views definition files (`views_es.properties`) look very similar.

Listing 17-21. *views_es.properties File*

```
#index
products-index.class=org.springframework.web.servlet.view.JstlView
products-index.url=/WEB-INF/views/es_ES/product/index.jsp
```

Note that we kept the same view name (`products-index`) and changed only the path to the translated JSP file.

Similarly, we are going to create `index.jsp` in English in the `/WEB-INF/views/en_GB/product` directory and `index.jsp` in Spanish in `/WEB-INF/views/es_ES/product`. Finally, we are going to modify the `ProductController` to return a dummy list of `Product` objects and display this list in the view (see Listing 17-22).

Listing 17-22. *Modified ProductController*

```
public class ProductController extends MultiActionController {
    private List products;

    private Product createProduct(Long productId, String name, ➥
                                Date expirationDate) {
        Product product = new Product();
        product.setId(productId);
        product.setName(name);
        product.setExpirationDate(expirationDate);

        return product;
    }

    public ProductController() {
        products = new ArrayList();
        Date today = new Date();
        products.add(createProduct(1L, "test", today));
        products.add(createProduct(2L, "Pro Spring Appes", today));
        products.add(createProduct(3L, "Pro Velocity", today));
        products.add(createProduct(4L, "Pro VS.NET", today));
    }

    public ModelAndView index(HttpServletRequest request,
        HttpServletResponse response) {

        return new ModelAndView("products-index", "products", products);
    }
    // other methods omitted for clarity
}
```

As you can see, we have not modified the `ProductController` in any unexpected way. However, the `ModelAndView` constructor we are calling in the `index()` method is `ModelAndView (String, String, Object)` rather than `ModelAndView (View, Map)`.

To test the application, make sure the preferred language in your browser is set to anything other than Spanish. Spring will create a `product-index` view of type `JstlView` with the URL set to `/WEB-INF/views/en_GB/product/index.jsp` and render the output shown in Figure 17-5.

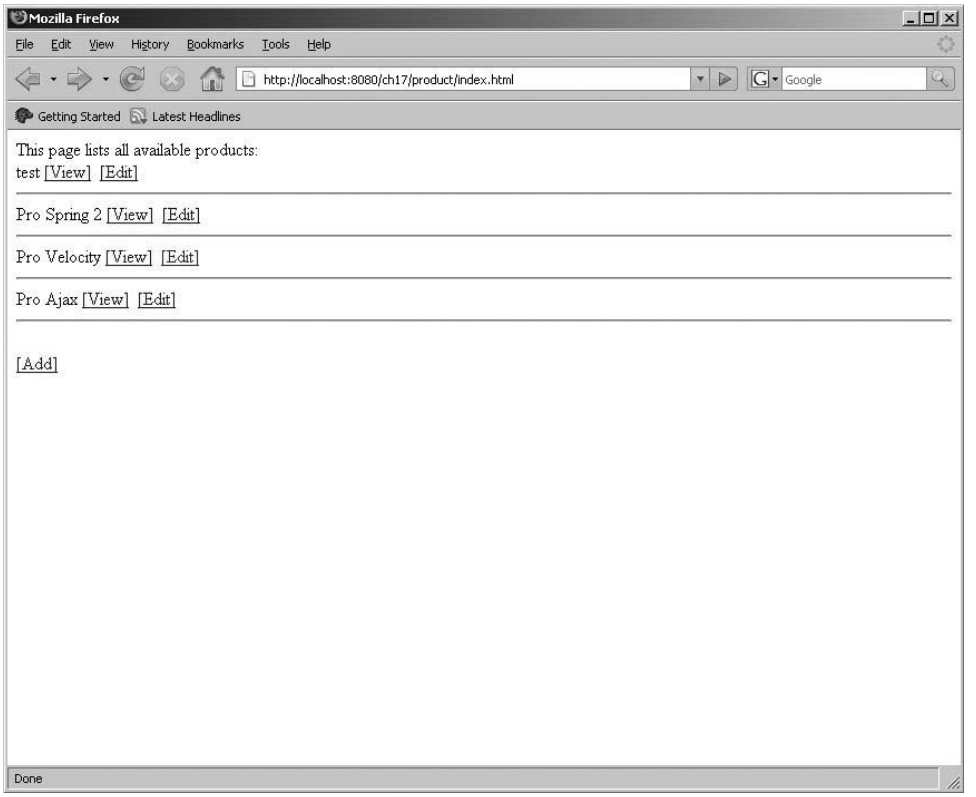

Figure 17-5. *The English version of the site*

If we now change the preferred language to Spanish, the view resolver is going to create an instance of `index_ES`, which is a `JstlView`, and its URL property points to `/WEB-INF/views/en_ES/products/index.jsp`. This renders the page shown in Figure 17-6.

Using view resolvers rather than manually instantiating the views has the obvious benefit of simpler configuration files, but it also reduces the application's memory footprint. If we defined each view as a Spring bean, it would be instantiated on application start; if we use view resolvers, however, the view will be instantiated and cached on first request.

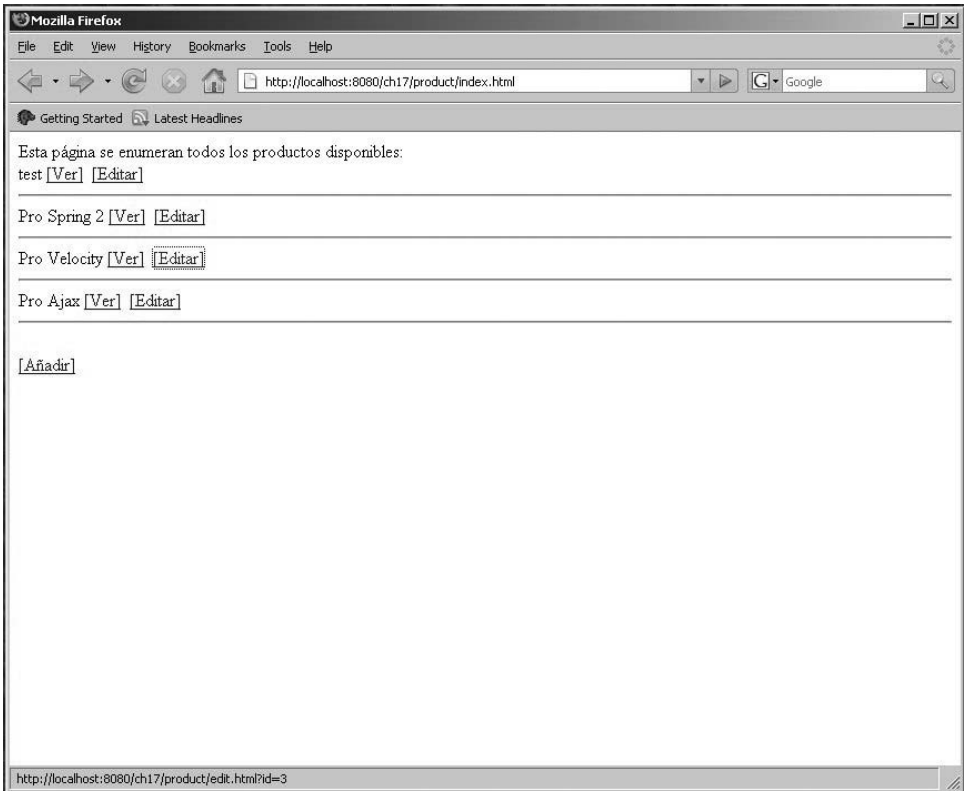

Figure 17-6. *The Spanish version of the site*

Using Localized Messages

Before we can discuss using locales in Spring web applications, we must take a look at how Spring resolves the actual text for messages to be displayed either in the `spring:message` tag (covered in more depth in the "Using Locales" section of this chapter) or as part of the validation process. The core interface, `MessageSource`, uses the `MessageSourceResolvable` interface to find the message key to be displayed. Spring comes with two implementations of the `MessageSource` interface: `ResourceBundleMessageSource` and `ReloadableResourceBundleMessageSource`. Both implementations use a standard properties file to load the messages, but `ReloadableResourceBundleMessageSource` can reload the contents of the properties file if it detects a change, eliminating the need to restart the application when the contents of the properties file are updated.

The default bean name Spring looks up is `messageSource`; our definition of this bean is shown in Listing 17-23.

Listing 17-23. *messageSource Bean Definition*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
...
    <bean id="messageSource"
        class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basename"><value>messages</value></property>
    </bean>
</beans>
```

Using Locales

We have already discussed the internationalization support in `ResourceBundleViewResolver`; let's now take a look at how things work under the hood. Spring uses the `LocaleResolver` interface to intercept the request and calls its methods to get or set the `Locale`. The implementations of `LocaleResolver`, each with its particular uses and properties, are shown in Table 17-5.

Table 17-5. *LocaleResolver Implementations*

Implementation	Description
`AcceptHeaderLocaleResolver`	This locale resolver returns the locale based on the accept-language header sent by the user agent to the application. If this resolver is used, the application will automatically appear in the user's preferred language (if we took the time to implement that language). If users wish to switch to another language, they have to change their browser settings.
`CookieLocaleResolver`	This locale resolver uses a cookie on the client's machine to identify the locale. This allows users to specify the language they want the application to appear in without changing their browser settings. It is not hard to imagine that users in Prague could have an English web browser, yet expect to see the application in Czech. Using this locale resolver, we can store the locale settings using users' browser cookie stores.
`FixedLocaleResolver`	The fixed locale resolver is a very simple implementation of `LocaleResolver` that always returns one configured locale.
`SessionLocaleResolver`	This resolver works very much like `CookieLocaleResolver`, but the locale settings are not persisted in a cookie and are lost when the session expires.

Using Themes

In addition to providing the application's views in users' languages, themes can be used to further improve the user experience. A theme is usually a collection of style sheets and images that are embedded into the rendered output. Spring also provides a tag library that you can use to enable theme support in your JSP pages. Listing 17-17 shows a typical directory structure of a Spring web application, with specific directory for themes.

As you can see from Figure 17-7, we have added a `themes` directory and created two new properties files: `cool.properties` and `default.properties`. Each line in these properties files specifies the location of a static theme resource. The key is the name of the resource, and the value is the path to the resource file, as shown in Listing 17-24. The path is relative to the context root of the web application.

Figure 17-7. *Directory and file structure for themes*

Listing 17-24. *The cool.properties File*

```
css=/themes/cool/main.css
```

We can use this definition in a JSP page using the Spring tag library, as shown in Listing 17-25.

Listing 17-25. *index.jsp*

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

<html>
    <head>
        <c:set var="css"><spring:theme code="css"/></c:set>
        <c:if test="${not empty css}">
            <link rel="stylesheet" href="<c:url value="${css}"/>" type="text/css" />
        </c:if>
    </head>
<body>
```

```
This page lists all available products:<br>
<c:forEach items="${products}" var="product">
    <c:out value="${product.name}"/>
    <a href="view.html?productId=
        <c:out value="${product.productId}"/>">[View]</a> 
    <a href="edit.html?productId=
        <c:out value="${product.productId}"/>">[Edit]</a> <br>
    <hr>
</c:forEach><br>
<a href="edit.html">[Add]</a>
</body>
</html>
```

Finally, we need to modify the Spring application context and add a `themeResolver` bean, as shown in Listing 17-26.

Listing 17-26. *The themeResolver Bean Definition*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
...>
    <bean id="themeResolver"
         class="org.springframework.web.servlet.theme.FixedThemeResolver">
        <property name="defaultThemeName"><value>cool</value></property>
    </bean>
    <!-- other beans as usual -->
</beans>
```

This application context file specifies that the application is going to be using `FixedThemeResolver` with `defaultThemeName` set to `cool`, which is going to be loaded from the `cool.properties` file in the root of the classpath because we have set the `defaultThemeName` to `cool`.

Themes can contain not only style sheets but references to any kind of static content, such as images and movies. Therefore, themes must support internationalization, as images may contain text that needs to be translated into other languages. The internationalization support in theme resolvers works exactly the same way as internationalization support in `ResourceBundleViewResolver`. The theme resolver will try to load `theme_<LID>.properties`, where LID is the locale identifier (`EN`, `ES`, `CS`, etc.). If the properties file with the LID does not exist, the resolver will try to load properties file without it.

Just like the `ViewResolvers` and the `LocaleResolvers`, there are several implementations of `ThemeResolvers`; they are shown in Table 17-6.

Table 17-6. *ThemeResolver Implementations*

ThemeResolver	Description
`CookieThemeResolver`	Allows the theme to be set per user and stores the theme preferences by storing a cookie on the client's computer
`FixedThemeResolver`	Returns one fixed theme, which is set in the bean's `defaultThemeName` property
`SessionThemeResolver`	Allows the theme to be set per user's session, so the theme is not persisted between sessions

Adding support for themes can give your application an extra visual kick with very little programming effort.

Command Controllers

Until now, we have been talking about ways to get the data to the user based on the request parameters and how to render the data passed from the controllers. A typical application also gathers data from the user and processes it. Spring supports this scenario by providing command controllers that process the data posted to the controllers. Before we can start discussing the various command controllers, we must take a more detailed look at the concept of command controllers.

The command controller allows a command object's properties to be populated from the `<FORM>` submission. Because the command controllers work closely with the Spring tag libraries to simplify data validation, they are an ideal location to perform all business validation. As validation occurs on the server, it is impossible for the users to bypass it. However, you should not rely on the web tier to perform all validation, and you should revalidate in the business tier.

On the technical side, the command controller implementations expose a command object, which is (in general) a domain object. These are the possible implementations:

- `AbstractCommandController`: Just like `AbstractController`, `AbstractCommandController` implements the `Controller` interface. This class is not designed to actually handle HTML `FORM` submissions, but it provides basic support for validation and data binding. You can use this class to implement your own command controller, in case the Spring controllers are insufficient to your needs.
- `AbstractFormController`: The `AbstractFormController` class extends `AbstractCommandController` and can actually handle HTML `FORM` submissions. In other words, this command controller will process the values in `HttpServletRequest` and populate the controller's command object. The `AbstractFormController` also has the ability to detect duplicate form submission, and it allows you to specify the views that are to be displayed in the code rather than in the Spring context file. This class has the useful method `Map referenceData()`, which returns the model (`java.util.Map`) for the form view. In this method, you can easily pass any parameter that you would use on the form page (a typical example is a `List` of values for an HTML `SELECT` field).
- `SimpleFormController`: This is the most commonly used command controller to process HTML `FORM` submissions. It is also designed to be very easy to use; you can specify the views to be displayed for the initial view and a success view; and you can set the command object you need to populate with the submitted data.
- `AbstractWizardFormController`: As the name suggests, this command controller is useful for implementing wizard-style sets of pages. This also implies that the command object must be kept in the current `HttpSession`, and you need to implement the `validatePage()` method to check whether the data on the current page is valid and whether the wizard can continue to the next page. In the end, the `AbstractWizardFormController` will execute the `processFinish()` method to indicate that it has processed the last page of the wizard process and that the data is valid and can be passed to the business tier. Instead of this command controller, Spring encourages the use of Spring Web Flow, which offers the same functionality with greater flexibility.

Using Form Controllers

Now that you know which form controller to choose, let's create an example that will demonstrate how a form controller is used. We will start with the simplest form controller implementation and move on to add validation and custom formatters.

The most basic controller implementation will extend `SimpleFormController`, override its `onSubmit()` method, and provide a default constructor.

Listing 17-27. *ProductFormController Implementation*

```
public class ProductFormController extends SimpleFormController {

    public ProductFormController() {
        setCommandClass(Product.class);
 setCommandName("product");
        setFormView("products-edit");
    }

    protected ModelAndView onSubmit(HttpServletRequest request,
        HttpServletResponse response, Object command,
        BindException errors) throws Exception {
        System.out.println(command);

        return new ModelAndView("products-index-r");
    }

}
```

The `ProductFormController`'s constructor defines that the command class is `Product.class`; this means that the object this controller creates will be an instance of `Product`.

Next, we override the `onSubmit()` method, which is going to get called when the user submits the form. The command object will have already passed validation so passing it to the business layer is safe, if doing so is appropriate. The `onSubmit()` method will return a `products-index-r` view, which is a `RedirectView` that will redirect to the `products/index.html` page. We need the `products-index-r` view because we want the `ProductController.handleIndex()` method to take care of the request.

Finally, the call to `setFormView()` specifies the view that will be used to display the form. In our case, it will be a JSP page, as shown in Listing 17-28.

Listing 17-28. *The edit.jsp Page*

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

<html>
<head>
    <c:set var="css"><spring:theme code="css"/></c:set>
        <c:if test="${not empty css}">
            <link rel="stylesheet" href="<c:url value="${css}"/>" type="text/css" />
        </c:if>
</head>
<body>
<form:form commandName="product" action="edit.html" method="post">
<input type="hidden" name="productId"
    value="<c:out value="${product.productId}"/>">
<table>
    <tr>
        <td>Name</td>
        <td><form:input path="name" />

        </td>
    </tr>
    <tr>
        <td>Expiration Date</td>
        <td><form:input path="expirationDate" />
```

```
        </td>
    </tr>
    <tr>
        <td></td>
        <td><input type="submit"></td>
    </tr>
</table>
</form:form>
</body>
</html>
```

The form tag library is provided in Spring version 2.0 or higher. It allows us to pass the values from the form in a very simple way. The form tag renders HTML's `<form>` tag and exposes its path for inner tags binding. The method and action attributes are the same as the HTML `<form>` tag's, and the `commandName` attribute sets the command name as it is set in the form controller (`setCommandName(String name)` method). If omitted, the `commandName` value defaults to `command`. The input tags render the HTML `<input>` tag of type text, with value as the bound property of the form object.

The form tag library is explained in more depth later in this chapter.

The last things we need to do are modify the application context file and the `productFormController` bean and add mapping for `/product/edit.html` to the form controller.

Listing 17-29. *The ProductFormController Definition and URL Mapping*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
...>
    <bean id="publicUrlMapping"
         class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <value>
                /index.html=indexController
                /product/index.html=productController
                /product/view.html=productController
               /product/edit.html=productFormController
            </value>
        </property>
    </bean>

    <!-- Product -->
    <bean id="productFormController"
        class="com.apress.prospring2.ch17.web.product.ProductFormController">
    </bean>
    <!-- other beans as usual -->
</beans>
```

As you can see, there is nothing unusual about the new definitions in the Spring application context file. If we now navigate to `http://localhost:8080/ch17/product/edit.html`, we will find a typical web page with a form to enter the data, as shown in Figure 17-8.

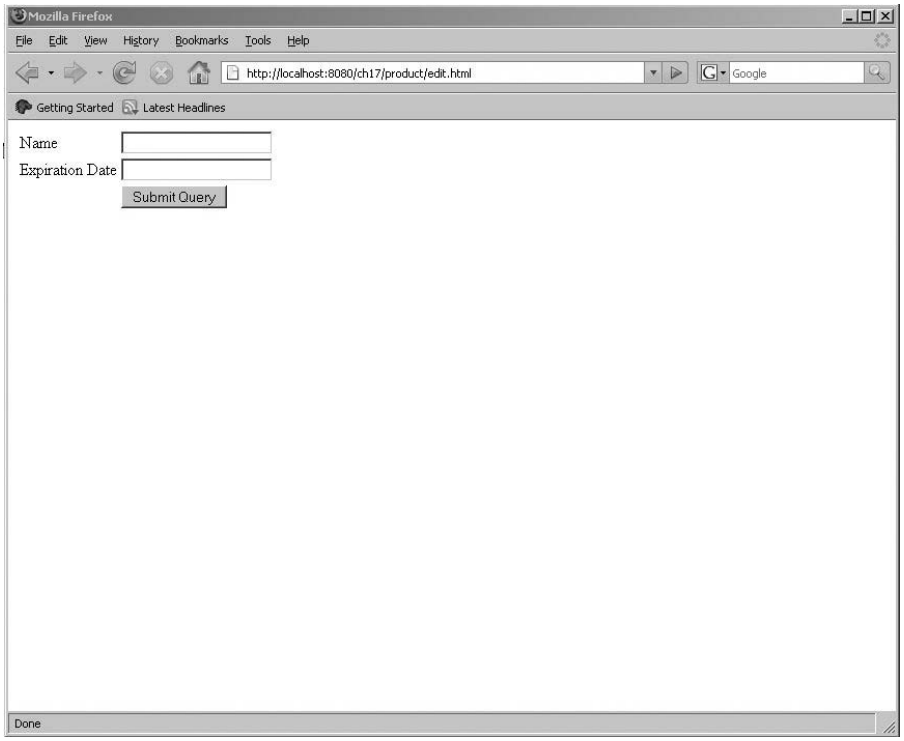

Figure 17-8. *The product editing form*

Unfortunately, the `expirationDate` property is of type `Date`, and Java date formats are a bit difficult to use. You cannot expect users to type **Sun Oct 24 19:20:00 BST 2004** for a date value. To make things a bit easier for the users, we will make our controller accept the date as a user-friendly string, for example, "24/10/2004". To do this, we must use an implementation of the `PropertyEditor` interface, which gives users support to edit bean property values of specific types. In this example, we will use `CustomDateEditor`, which is the `PropertyEditor` for `java.util.Date`. Now, we have to let our controller know to use our date editor (`CustomDateEditor`) to set properties of type `java.util.Date` on the command object. We will do this by registering `CustomDateEditor` for the `java.util.Date` class to the `ServletRequestDataBinder`. `ServletRequestDataBinder` extends the `org.springframework.validation.DataBinder` class to bind the request parameters to the JavaBean properties. To register our date editor, we are going to override the `initBinder()` method (see Listing 17-30).

Listing 17-30. *CustomEditor Registration in ProductFormController*

```
public class ProductFormController extends SimpleFormController {

    // other methods omitted for clarity

    protected void initBinder(HttpServletRequest request,
        ServletRequestDataBinder binder) throws Exception {
        SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, null,
```

```
            new CustomDateEditor(dateFormat, false));
    }
}
```

The newly registered custom editor will be applied to all `Date.class` values in the requests processed by `ProductFormController`, and the values will be parsed as `dd/MM/yyyy` values, thus accepting "24/10/2004" instead of "Sun Oct 24 19:20:00 BST 2004" as a valid date value.

There is one other important thing missing from our `Controller`—validation. We do not want to allow users to add a product with no name. To implement validation, Spring provides `Validator` interface, which we have implemented in Listing 17-31. Next, we need to register the `ProductValidator` bean as a Spring-managed bean and set the `ProductFormController`'s validator property to the `productValidator` bean (Listing 17-32 shows the bean definition).

Listing 17-31. *ProductValidator Bean Implementation*

```
public class ProductValidator implements Validator {

    public boolean supports(Class clazz) {
        return clazz.isAssignableFrom(Product.class);
    }

    public void validate(Object obj, Errors errors) {
        Product product = (Product)obj;
        if (product.getName() == null || product.getName().length() == 0) {
            errors.rejectValue("name", "required", "");
        }
    }

}
```

This `Validator` implementation will add a validation error with `errorCode` set to `required`. This code identifies a message resource, which needs to be resolved using a `messageSource` bean. The `messageSource` bean allows externalization of message strings and supports internationalization as well. The rules for creating internationalized messages are exactly the same as rules for creating internationalized views and themes, so we will show only the final application context file in Listing 17-32, without going into the details of the `messages.properties` and `messages_CS.properties` files.

Listing 17-32. *The ProductFormController Definition and URL Mapping*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
...>
    <bean id="messageSource"
        class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basename" value="messages/>
    </bean>

    <bean id="productValidator"
        class="com.apress.prospring2.ch17.business.validators.ProductValidator"/>

    <!-- Product -->
    <bean id="productFormController"
        class="com.apress.prospring2.ch17.web.product.ProductFormController">
        <property name="validator" ref="productValidator"/>
    </bean>
```

```
    <!-- other beans as usual -->
</beans>
```

Spring provides the convenience utility class `ValidationUtils` for easier, more intuitive validation. Using `ValidationUtils`, we can improve our `ProductValidator` as shown in Listing 17-33.

Listing 17-33. *ProductValidator Bean Implementation*

```
public class ProductValidator implements Validator {

    public boolean supports(Class clazz) {
        return clazz.isAssignableFrom(Product.class);
    }

    public void validate(Object obj, Errors errors) {
        Product product = (Product)obj;
       ValidationUtils.rejectIfEmptyOrWhitespace(errors, ➥
                    "name", "required", "Field is required.");
    }

}
```

If we want validation errors to be displayed on the form page to the user, we must edit the `edit.jsp` page as shown in Listing 17-34.

Listing 17-34. *The edit.jsp Page with Errors*

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

<html>
<head>
    <c:set var="css"><spring:theme code="css"/></c:set>
        <c:if test="${not empty css}">
            <link rel="stylesheet" href="<c:url value="${css}"/>" type="text/css" />
        </c:if>
</head>
<body>
<form:form commandName="product" action="edit.html" method="post">
<input type="hidden" name="productId"
    value="<c:out value="${product.productId}"/>">
<table>
    <tr>
        <td>Name</td>
        <td><form:input path="name" />
                  <form:errors path="name" />
        </td>
    </tr>
    <tr>
        <td>Expiration Date</td>
        <td><form:input path="expirationDate" />
                <form:errors path="expirationDate" />
        </td>
    </tr>
    <tr>
        <td></td>
        <td><input type="submit"></td>
```

```
    </tr>
</table>
</form:form>
</body>
</html>
```

If we rebuild and redeploy the application, go to the `product/edit.html` page, and try to submit the form with a valid expiration date but no product name, we will see an error message in the appropriate language, as shown in Figure 17-9.

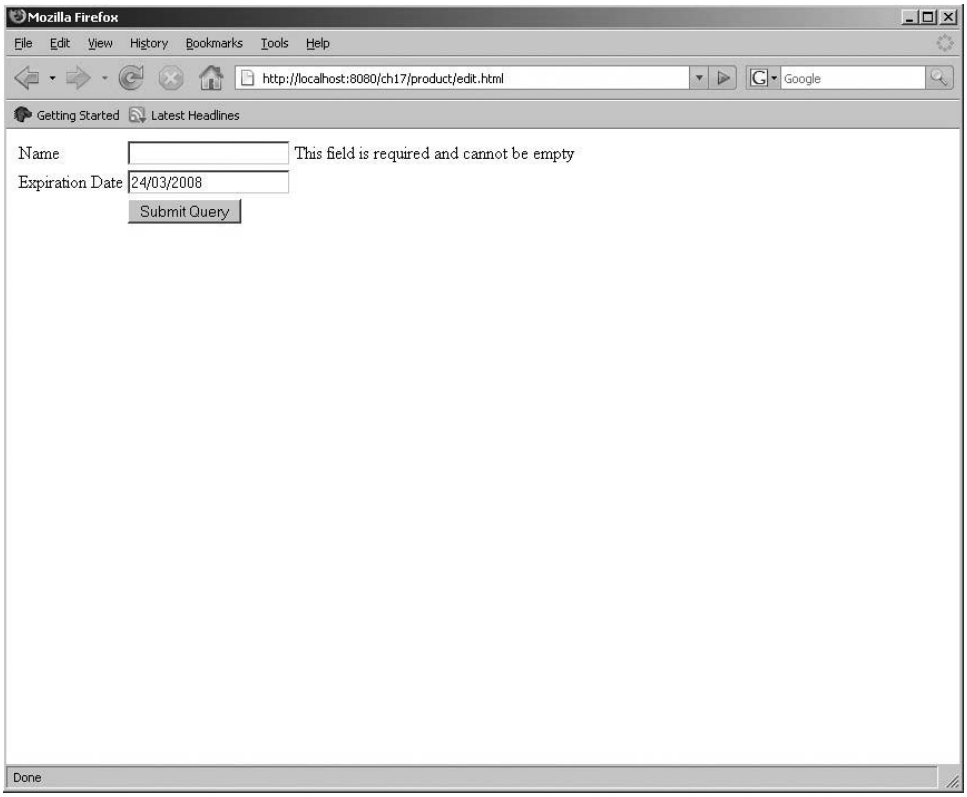

Figure 17-9. *The edit page with validation errors*

You now know how to get new data from users, but in a typical application, you have to deal with edits as well. There must be a way to prepare the command object so it contains data retrieved from the business layer. Typically, this means that the request to the edit page will contain a request parameter that specifies the object identity. The object will then be loaded in a call to the business layer and presented to the user. To do this, override the `formBackingObject()` method.

Listing 17-35. *Overriding the formBackingObject() Method*

```
public class ProductFormController extends SimpleFormController {

    // other methods omitted for clarity
```

```
        protected Object formBackingObject(HttpServletRequest request) ➥
                        throws Exception {
        Product command = new Product();
        long productId = ServletRequestUtils.getLongParameter(request, "id", 0);
        if (id != 0) {
            // load the product
            command.setId(id);
            command.setName("loaded");
        }

        return command;
}
```

And behold: when we make a request to `edit.html` with the request parameter product ID set to 2, the command object's `name` property will be set to `loaded`. Of course, instead of creating an instance of the `Product` object in the controller, we would use a business layer to pass the object identified by the ID.

The other controllers follow the same rules for processing form submission and validation, and the Spring sample applications explain the uses of other controllers, so there is no need to describe them in further detail.

Exploring the AbstractWizardFormController

As we stated in the previous section, Spring advocates using Spring Web Flow for implementing wizard-style forms. However, since `AbstractWizardFormController` is the predecessor of Spring Web Flow, we believe familiarity with this `Controller` implementation would be useful. To demonstrate how to use `AbstractWizardFormController`, we are going to begin with a simple set of JSP pages: `step1.jsp`, `step2.jsp`, and `finish.jsp`. The code of these JSP pages (see Listing 17-36) is not too different from the code used in the `edit.jsp` page from the previous section (see Listing 17-34).

Listing 17-36. *Code for the step1.jsp, step2.jsp, and finish.jsp Pages*

```
// step1.jsp
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core_rt" %>
<%@taglib prefix="spring" uri="http://www.springframework.org/tags"%>

<html>
<head>
    <c:set var="css"><spring:theme code="css"/></c:set>
       <c:if test="${not empty css}">
            <link rel="stylesheet" href="<c:url value="${css}"/>" ➥
                  type="text/css" />
       </c:if>
</head>
<body>
<form action="wizard.html?_target1" method="post">
<input type="hidden" name="_page" value="0">
<table>
    <tr>
        <td>Name</td>
        <td><spring:bind path="command.name">
                <input name="name" value="<c:out value="${status.value}"/>">
                <span class="error"><c:out value="${status.errorMessage}"/></span>
            </spring:bind>
        </td>
```

```
    </tr>
    <tr>
        <td></td>
        <td><input type="submit" value="Next"></td>
    </tr>
</table>
</form>
</body>
</html>

// step2.jsp
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core_rt" %>
<%@taglib prefix="spring" uri="http://www.springframework.org/tags"%>

<html>
<head>
    <c:set var="css"><spring:theme code="css"/></c:set>
       <c:if test="${not empty css}">
          <link rel="stylesheet" href="<c:url value="${css}"/>➥
                " type="text/css" />
       </c:if>
</head>
<body>
<form action="wizard.html?_target2" method="post">
<input type="hidden" name="_page" value="1">
<table>
    <tr>
        <td>Expiration Date</td>
        <td><spring:bind path="command.expirationDate">
                <input name="expirationDate" ➥
                       value="<c:out value="${status.value}"/>">
                <span class="error"><c:out value="${status.errorMessage}"/></span>
            </spring:bind>
        </td>
    </tr>
    <tr>
        <td></td>
        <td><input type="submit" value="Next"></td>
    </tr>
</table>
</form>
</body>
</html>

// finish.jsp
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core_rt" %>
<%@taglib prefix="spring" uri="http://www.springframework.org/tags"%>

<html>
<head>
    <c:set var="css"><spring:theme code="css"/></c:set>
       <c:if test="${not empty css}">
            <link rel="stylesheet" href="<c:url value="${css}"/>" ➥
                  type="text/css" />
       </c:if>
</head>
<body>
```

```
<form action="wizard.html?_finish" method="post">
<input type="hidden" name="_page" value="2">
<table>
    <tr>
        <td>Register now?</td>
        <td><c:out value="${command}"/></td>
    </tr>
    <tr>
        <td></td>
        <td><input type="submit" value="Next"></td>
    </tr>
</table>
</form>
</body>
</html>
```

As you can see, the `step1.jsp` and `step2.jsp` pages simply populate the `name` and `expirationDate` properties of the command object, which is an instance of the `Product` domain object.

The `AbstractWizardFormController` uses several request parameters to control the page flow of the wizard. All possible parameters are summarized in Table 17-7.

Table 17-7. *Page Flow Request Parameters*

Parameter	Description
`_target<value>`	The value is a number that specifies the `pages[]` property's index that the controller should go to when the current page is submitted. The current page must be valid, or the `allowDirtyForward` or `allowDirtyBack` properties must be set to `true`.
`_finish`	If this parameter is specified, the `AbstractWizardFormController` will invoke the `processFinish()` method and remove the command object from the session.
`_cancel`	If this parameter is specified, the `AbstractWizardFormController` will invoke the `processCancel()` method, which, if not overridden, will just remove the command object from the session. If you choose to override this method, do not forget to call the `super()` method or remove the command object from the session yourself.
`_page`	This parameter (usually specified as `<input type="hidden" name="_page" value="">`) specifies the index of the page in the `pages` property.

Now that we have the JSP pages that form the wizard steps, we need to implement the `RegistrationController` as a subclass of the `AbstractWizardFormController`, as shown in Listing 17-37.

Listing 17-37. *RegistrationController Implementation*

```
package com.apress.prospring2.ch17.web.registration;

public class RegistrationController extends AbstractWizardFormController {

    public RegistrationController() {
        setPages(new String[] {"registration-step1", "registration-step2",
            "registration-finish"});
        setSessionForm(true);
        setCommandClass(Product.class);
    }
```

```
    protected ModelAndView processFinish(HttpServletRequest request,
        HttpServletResponse response, Object command,
        BindException errors) throws Exception {
        Product product = (Product)command;

        System.out.println("Register " + product);
        return null;
    }

    protected void initBinder(HttpServletRequest request,
        ServletRequestDataBinder binder) throws Exception {
        SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, null,
            new CustomDateEditor(dateFormat, false));
    }

    protected void validatePage(Object command, Errors errors, int page,
        boolean finish) {
        getValidator().validate(command, errors);
    }
}
```

The code shown represents almost the simplest implementation of the `AbstractWizardFormController` subclass. Technically, all we have to implement is the `processFinish()` method, but in our case, we also needed to register a custom editor for the `Date` class. Finally, we wanted set the `commandClass` property to `Product.class`. We could have set the `pages` and `sessionForm` properties in the bean definition, which is shown in Listing 17-38, but we have decided to set the properties in the constructor.

Listing 17-38. *The RegistrationController Bean and URL Mappings*

```
<beans xmlns="http://www.springframework.org/schema/beans"
       ...>
    <bean id="publicUrlMapping"
        class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="interceptors">
            <list>
                <ref local="bigBrotherHandlerInterceptor"/>
            </list>
        </property>
        <property name="mappings">
            <value>
                <!-- other  omitted -->
                /registration/wizard.html=registrationController
            </value>
        </property>
    </bean>
    <bean id="registrationController"
        class="com.apress.prospring.ch17.web.registration.RegistrationController">
        <property name="validator"><ref bean="productValidator"/></property>
    </bean>
</beans>
```

Notice that we have not created mappings for the `step1.jsp`, `step2.jsp`, and `finish.jsp` pages; instead, we have only created a single mapping for `/registration/wizard.html`, which is handled by the `registrationController` bean. We also set the `validator` property of the `registrationController` to the `productValidator` bean. We use the `validator` property in the `validatePage()` method to show that we can validate each page. The implementation we have chosen is exactly the same as the default implementation in `AbstractWizardFormController`, but if we wanted to, we could allow the user to move to the next page. The `AbstractWizardFormController` performs the validation before calling the `processFinish()` method, so there is no way to avoid validation and skipping validation on certain pages is safe—the command object in the `processFinish()` method is guaranteed to be valid.

■ **Note** The command object will be valid only if we have supplied an appropriate `Validator` implementation.

The explanation of the `AbstractWizardFormController` we have offered here is quite simple, but it should give you a good starting point if you decide to use `AbstractWizardFormController` subclasses in your application.

File Upload

Spring handles file upload through implementations of the `MultipartResolver` interface. Out of the box, Spring comes with support for Commons FileUpload (`http://commons.apache.org/fileupload/`). By default, there is no default `multipartResolver` bean declared, so if you want to use the Commons implementation or provide your own implementation, you have to declare the `multipartResolver` bean in the Spring application context, as shown in Listing 17-39.

Listing 17-39. *MultipartResolver Declaration for Commons FileUpload*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
...>
    <bean id="multipartResolver"
        class="org.springframework.web.multipart.➥
            commons.CommonsMultipartResolver">
        <property name="maxUploadSize"> <value>100000</value> </property>
    </bean>

    <!-- other beans as usual -->
</beans>
```

Do not forget that you can only have one `multipartResolver` bean, so you have to choose which one to use when you declare the beans. Once the `multipartResolver` bean is configured, Spring will know how to handle multipart form-data-encoded requests; that means it will transform the form data into a `byte[]` array. To demonstrate that our newly configured `multipartResolver` works, we are going to create `ProductImageFormController` and `ProductImageForm` classes. The first one will extend `SimpleFormController` and handle image upload, while the second one is going to contain properties for the image name and contents. The `ProductImageForm` implementation is shown in Listing 17-40.

Listing 17-40. *ProductImageForm Implementation*

```
public class ProductImageForm {

    private String name;
    private byte[] contents;

    public byte[] getContents() {
        return contents;
    }

    public void setContents(byte[] contents) {
        this.contents = contents;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

There is nothing spectacular about this class; it is a simple Java bean that exposes the `name` and `contents` properties. The `ProductImageFormController` class's `initBinder()` method makes it much more interesting; Listing 17-41 shows this class.

Listing 17-41. *ProductImageFormController Implementation*

```
public class ProductImageFormController extends SimpleFormController {

    public ProductImageFormController() {
        super();
        setCommandClass(ProductImageForm.class);
        setFormView("products-image");
        setCommandName("product");
    }

    protected ModelAndView onSubmit(HttpServletRequest request,
        HttpServletResponse response, Object command,
        BindException errors) throws Exception {
        ProductImageForm form = (ProductImageForm)command;

        System.out.println(form.getName());
        byte[] contents = form.getContents();
        for (int i = 0; i < contents.length; i++) {
            System.out.print(contents[i]);
        }

        return new ModelAndView("products-index-r");
    }
```

```
    protected void initBinder(HttpServletRequest request,
        ServletRequestDataBinder binder) throws Exception {
        binder.registerCustomEditor(byte[].class,
            new ByteArrayMultipartFileEditor());
    }

}
```

The `ByteArrayMultipartResolver` class uses the `multipartResolver` bean from the application context to parse the contents of the multipart stream and return it as `byte[]` array, which is then processed in the `onSubmit()` method.

Be careful when coding the JSP page for the file upload: the most usual error is to forget the `enctype` attribute of the form element (as shown in Listing 17-42).

Listing 17-42. *The image.jsp Form*

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

<html>
<head>
    <c:set var="css"><spring:theme code="css"/></c:set>
        <c:if test="${not empty css}">
            <link rel="stylesheet" href="<c:url value="${css}"/>" type="text/css" />
        </c:if>
</head>
<body>
<form:form commandName="product" action="image.html" ➥
              method="post" enctype="multipart/form-data">
<input type="hidden" name="productId"
    value="<c:out value="${product.productId}"/>">
<table>
    <tr>
        <td>Name</td>
        <td><form:input path="name" />
                <form:errors path="name" />
        </td>
    </tr>
    <tr>
        <td>Expiration Date</td>
        <td><input name="contents" type="file" />
                <form:errors path="contents" />
        </td>
    </tr>
    <tr>
        <td></td>
        <td><input type="submit"></td>
    </tr>
</table>
</form:form>
</body>
</html>
```

As you can see, the JSP page is a plain HTML page, except for the `enctype` attribute. You must not forget to define this JSP page as a view in the `views.properties` file. Once you have defined the view and recompiled and redeployed the application, you should be able to use the file upload page at `products/image.html`, which is shown in Figure 17-10.

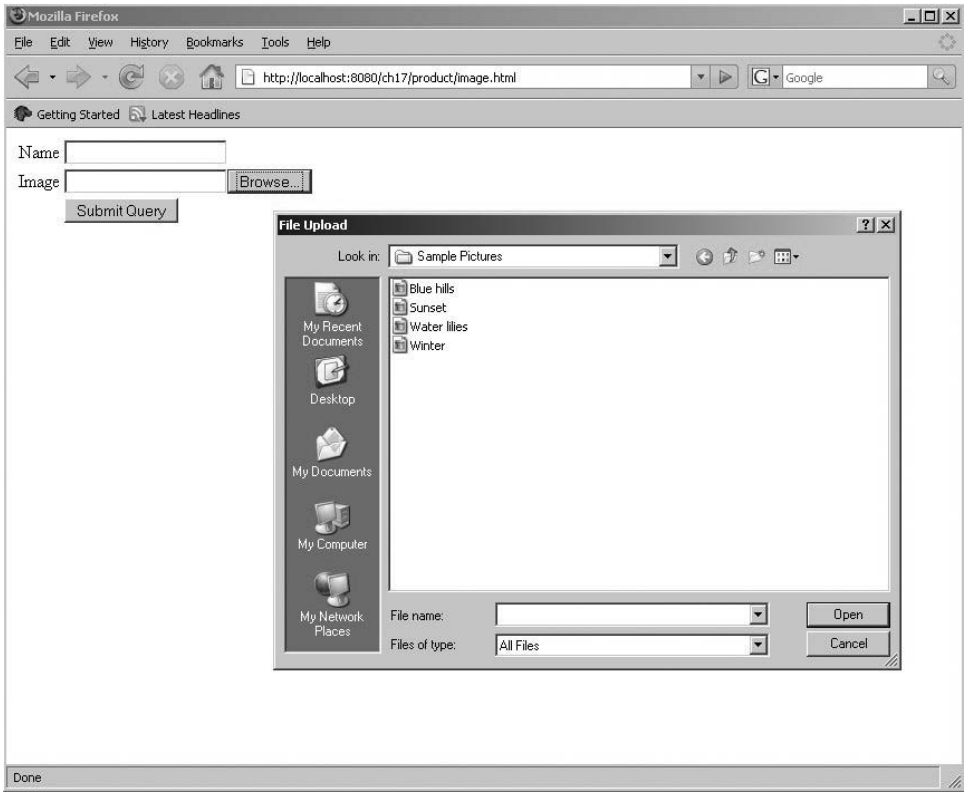

Figure 17-10. *File uploading*

Handling Exceptions

What will you do when something unexpected happens in your web application and an exception gets thrown? Showing the web user an ugly nested exception message isn't very nice. Fortunately, Spring provides `HandlerExceptionResolver` to make life easy when handling exceptions in your controllers. `HandlerExceptionResolver` provides information about what handler was executing when the exception was thrown, as well as many options to handle the exception before the request is forwarded to a user-friendly URL. This is the same end result as when using the exception mappings defined in `web.xml`.

Spring MVC provides one convenient, out-of-the-box implementation of `HandlerExceptionResolver`: `org.springframework.web.servlet.handler.SimpleMappingExceptionResolver`. This resolver enables you to take the class name of any exception that might be thrown and map it to a view name. It is easily configured in your Spring configuration files, as shown in Listing 17-43.

Listing 17-43. *ExceptionResolver Spring Configuration*

```
<bean id="exceptionResolver"
    class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
          <property name="defaultErrorView" value=""/>
          <property name="exceptionMappings">
                    <value>
```

```
                        java.lang.NullPointerException=nullPointerErrorView
                        javax.servlet.ServletException=servletErrorView
                    </value>
            </property>
</bean>
```

The `ExceptionMappings` property in our example maps `java.lang.NullPointerException` to a view named `nullPointerErrorView` and `javax.servlet.ServletException` to `servletErrorView`. The `defaultErrorView` property maps a view to which the request will be forwarded if the exception thrown is not mapped in the `exceptionMappings` property.

These views have to be defined in `views.properties`, `urlMapping`, and `Controller`, just like any other views (see Listing 17-44).

Listing 17-44. *Exception Views Definitions*

```
defaultErrorView.class=org.springframework.web.servlet.view.JstlView
defaultErrorView.url=/WEB-INF/views/en_GB/exception/default.jsp

nullPointerErrorView.class=org.springframework.web.servlet.view.JstlView
nullPointerErrorView.url=/WEB-INF/views/en_GB/exception/nullpointer.jsp

servletErrorView.class=org.springframework.web.servlet.view.JstlView
servletErrorView.url=/WEB-INF/views/en_GB/exception/servlet.jsp

<bean id="urlMapping" class="org.springframework.web.➥
             servlet.handler.SimpleUrlHandlerMapping">
       <property name="mappings">
            <value>

               /exception/*.html=exceptionController
                ...
           </value>
       </property>
</bean>
```

Listing 17-45 shows `Controller` implementation:

Listing 17-45. *ExceptionController Implementation*

```
public class ExceptionController extends MultiActionController{

          public ModelAndView defaultErrorHandler(HttpServletRequest request,
            HttpServletResponse response){
                    return new ModelAndView("defaultErrorView");
        }

       public ModelAndView nullPointerErrorHandler(HttpServletRequest request,
            HttpServletResponse response){
              return new ModelAndView("nullPointerErrorView");
      }

      public ModelAndView servletErrorHandler(HttpServletRequest request,
            HttpServletResponse response){
              return new ModelAndView("servletErrorView");
     }
}
```

Let's now look at a simple controller example. In Listing 17-46, you can see the `IndexController` implementation, which has only one method, `handleRequestInternal()`. Inside the method, we have only one line of code, `throw new NullPointerException()`. Every time this controller is invoked, `NullPointerException` will be thrown. You won't see an implementation like this in the real world; it's just a simplified example.

Listing 17-46. *IndexController Implementation*

```
public class IndexController extends AbstractController {

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

                throw new NullPointerException();

        }
```

As we would expect, after pointing the browser to `/index.html`, we will see the `NullPointerException` view message shown in Figure 17-11.

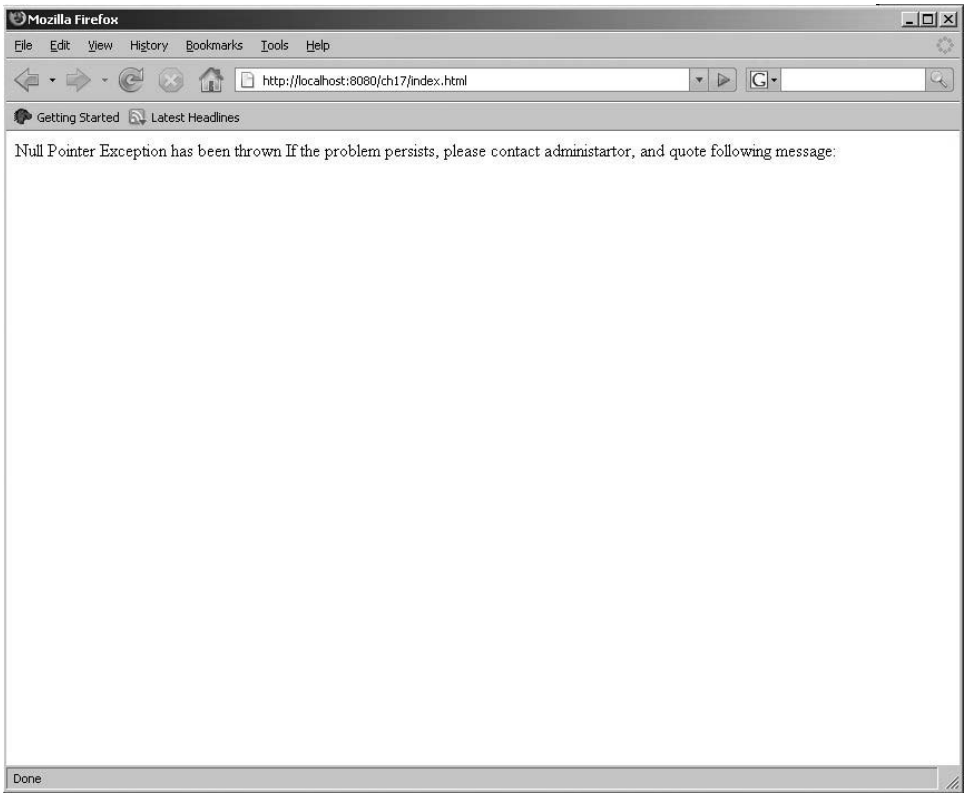

Figure 17-11. *Exception message in the browser*

This is functionally the same as the exception mapping feature from the Servlet API, but you could also implement finer grained mappings of exceptions from different handlers. Also, you can always implement your own `HandlerExceptionResolver`; all you have to do is implement its `public ModelAndView resolveException(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)` method. In example in the Listing 17-47, we create a custom exception resolver, that simply display an `Exception` message in the browser.

Listing 17-47. *Custom Implementation and Configuration of ExceptionResolver*

```
<bean id="exceptionResolver" class="com.apress.prospring2. ➥
        ch17.web.exception.ApressExceptionResolver"/>
...
public class ApressExceptionResolver implements HandlerExceptionResolver {

            public ModelAndView resolveException(HttpServletRequest request, ➥
                       HttpServletResponse response, Object handler, Exception ex) {

                   Map<String, Object> model = new HashMap<String, Object>();

                    model.put("message", ex.getMessage());

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

Note that only exceptions that occur before rendering the view are resolved by the `ExceptionResolver`. If your application throws an exception while rendering the view to the browser (for example if you have used a variable in your JSP pages without passing it to the model), you will still get a long ugly generic exception message. However, these situations are not supposed to happen in a web application, so this isn't a big complication after all.

Spring and Other Web Technologies

In the previous sections, we used JSP pages to generate output that is sent to the client's browser for rendering. We could naturally build the entire application using just JSP pages, but the application and JSP pages would probably become too complex to manage.

Even though JSP pages are very powerful, they can present a considerable processing overhead. Because Spring MVC fully decouples the view from the logic, the JSP pages should not contain any Java scriptlets. Even if this is the case, the JSP pages still need to be compiled, which is a lengthy operation, and their runtime performance is sometimes not as good as we would like. The Velocity templating engine from Apache (described in more detail in the "Using Velocity" section later in this chapter) is a viable alternative, offering much faster rendering times while not restricting the developer too much.

In addition to Velocity, we are going to explore the Tiles framework, which allows you to organize the output generated by the controllers into separate components that can be assembled together using a master template. This greatly simplifies the visual design of the application, and any changes to the overall layout of the output can be made very quickly with fewer coding mistakes and easier code management.

Further, we will explain how to use a PDF or an Excel spreadsheet as output instead of HTML. In some cases, HTML output is simply not suitable, and we have to use PDF or Excel output to present data to the user.

Using JSP

JSP is a tested technology that many developers are familiar with. Because JSP pages are compiled into Java classes and can contain Java code, the developers may be too tempted to move parts of the business logic into the JSP pages (in JSP technology, a code fragment that performs some business logic and is run at request-time processing is called scriptlet). Needless to say, that is a very bad thing that not only violates the MVC model two architecture but makes the application very difficult to maintain.

There is no way to stop developers from using Java scriptlets in JSP pages, but there should be no need to use them. If you ever find yourself in need of a scriptlet or creating a lot of logic using the standard JSTL (Java Standard Tag Library) tags, you should consider writing a custom tag.

Spring offers a list of specific tags that simplify access to Spring features in your JSP pages, as shown in Table 17-8.

Table 17-8. *Spring Tags*

Custom Tag	Description
`htmlEscape`	Sets a value indicating whether the output of other Spring tags should be escaped. If `true`, the HTML formatting strings (such as <, />, and &) will be replaced by their HTML visual codes.
`message`	Displays a message, identified by its code, that is retrieved from the Spring `messageSource` beans.
`theme`	Retrieves a value for the element defined in the current theme.
`hasBindErrors`	Binds to the errors object, enabling you to inspect any binding errors and display them in the HTML page.
`nestedPath`	Sets a nested path that is then used by the `bind` tag.
`bind`	Binds to an object and provides the object that allows you to access the bound value and any error messages.
`transform`	Transforms a variable to a string using the currently registered `PropertyEditor`. It can be used only with the bind tag.

Using the message Tag

Let's take a closer look at Spring custom tags, beginning with the `message` tag. First, let's create a `messageSource` bean definition in the application context file and create an appropriate `message.properties` file. Listing 17-48 shows the definition of the `messageSource` bean.

Listing 17-48. *messageSource Bean Definition*

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
...>
    <bean id="messageSource"
        class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basename"><value>messages</value></property>
    </bean>
    <!-- other beans omitted -->
</beans>
```

Next, we will create our trivial `message.properties` file, though if you want, you can create `message.properties` files for multiple languages. Listing 17-49 shows the `message.properties` file in English.

Listing 17-49. *message.properties File*

```
greeting=Hello <b>Spring</b> Framework
required=This field is required and cannot be empty
```

Next, we are going to create a `default.jsp` page that will use the Spring JSTL library to display the greeting message inside an `H1` element. Listing 17-50 shows the code for this page.

Listing 17-50. *Code for the default.jsp Page*

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

<html>
<head>
<title>Pro Spring</title>
</head>
<body>
<h1><spring:message code="greeting"/></h1>
</body>
</html>
```

The usage of the `message` tag is quite simple: it simply outputs the text that is looked up in the `messageSource` bean. This is obviously the simplest use of this tag, which can be modified by setting its attributes, shown in Table 17-9.

Table 17-9. *message Tag Attributes*

Attribute	Description
`code`	The code used to look up the message text in the `messageSource` bean.
`arguments`	Comma-separated list of arguments that will be passed to the `getMessage()` method of the `messageSource` bean.
`text`	Text that will be displayed if the entry code is not found in the `messageSource`. Internally, this is used as an argument in the `MessageSource.getMessage()` call.
`var`	Specifies the object that will be set to the value of the message.
`scope`	Specifies the scope to which the object specified in the `var` attribute is going to be inserted.
`htmlEscape`	Indicates whether the message tag should escape the HTML text. If not specified, the global value defined by the `htmlEscape` tag is used.

Using the theme Tag

The `theme` tag allows you to create themed pages. It uses the `themeResolver` bean to load all theme element definitions. To see the tag work, we need to make sure that the application context file contains a `themeResolver` bean definition, as shown in Listing 17-51.

Listing 17-51. *themeResolver Bean Definition*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/beans...>
    <bean id="themeResolver"
        class="org.springframework.web.servlet.theme.FixedThemeResolver">
        <property name="defaultThemeName"><value>cool</value></property>
```

```
    </bean>
    <!-- other beans omitted -->
</beans>
```

Typical usage of the `theme` bean is shown in the code fragment of the `default.jsp` page in Listing 17-52.

Listing 17-52. *theme Tag Usage*

```
<link rel="stylesheet" href="<spring:theme code="css"/>">
```

The `theme` tag has similar attributes to the `message` tag, allowing you to further control the values it generates. The attributes you can use are summarized in Table 17-10.

Table 17-10. *theme Tag Attributes*

Attribute	Description
`code`	The code used to look up the theme element text in the `themeResolver` bean.
`arguments`	Comma-separated list of arguments that will be passed to the `getMessage()` method of the `messageSource` bean.
`text`	Text that will be displayed if the entry code is not found in the `messageSource`. Internally, this is used as an argument in the `MessageSource.getMessage()` call.
`var`	Specifies the object that will be set to the value of the theme resource.
`scope`	Specifies the scope to which the object specified in the `var` attribute is going to be inserted.
`htmlEscape`	Indicates whether the `message` tag should escape the HTML text. If not specified, the global value defined by the `htmlEscape` tag is used.

In fact, the `theme` tag is a subclass of the `message` tag; the only difference is that the codes are looked up in a `themeResolver` rather than `messageSource` bean.

Using the hasBindErrors Tag

The `hasBindErrors` tag evaluates its body if the bean specified in its name attribute has validation errors. If there are no errors, the body of the tag is skipped. The usage of this tag in a JSP page is best demonstrated by the code fragment in Listing 17-53.

Listing 17-53. *The hasBindErrors Tag*

```
<spring:hasBindErrors name="command">
    There were validation errors: <c:out value="${errors}"/>
</spring:hasBindErrors>
```

Table 17-11 lists the attributes you can use to fine-tune the behavior of this tag.

Table 17-11. *hasBindErrors Attributes*

Attribute	Description
`name`	Name of the bean for which errors will be looked up.
`htmlEscape`	Indicates whether the message tag should escape the HTML text. If not specified, the global value defined by the `htmlEscape` tag is used.

Using the nestedPath Tag

The `nestedPath` tag is useful to include another object in the path resolution performed by the `bind` tag. The `nestedPath` tag will also copy objects in an existing `nestedPath` to the `nestedPath` it creates.

The only attribute of this tag is `path`, which specifies the path to the object that is to be included in the nested path. The code in Listing 17-54 shows the use of the `nestedPath` tag to add a user attribute to the nested path, giving the `bind` tag access to properties exposed by the `user` bean.

Listing 17-54. *nestedPath Tag Usage*

```
<spring:nestedPath name="user"/>
```

Using the bind Tag

The `bind` tag is the Spring tag used to simplify data entry and validation. In its simplest form, it looks up an object's property identified by the `path` attribute and exposes a new variable named status that holds the object's value and validation errors. This is the most common usage scenario and is shown in the code fragment in Listing 17-55.

Listing 17-55. *The bind Tag*

```
<spring:bind path="command.name">
    <input name="name" value="<c:out value="${status.value}"/>">
    <span class="error"><c:out value="${status.errorMessage}"/></span>
</spring:bind>
```

This code assumes there is a `command` object in the current scope, and that it exposes the `name` property. The `bind` tag will then create a status object and set its `status` property to an instance of the `BindStatus` object for the property identified by the `path` attribute. If there are any validation errors, the `errors` property will contain an instance of `Errors`; the `PropertyEditor` used to parse the value is in the `propertyEditor` property.

This tag can be further controlled through the attributes listed in Table 17-12.

Table 17-12. *bind Tag Attributes*

Attribute	Description
`path`	This attribute identifies the object and its property that will be used to create the `status`, `errors`, and `propertyEditor` properties.
`ignoreNestedPath`	If this is set to `true`, the `bind` tag will ignore any available nested paths as set by the `nestedPath` tag.
`htmlEscape`	This indicates whether the message tag should escape the HTML text. If not specified, the global value defined by the `htmlEscape` tag is used.

Using the transform Tag

The `transform` tag tag looks up the appropriate `PropertyEditor` for the object specified in the value attribute and uses it to output the `String` representation. You can use this attribute instead of the format JSTL tags to keep the formatting rules in Spring. The usage is shown in Listing 17-56.

Listing 17-56. *transform Tag Usage*

```
<spring:transform value="command.expirationDate"/>
```

If the `expirationDate` property of the command object is of `Date` class and if there is a `PropertyEditor` registered for `Date.class` with specified formatting rules, the `transform` tag will output the appropriately formatted date string. All attributes of the transform tag are listed in Table 17-13.

Table 17-13. *transform Tag Attributes*

Attribute	Description
`value`	Identified the value to be formatted
`var`	If set, the tag will set the formatted output to an object with the name set to the specified value.
`scope`	Specifies the scope into which the object specified in the `var` attribute is going to be inserted.
`htmlEscape`	Indicates whether the `message` tag should escape the HTML text. If not specified, the global value defined by the `htmlEscape` tag is used.

The Spring Form Tag Library

Spring 2.0 introduces a new tag library used to bind HTML form elements when working with JSP and Spring controllers. Each tag provides support for an HTML tag counterpart, which makes use easy and intuitive. All tags in the Form Tag library generate HTML 4.01–compliant and XHTML 1.0–compliant code.

Spring form tag library support is included in Spring MVC, so you'll be able to easily integrate it with your controllers.

To take advantage of this library, each of your JSP pages include the library declaration in the beginning of its file (see Listing 17-57).

Listing 17-57. *Form Tag Library Declaration*

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

You can choose the value of the `prefix` attribute as you like and use the tag library on your page with that prefix (in our example `"form"`).

The form Tag

The `form` tag is the main one in the library and is used to generate the HTML `form` tag, and all other tags in the Spring form tag library are nested tags of the `form` tag. The `form` tag will put the command object in the `PageContext` and make the command object available to inner tags.

Let's assume we have a domain object called `Customer`. It is a Java Bean with properties such as `username` and `fullName`. We will use it as the form-backing object of our form controller, which returns `form.jsp`. Listing 17-58 shows an example `form.jsp` file. This example just uses the provided binding of the `form` tag, and actual input fields are built using regular HTML input tags.

Listing 17-58. *The form Tag Example*

```
<form:form action="edit.html" commandName="customer">
    <table>
        <tr>
            <td>Username:</td>
            <td><input name="username" type="text" ➥
                                value="${customer.username}"/></td>
        </tr>
        <tr>
            <td>Full Name:</td>
            <td><input name="fullName" type="text" ➥
                               value="${customer.fullName}"/></td>
        </tr>
        <tr>
            <td>
                <input type="submit" value="Save" />
            </td>
        </tr>
    </table>
</form:form>
```

The `firstName` and `lastName` values are retrieved from the command object placed in the `PageContext` by the page controller.

The generated HTML looks like a standard form; see Listing 17-59.

Listing 17-59. *The HTML Generated by Listing 17-58*

```
<form method="POST" action="edit.html">
    <table>
      <tr>
          <td>Username:</td>
          <td><input name="firstName" type="text" value="janed"/></td>
          <td></td>
      </tr>
      <tr>
          <td>Full Name:</td>
          <td><input name="fullName" type="text" value="Jane Doe"/></td>
          <td></td>
      </tr>
      <tr>
          <td>
            <input type="submit" value="Save Changes" />
          </td>
      </tr>
    </table>
</form>
```

The `commandName` attribute of the form tag can be omitted; in that case, the command name will default to `"command"`.

This tag can be further controlled through the attributes. Main attributes are listed in Table 17-14.

Table 17-14. *form Tag Attributes*

Attribute	Description
`modelAttribute`	Identifies the name of the command object, same as `commandName`.
`acceptCharset`	A list of character encodings that are valid for the server. The list has to be a space- or comma-delimited set of values.
`htmlEscape`	Controls the HTML escaping of the rendered values.

■ **Note** The `form` tag can use certain regular HTML attributes, such as `encoding`, `action`, and `method`. At the same time, all standard HTML attributes and HTML event attributes can be used, such as `title`, `name`, `onclick`, and `onmouseover`. All HTML standard attributes and events can also be used on all nested attributes. For style purposes, all tags in the Spring form tag library have `cssStyle`, `cssClass`, and `cssErrorClass` attributes. The `cssStyle` tag is equivalent to standard HTML `style` attribute, and `cssClass` is equivalent to the standard HTML `class` attribute. The `cssErrorClass` attribute is used as a `class` attribute when the `errors` object is provided for given filed, for example, when a field has validation errors.

The example in Listings 17-58 and 17-59 uses just the `form` tag, but it uses normal HTML input tags in the form. We will now use the Spring form tag library input tag instead of HTML input tag.

The input Tag

The most common type of a field in forms is the `input` field. In Listing 17-59, we used a standard HTML `input` tag for `input` fields. Listing 17-60 shows how we can improve that example, using the `form:input` tag.

Listing 17-60. *An input Form Tag Example*

```
<form:form action="edit.html" commandName="customer">
    <table>
        <tr>
            <td>Username:</td>
            <td><form:input path="username" /></td>
        </tr>
        <tr>
            <td>Full Name:</td>
            <td><form:input path="fullName" /></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form:form>
```

The `input` fields will bind to paths: `customer.username` and `customer.fullName` respectively (as supplied via the `path` attribute). Table 17-15 shows some useful attributes for further controlling the `input` tag.

Table 17-15. *input Tag Attributes*

Attribute	Description
path	Identifies the object and its property that will be used to create the status and/or errors
htmlEscape	Controls the HTML escaping of the rendered values

Standard CSS style attributes, mentioned earlier, are also available for this tag.

The checkbox Tag

The checkbox tag renders an HTML input tag with type checkbox. There are three common scenarios for using the checkbox tag; these should meet all your check box needs. The scenarios are based on the type of the object that is bound to the field, depending weather the object is of the Boolean, Collection, or any other type.

When the bound value is of type java.lang.Boolean, the check box is marked as checked if the value of the property is true. Listing 17-61 shows Customer object with new property subscribed.

Listing 17-61. *Customer Bean Properties*

```
public class Customer{
    //other fields omitted for clarity
    private Boolean subscribed;
    public isSubscribed(){
        return subscribed;
}
    public setSubscribed(Boolean subscribed){
        this.subscribed=subscribed;

    }
}
```

In Listing 17-62, we have created appropriate form, with the subscribed property bound to the checkbox field.

Listing 17-62. *The checkbox Form Tag*

```
<form:form action="edit.html" commandName="customer">
    <table>
        <!-- Other properties, omitted for clarity-- >
        <tr>
            <td>Subscribe to newsletter?:</td>
            <td><form:checkbox path="customer.subscribed"/></td>
            <td> </td>
        </tr>

    </table>
</form:form>
```

If the customer command's subscribed property is set to true, the check box will be checked. Otherwise, it will remain unchecked.

The second usual scenario is that the bound property is of type java.util.Collection (or an array of any type). The check box is marked as checked if the property (Collection or array) contains the value.

To see an example of this property, we will add property `Set<String> categories` to the `Customer` object, referencing the product categories that the `Customer` is interested in (see Listing 17-63).

Listing 17-63. *Changed Customer Object with categories Property*

```
...
private Set<String> categories;
public getCategories(){
      return categories;
}
...
public setCategories(Set<String> categories){
      this.categories=categories;
}...
```

Listing 17-64 shows the form, with the check box bound to the `customer.categories` property.

Listing 17-64. *The Check Box Mapped to the Collection*

```
<form:form action="edit.html" commandName="customer">
    <table>
        <!-- Other properties, omitted for clarity-- >
        <tr>
            <td>Books:</td>
            <td><form:checkbox path="customer.categories" value="Books"/></td>
            <td> </td>
        </tr>
    </table>
</form:form>
```

This form will generate a check box that will be checked if the bound property `customer.categories` contains the value `Books`. If the `Books` value is not contained in the `categories` property of the `Customer`, the check box will not be checked.

The third scenario includes all other types of bound properties.

For this example, add the property `favouriteProduct` to the `Customer` bean, and add this property to the customer editing form (see Listings 17-65 and 17-66).

Listing 17-65. *Changed Customer Java Bean with favouriteProduct property*

```
private String favouriteProduct;
public getFavouriteProduct(){
      return favouriteProduct;
}
public setFavouriteProduct(String favouriteProduct){
      this.favouriteProduct=favouriteProduct;
}
```

Listing 17-66. *A Form with the checkbox Tag*

```
<form:form action="edit.html" commandName="customer">
    <table>
        <!-- Other properties, omitted for clarity-- >
        <tr>
            <td>Pro Spring 2 favourite artist:</td>
            <td><form:checkbox path="customer.favouriteProduct" ➥
                              value="Amy Winehouse CD"/></td>
            <td> </td>
```

```
        </tr>
    </table>
</form:form>
```

The `customer.favouriteProduct` property value is compared to the value attribute of the `checkbox` tag, and if they are the equal, the check box status will be `checked`; otherwise, the box will be unchecked.

The generated HTML will look exactly the same, no matter which approach we choose. In Listing 17-67, you can see an example of generated HTML for the `categories` property.

Listing 17-67. *Generated HTML for the categories Property*

```
<tr>
    <td>Categories:</td>
    <td>
        Books: <input name="customer.categories" type="checkbox" value="Books"/>
        <input type="hidden" value="1" name="_customer.categories "/>
        MP3 Players: <input name="customer.categories" type="checkbox" ➥
                                value="MP3 Players"/>
        <input type="hidden" value="1" name="_customer.categories"/>
        CDs: <input name="customer.categories" type="checkbox"
            value="CDs"/>
        <input type="hidden" value="1" name="_customer.categories"/>
    </td>
    <td> </td>
</tr>
```

The only uncommon things in this example are the hidden fields after every check box, which are there to overcome HTML restrictions within Spring. When a check box in an HTML page is not checked, its value will not be sent to the server as part of the HTTP request parameters when the form is submitted. But we need that information in the request, as otherwise, Spring form binding will not work (Spring would complain if a bound field were not found). We need a workaround to keep Spring's form data binding working. The `checkbox` tag follows the existing Spring convention of including a hidden parameter prefixed by an underscore (`_`) for each check box. This way, you will be telling Spring what check boxes were visible in the form and to bind data to properties of check box fields even if they are not part of the request (i.e., if they are not checked).

Table 17-16 lists some specific attributes for the `checkbox` tag.

Table 17-16. *checkbox Tag Attributes*

Attribute	Description
`path`	Identifies the object and its property that will be used to create the status and/or errors
`label`	Label to be displayed with the tag path property value
`htmlEscape`	Controls the HTML escaping of the rendered values

Standard CSS style attributes, mentioned earlier, are also available for this tag.

The checkboxes Tag

Spring provides the convenient `checkboxes` tag for rendering multiple check box fields.

This tag works in a similar way to the `checkbox` tag. The main addition is the `items` attribute, which accepts the `java.util.Collection` of values on which to build check boxes. Using the `items` attribute, you would render a check box for each value in the items' `Collection` in your HTML.

Let's use the example from Listing 17-61, the `Set<String> categories` property of the `Customer` object. We will create form page, using the `checkboxes` tag for binding categories (see Listing 17-68).

Listing 17-68. *checkbox Tag Mapped to Collection*

```
<form:form action="edit.html" commandName="customer">
    <table>
        <!-- Other properties, omitted for clarity-- >
        <tr>
            <td>Books:</td>
            <td><form:checkbox path="customer.categories" ➥
                            items="availableCategories"/></td>
            <td> </td>
        </tr>
    </table>
</form:form>
```

We will have to make the `availableCategories Collection` available as a model attribute by passing it to the model in our controller. Listing 17-69 shows how to achieve this.

Listing 17-69. *Controller Passing Available Check Boxes' Values*

```
Public class EditCustomerController extends SimpleFormController{
       //other methods omitted for clarity
    Map referenceData(HttpServletRequest request){
        Map<String, Object> model = new HashMap<String, Object>();
        List<String> availableCategories = new LinkedList<String>();
        availableCategories.add("Books");
        availableCategories.add("CDs");
        availableCategories.add("MP3Players");
        model.put("availableCategories", availableCategories);.
        return model;
    }
}
```

The generated HTML will have three check boxes rendered, with values passed via `availableCategories` list in the model (see Listing 17-70).

Listing 17-70. *Generated HTML for Check Boxes from Listing 17-68*

```
<tr>
    <td>Categories:</td>
    <td>
        Books: <input name="customer.categories" type="checkbox" value="Books"/>
        <input type="hidden" value="1" name="_customer.categories "/>
        MP3 Players: <input name="customer.categories" type="checkbox" ➥
                                        value="MP3 Players"/>
        <input type="hidden" value="1" name="_customer.categories"/>
        CDs: <input name="customer.categories" type="checkbox"
            value="CDs"/>
        <input type="hidden" value="1" name="_customer.categories"/>
    </td>
    <td> </td>
</tr>
```

The value of the `items` attribute can be the `Map` as well. In this case, the `Map` keys will be the values of rendered check boxes, and the `Map` value for the specific key will be the label for that check box.

Table 17-17 shows main attributes that can be used with the `checkboxes` tag.

Table 17-17. *checkboxes Tag Attributes*

Attribute	Description
`path`	Identifies the object and its property that will be used to create the status and/or errors.
`items`	The `Collection`, `Map`, or array that holds values for the rendered multiple check boxes.
`itemLabel`	The label to be displayed.
`itemValue`	The name of the property to be used as value for the check boxes.
`delimiter`	The HTML tag or any `String` to be used as delimiter between the check boxes. The default is `none`.
`element`	The HTML tag used to enclose every rendered check box. By default, all `<input type="checkbox">` tags will be enclosed with `<span>` HTML tags.
`htmlEscape`	Controls the HTML escaping of the rendered values.

Standard CSS style attributes, mentioned earlier, are also available for this tag.

The radiobutton Tag

The `radiobutton` tag renders an HTML `input` tag with type `radio`.

Listing 17-71 shows a typical radio button example. The `String` property `"answer"` is bound to multiple radio button tags. Only one of the radio buttons bound to the same property can be selected at any time.

Listing 17-71. *Radio Button Example*

```
private String answer;
//getters and setters
<tr>
    <td>Answer:</td>
    <td>A: <form:radiobutton path="answer" value="A"/> <br/>
        B: <form:radiobutton path="answer " value="B"/> </td>
    <td> </td>
</tr>
```

Table 17-18 shows some attributes that can be used with the `radiobutton` tag.

Table 17-18. *radiobutton Tag Attributes*

Attribute	Description
`path`	Identifies the object and its property that will be used to create the status and/or errors
`label`	Label to be displayed with the tag path property value
`htmlEscape`	Controls the HTML escaping of the rendered values

Standard CSS style attributes, mentioned earlier, are also available for this tag.

The radiobuttons Tag

The `radiobuttons` tag is convenient for rendering multiple radio buttons based on the values of the `Collection`, `Map`, or array of objects passed to this tag using the `items` attribute.

Listing 17-72 shows the form example using `radiobuttons` tag.

Listing 17-72. *A radiobuttons Tag Mapped to a Collection*

```
<form:form action="edit.html" >
    <table>
        <!-- Other properties, omitted for clarity-- >
        <tr>
            <td>Who is the murderer?</td>
            <td><form:radiobuttons path="command.answer" ➥
                        items="availableAnswers" delimiter="<br/>"/></td>m
            <td> </td>
        </tr>
    </table>
</form:form>
```

We will have to make `availableAnswers` available as a model attribute. Listing 17-73 gives example of how to do it using `Map`.

Listing 17-73. *Controller Passing Available radiobuttons Values*

```
Public class AnswerController extends SimpleFormController{
       //other methods omitted for clarity
    Map referenceData(HttpServletRequest request){
        Map<String, Object> model = new HashMap<String, Object>();
        Map<String, String> availableAnswers = new HashMap<String, String>();
        availableAnswers.put("A", "Butler");
        availableAnswers.put("B", "Gardener");
        availableAnswers.put("C", "Lawyer");
        availableAnswers.put("D", "Inspector");
        model.put("availableCategories", availableCategories);.
        return model;
    }
}
```

The map keys will be used for radio button values, and the `Map` entry for the specific key will be used for the radio button label. Listing 17-74 shows generated HTML.

Listing 17-74. *Generated HTML for the radiobuttons Tag from Listing 17-72*

```
<tr>
    <td>Who is the murderer:</td>
    <td>
        Butler: <input name="answer" type="radio" value="A"/><br/>
        Gardener: <input name="answer" type="radio" value="B"/><br/>
        Lawyer: <input name="answer" type="radio" value="C"/><br/>
        Inspector: <input name="answer" type="radio" value="D"/><br/>
</tr>
```

The `<br/>` HTML tag is used as delimiter as specified in the `tag` attribute. Similar to the `checkboxes` tag, items can be either `Map`, `Collection`, or array of objects. Table 17-19 shows attributes that can be used with the `radiobuttons` tag.

Table 17-19. *radiobuttons Tag Attributes*

Attribute	Description
path	Identifies the object and its property that will be used to create the status and/or errors.
items	The Collection, Map, or array that holds values for the rendered multiple radio buttons.
itemLabel	Label to be displayed.
itemValue	Name of the property to be used as value for the radio buttons.
delimiter	The HTML tag or any String to be used as delimiter between the radio buttons. The default is none.
element	The HTML tag used to enclose every rendered radio button. By default, all <input type="radio"> tags will be enclosed with HTML tags.
htmlEscape	Controls the HTML escaping of the rendered values.

Standard CSS style attributes, mentioned earlier, are also available for this tag.

The password Tag

The password tag renders an HTML input tag with type password. By default, the value of the field will not be shown in the browser. However, you can overcome this by setting the attribute showPassword to true. Listing 17-75 shows simple example.

Listing 17-75. *Password tag Example*

```
<tr>
    <td>Password:</td>
    <td>
        <form:password path="password" />
    </td>
</tr>
```

This tag can be further controlled using the attributes listed in Table 17-20.

Table 17-20. *password Tag Attributes*

Attribute	Description
path	Identifies the object and its property that will be used to create the status and/or errors.
showPassword	Determines if the password should be shown in the browser. The default is false.
htmlEscape	Controls the HTML escaping of the rendered values.

Standard CSS style attributes, mentioned earlier, are also available for this tag.

The select tag

The select tag renders an HTML select element and works in similar fashion to the checkboxes tag. The only difference is that it will be rendered as a select HTML element, not as multiple check boxes.

You can see example of the select tag in the Listing 17-76.

Listing 17-76. *The select Form Tag*

```
<form:form action="edit.html" commandName="customer">
    <table>
        <!-- Other properties, omitted for clarity-- >
        <tr>
            <td>Category:</td>
            <td><form:select path="customer.categories" ➥
                             items="availableCategories"multiple="true"/></td>
            <td> </td>
        </tr>
    </table>
</form:form>
```

We will need to pass the available categories as a model attribute using the controller, in the same way as in Listing 17-69.

The value of the `items` attribute can be the `Map` as well. In this case, the `Map` keys will be the values of rendered check boxes, and the `Map` value for the specific key will be the label for that check box.

Table 17-21 shows main attributes that can be used with the `select` tag.

Table 17-21. *select Tag Attributes*

Attribute	Description
`path`	Identifies the object and its property that will be used to create the status and/or errors
`items`	The `Collection`, `Map`, or array that hold values for the `select` field
`itemLabel`	Name of the property to be displayed as visible text for the option
`itemValue`	Name of the property to be used as the value for the options
`htmlEscape`	Controls the HTML escaping of the rendered values

Standard CSS style attributes, mentioned earlier, are also available for this tag.

The option Tag

The `option` tag renders an HTML `option`. When using this tag, we don't have to supply an `items` attribute to the `select` tag, but we do need to add all options inside the `select` tag.

Listing 17-77. *The option Form Tag*

```
<form:form action="edit.html" commandName="customer">
    <table>
        <!-- Other properties, omitted for clarity-- >
        <tr>
            <td>Category:</td>
            <td><form:select path="customer.categories" multiple="true">
                        <form:option value="Books"/>
                        <form:option value="CDs"/>
                        <form:option value="MP3"/>
                </form:select></td>
            <td> </td>
        </tr>
    </table>
</form:form>
```

Table 17-22 shows the attribute that can be used with the `option` tag.

Table 17-22. *option Tag Attribute*

Attribute	Description
`htmlEscape`	Controls the HTML escaping of the rendered values

Standard CSS style attributes, mentioned earlier, are also available for this tag.

The options Tag

The `options` tag renders a list of HTML `option` tags and is typically used with the singular `option` tag when we want to add an option for a `select` field that is not one of the values provided in the `items` attribute for that `select` field. See Listing 17-78 for an example.

Listing 17-78. *The options Form Tag*

```
<form:form action="edit.html" commandName="customer">
    <table>
        <!-- Other properties, omitted for clarity-- >
        <tr>
            <td>Category:</td>
            <td><form:select path="customer.categories">
                        <form:option value="-" label="--Please Select--"/>
                        <form:options items="${availableCategories}" />
                </form:select></td>
            <td> </td>
        </tr>
    </table>
</form:form>
```

Table 17-23 shows the main attributes that can be used with the `options` tag.

Table 17-23. *options Tag Attributes*

Attribute	Description
`items`	The `Collection`, `Map`, or array that hold values for the `select` field
`itemLabel`	Name of the property to be displayed as visible text for the option
`itemValue`	Name of the property to be used as value for the options
`htmlEscape`	Controls the HTML escaping of the rendered values

Standard CSS style attributes, mentioned earlier, are also available for this tag.

The textarea tag

The `textarea` tag simply renders a `textarea` HTML field. See Listing 17-79 for a simple example.

Listing 17-79. *The textarea Form Tag*

```
<tr>
    <td>Comments:</td>
    <td><form:textarea path="comments" rows="3" cols="20" /></td>
```

```
    <td><form:errors path="comments " /></td>
</tr>
```

This tag can be further controlled using the attributes listed in Table 17-24.

Table 17-24. *textarea Tag Attributes*

Attribute	Description
`path`	Identifies the object and its property that will be used to create the status and/or errors
`htmlEscape`	Controls the HTML escaping of the rendered values

Standard CSS style attributes, mentioned earlier, are also available for this tag.

The hidden tag

The `hidden` tag is used for rendering hidden fields in HTML forms; see Listing 17-80.

Listing 17-80. *The hidden Form Tag*

```
<form:hidden path="id" />
```

The `hidden` tag can be further controlled using the attributes listed in Table 17-25.

Table 17-25. *hidden Tag Attributes*

Attribute	Description
`path`	Identifies the object and its property that will be used to create the status and/or errors
`htmlEscape`	Controls the HTML escaping of the rendered values

Standard CSS style attributes, mentioned earlier, are also available for this tag.

The errors tag

The `errors` tag provides easy and convenient access to the errors created either in your controller or by any validators associated with the controller.

Listing 17-81 shows an example using `EditProductController`, which edits product properties using `ProductValidator`.

Listing 17-81. *ProductValidator Implementation*

```
public class ProductValidator implements Validator {

    public boolean supports(Class product) {
        return Product.class.isAssignableFrom(product);
    }

    public void validate(Object obj, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "name", ➥
                                        "required", "Field is required.");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "expirationDate", ➥
                                        "required", "Field is required.");
```

```
    }
}
```

In Listing 17-82, we create the form that will be validated with our `ProductValidator`.

Listing 17-82. *The Form with the errors Tag*

```
<form:form>
    <table>
        <tr>
            <td>Name:</td>
            <td><form:input path="name" /></td>
            <td><form:errors path="name" /></td>
        </tr>
        <tr>
            <td>Expiration Date:</td>
            <td><form:input path="expirationDate " /></td>
            <td><form:errors path="expirationDate "  /></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form:form>
```

Now, let's see what happens if we submit the form with errors in it. If we submit a form with empty values in the name and `expirationDate` fields, the validator will pick up those errors, and the `form:errors` tag will display them in the HTML. In Listing 17-83, you can see how the generated HTML looks after validation.

Listing 17-83. *Generated HTML for the errors Tag Example from Listing 17-82*

```
<form method="POST">
    <table>
        <tr>
            <td>Name:</td>
            <td><input name="name" type="text" value=""/></td>
            <td><span name="name.errors">Field is required.</span></td>
        </tr>

        <tr>
            <td>Expiration Date:</td>
            <td><input name="expirationDate" type="text" value=""/></td>
            <td><span name="expiration.errors">Field is required.</span></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form>
```

You can use the wildcard character (*) in the `path` attribute of the `errors` tag. If you, for example, want to display all errors associated with the form on the top of the page, you can just add this line to your code: `<form:errors path="*" cssClass="errors" />`.

If you want to display all errors for one field, use standard wildcard character syntax:

```
<form:errors path="name*" cssClass=" errors " />
```

Table 17-26 shows attributes that can be used with this tag.

Table 17-26. *errors Tag Attributes*

Attribute	Description
`path`	Identifies the object and its property that will be used to create the status and/or errors.
`delimiter`	The HTML tag or any `String` to be used as delimiter between the errors. The default is the ` ` tag.
`element`	The HTML tag used to enclose every rendered error.
`htmlEscape`	Controls the HTML escaping of the rendered values.

Standard CSS style attributes, mentioned earlier, are also available for this tag.

JSP Best Practices

As we have stated before, JSP pages offer the largest set of features you can use to generate HTML output. Because JSP pages are compiled into Java classes, applications using JSP pages (as opposed to, for example, Velocity) can suffer from low performance, especially when the pages are still being compiled. Another consideration is that if the content of the page is too big, the JSP will not compile into a Java class, as Java methods cannot be more than 64kB long.

From an architectural point of view, there is danger that developers will use Java code in the pages to perform business operations, which clearly violates the MVC architecture and will cause problems in the future development of the application.

If you keep all these potential limitations in mind, you will find that JSP pages are an excellent technology to use, especially when combined with tag libraries. However, there are other view technologies that may not be as feature-rich as JSP but offer other benefits—the first in line is Velocity.

Using Velocity

Velocity (`http://velocity.apache.org`) is another view templating engine that can be used in Spring. Unlike JSP pages, Velocity templates are not compiled into Java classes. They are interpreted by the Velocity engine every time they are used—even though the pages are interpreted, the time needed to generate the output is very short. In fact, in most cases, Velocity templates will outperform JSP pages. However, using Velocity has some drawbacks: JSP custom tags cannot be used in Velocity, and developers have to familiarize themselves with the syntax of the Velocity Template Language.

Velocity is a stand-alone project of the Apache Software Foundation that is used in many applications that produce text output. You can download the latest version of the Velocity libraries from `http://velocity.apache.org`.

Integrating Velocity and Spring

The Velocity engine requires initialization before it can be used. This initialization is performed by the `VelocityConfigurer` class. We need this bean to set up resource paths to the Velocity template files.

Once an instance of `VelocityConfigurer` class is created in the application context, we can use Velocity templates just like any other view in our application. Listing 17-84 shows how to declare a `velocityConfigurer` bean.

Listing 17-84. *velocityConfigurer Bean Declaration*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
...>
    <bean id="velocityConfigurer"
        class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
        <property name="resourceLoaderPath">
            <value>WEB-INF/views/</value></property>
    </bean>
</beans>
```

Next, we need to add a view definition in the `views.properties` file, as shown in Listing 17-85.

Listing 17-85. *A Velocity View Definition*

```
#product/index
product-index.class=org.springframework.web.servlet.view.velocity.VelocityView
product-index.url=product/index.vm
```

The `product-index` view is going to be created as an instance of `VelocityView`, and the template file will be loaded from `WEB-INF/views/product/index.vm`. The first part of the path to the template file is defined in the `velocityConfigurer` bean, and the second part is declared in the `views.properties` file. Next, we create a simple Velocity template, as shown in Listing 17-86.

Listing 17-86. *product/.vm Velocity Template*

```
<html>
<head>
</head>
<body>

All products:<br>
#foreach($product in $products)
    ${product.name}<br>
#end

</body>
</html>
```

Finally, we are going to modify the `ProductController` used in this chapter to make sure it returns an instance of the `ModelAndView ("product-index", . . .)`.

Listing 17-87. *Modified ProductController That Returns the Velocity View*

```
public class ProductController extends MultiActionController {
    private List<Product> products;

    public ModelAndView index(HttpServletRequest request,
        HttpServletResponse response) {
        return new ModelAndView("product-index", "products", products);
    }

}
```

When the application is rebuilt and deployed, we should see a list of products at `/ch17/product/index.html`, as shown in Figure 17-12.

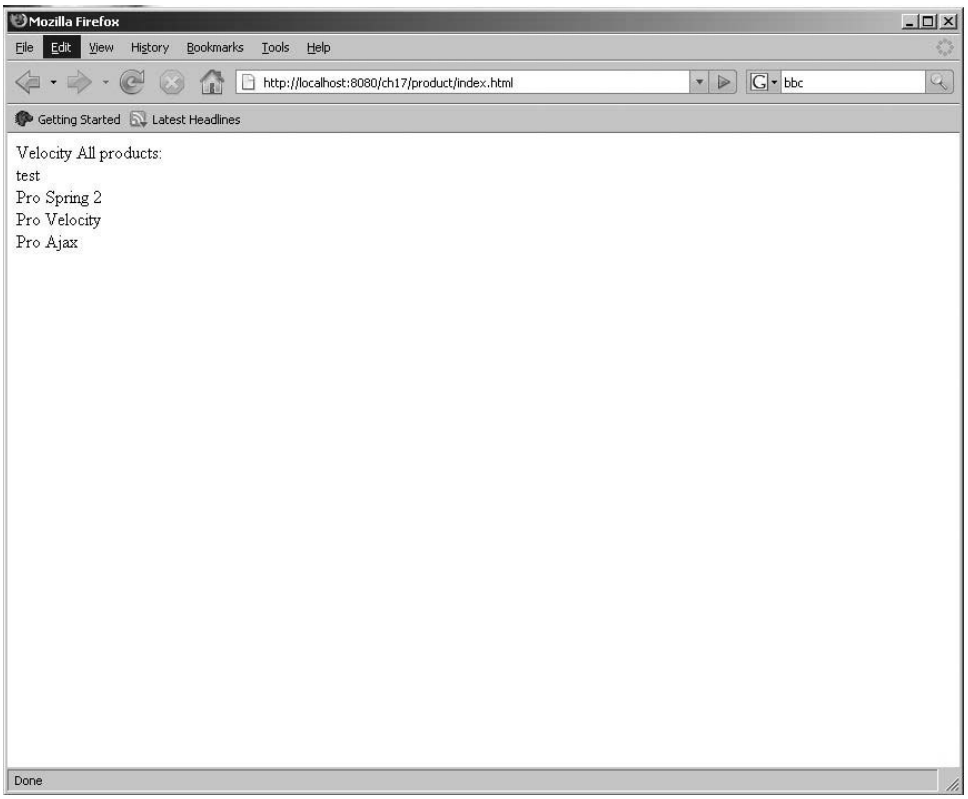

Figure 17-12. *The Velocity example output*

Advanced Velocity Concepts

So far, you have seen simple examples of what Velocity can do used on its own. Using Velocity together with Spring allows you to go much further: you can customize the properties of the Velocity Engine by setting its properties through the `velocityConfigurer` bean and provide macros with functionality similar to the Spring JSTL tags introduced in the "Using JSP Pages" section earlier in this chapter.

There are two ways to set properties of the Velocity Engine: provide a standard properties file or set the properties directly in the `velocityConfigurer` bean definition, as shown in Listing 17-88.

Listing 17-88. *Setting Velocity Engine Properties*

```
 <?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
...>
    <bean id="velocityConfigurer"
        class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
        <property name="resourceLoaderPath" value="WEB-INF/views/"/>
        <property name="velocityProperties">
```

```
            <value>
                file.resource.loader.cache=false
            </value>
        </property>
        <property name="configLocation"
value="/WEB-INF/classes/velocity.properties"/>
    </bean>
</beans>
```

Naturally, you have to decide whether you want to set the Velocity Engine properties in a properties file or keep them in the bean definition.

Perhaps the most important Velocity support comes from the Velocity macros Spring exposes in the Velocity templates. These macros work just like the Spring JSTL tags; the most important macro is the `#springBind` macro. It gives you access to the Spring Validator framework from your Velocity templates. To use the Spring macros in a particular view, the `exposeSpringMacroHelpers` property must be set to `true` (the value of this property is `true` by default).

To demonstrate the usage of the Velocity macros, we are going to create a Velocity template that will allow the users to enter product details, with full validation and error control. We will use the `ProductFormController` introduced earlier in this chapter, but with a modified `views.properties` file and a new `edit.vm` template. We'll start with the modified `views.properties` file, which is shown in Listing 17-89.

Listing 17-89. *The Velocity views.properties File*

```
#products
product-index.class=org.springframework.web.servlet.view.velocity.VelocityView
product-index.url=product/index.vm
product-index-r.class=org.springframework.web.servlet.view.RedirectView
product-index-r.url=/ch17/product/index.html
product-edit.class=org.springframework.web.servlet.view.velocity.VelocityView
product-edit.url=product/edit.vm

#other views omitted
```

The code in bold shows the lines we have added to support the `product-edit` view as a Velocity template. There was no need to set the `exposeSpringMacroHelpers` property to `true`, as it is `true` by default. This configuration allows us to create the `edit.vm` template, shown in Listing 17-90.

Listing 17-90. *The edit.vm Template Contents*

```
<form action="edit.html" method="post">
<input type="hidden" name="productId" value="${command.productId}">
<table>
    <tr>
        <td>Name</td>
        <td>#springBind("command.name")
                <input name="name" value="$!status.value">
                <span class="error">$status.errorMessage</span>
        </td>
    </tr>
    <tr>
        <td>Expiration Date</td>
        <td>#springBind("command.expirationDate")
                <input name="expirationDate" value="$!status.value">
                <span class="error">$status.errorMessage</span>
        </td>
```

```
    </tr>
    <tr>
        <td></td>
        <td><input type="submit"></td>
    </tr>
</table>
</form>
```

Notice that we can use the `#springBind` macro in the template. This macro does precisely the same work its JSTL counterpart: it allows us to access the Spring Validator framework from the template. The result is exactly what you would expect: Figure 17-13 shows a standard HTML form with a working validator in the browser.

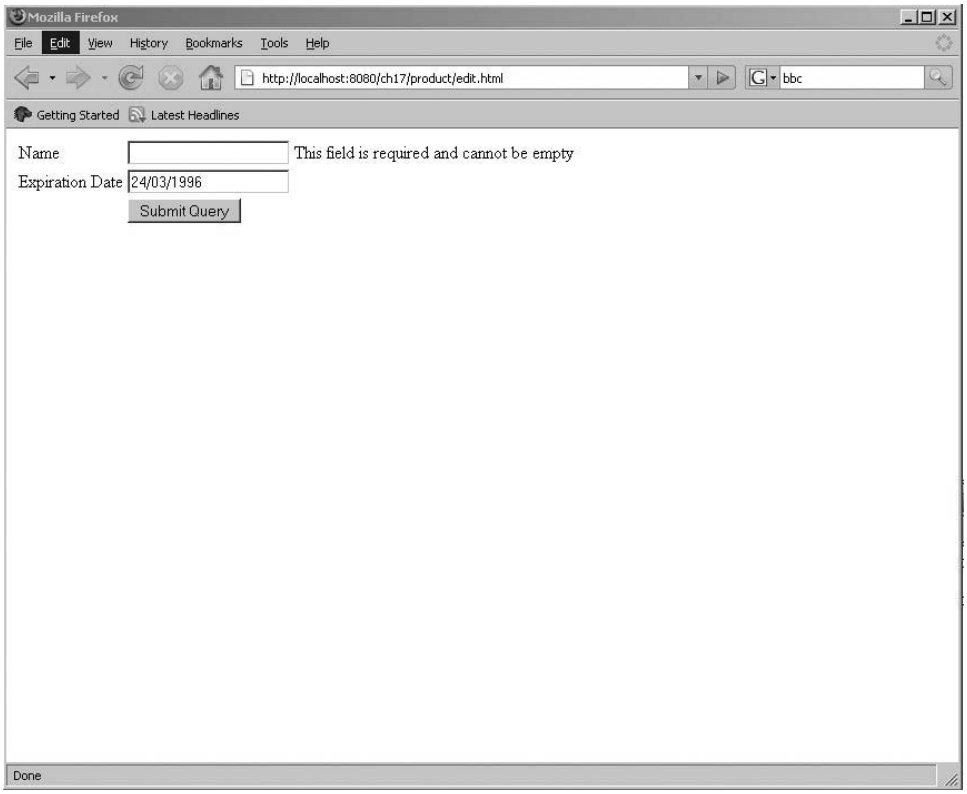

Figure 17-13. *The edit form with a validator message*

There is no difference in the `ProductFormController` code from previous examples; the only difference is the view definition in the `views.properties` file and the presence of the `velocityConfigurer` bean in the application context.

The code in the `edit.vm` file in Listing 17-90 still doesn't take full advantage of the available Spring macros. You could further simplify the code by using the `#springFormInput`, `#springFormTextarea`, `#springFormSingleSelect`, `#springFormMultiSelect`, `#springFormRadioButtons`, `#springCheckboxes`, and `#showErrors` macros.

Velocity provides a good alternative to JSP pages; the lack of features is well justified by its speed.

FreeMarker

FreeMarker is another templating engine that can be used as view technology in Spring. It is a stand-alone project and can be found at `http://freemarker.org/`. Although FreeMarker has some programming capabilities, it is *not* a full-blown programming language. It is designed to be a view-only templating engine, and Java programs are responsible for preparing all the data for the template.

You will have to include `freemarker-2-x.jar` in your application classpath to be able to use FreeMarker as templating language.

Spring Configuration with FreeMarker

As with the Velocity, all you have to do is to add the configurer bean to your `application-servlet.xml` file, as shown in Listing 17-91.

Listing 17-91. *freemarkerConfig Declaration*

```
<bean id="freemarkerConfig" class="org.springframework.web.➥
                servlet.view.freemarker.FreeMarkerConfigurer">
    <property name="templateLoaderPath" value="/WEB-INF/views/"/>
</bean>
```

`TemlateLoaderPath` specifies the path where Spring will look for your FreeMarker templates.

If you want to use FreeMarker as view technology for your web application, you will have to define the `viewResolver` bean in your servlet context file, as shown in Listing 17-92.

Listing 17-92. *Freemarker viewResolver Bean Declaration*

```
<bean id="viewResolver" class="org.springframework.web.➥
              servlet.view.freemarker.FreeMarkerViewResolver">
  <property name="cache" value="true"/>
  <property name="prefix" value=""/>
  <property name="suffix" value=".ftl"/>
</bean>
```

This example configuration has the default prefix and `.ftl` suffix, which indicates a FreeMarker template. Let's create simple FreeMarker template and save it as `/WEB-INF/views/en_GB/product/index.ftl`. Listing 17-93 shows the template code.

Listing 17-93. *FreeMarker Template index.ftl File*

```
<html>
<head>
</head>
<body>

Freemarker All products:<br>
<#list products as product>
    {product.name}<br>
</#list>

</body>
</html>
```

All we have to do now is add a handler method to our `Controller`. Let's add the `indexHandler()` method to previously introduced `ProductController`, as shown in Listing 17-94.

Listing 17-94. *ProductController's Handler Method*

```
   //other methods omitted for clarity
public ModelAndView indexHandler(request, response){
    return new ModelAndView("product/index");
}
```

View with name `product/index` will automatically be mapped to template `index.ftl`, located in directory set in FreeMarker configurer bean (`/WEB-INF/views/en_GB/`). Spring will load the template, populate the variables, and display rendered result in the browser (see Figure 17-14).

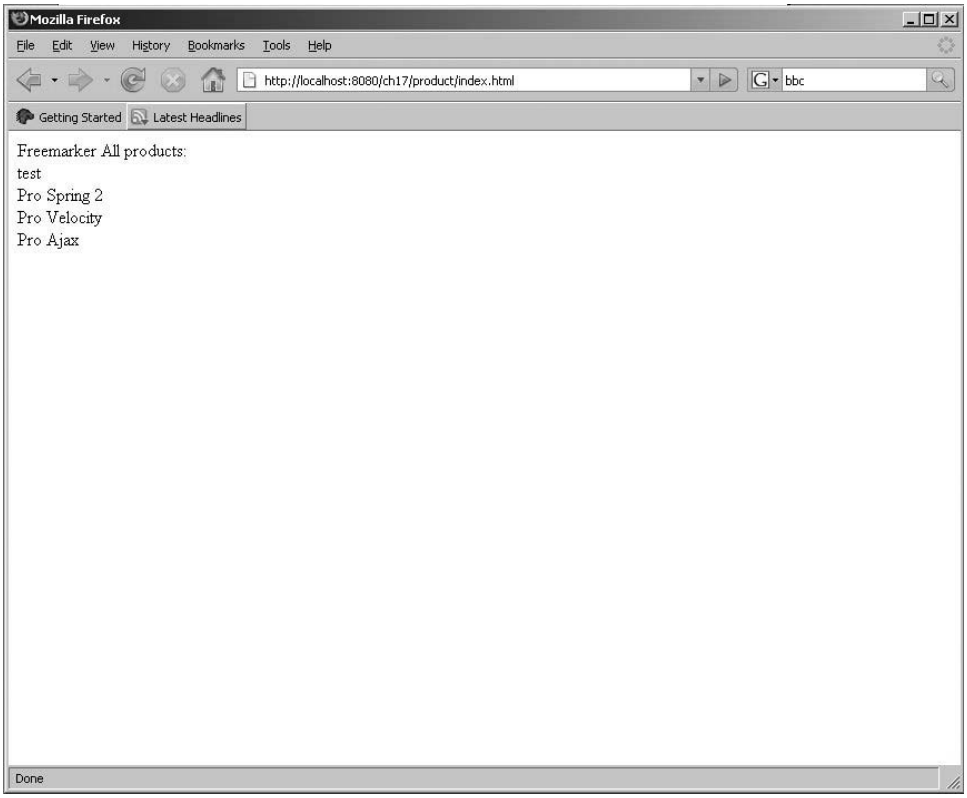

Figure 17-14. *The FreeMarker page in the browser*

As of version 1.1, Spring provides macros for form binding as well as additional convenient macros for easily building form elements in FreeMarker templates. To use Spring FreeMarker macros, you will have to include a macros definition file at the beginning of your template file. The macro definition file is called `spring.ftl` and can be found in the package `org.springframework.web.servlet.view.freemarker` of the `spring.jar` distribution file. Listing 17-95 shows how to import and use the Spring macro library in your FreeMarker templates.

Listing 17-95. *FreeMarker Binding Tags Example*

```
<#import "/spring.ftl" as spring />

<!-- freemarker macros have to be imported into a namespace.  We strongly
recommend sticking to 'spring' -->
```

```
<#import "spring.ftl" as spring />
<html>

<form action="" method="POST">
  Name:
  <@spring.bind "product.name" />
  <input type="text"
    name="${spring.status.expression}"
    value="${spring.status.value?default("")}" /><br>
  <#list spring.status.errorMessages as error> <b>${error}</b> <br> </#list>
  <br>
  Name:
  <@spring.bind "product.expirationDate" />
  <input type="text"
    name="${spring.status.expression}"
    value="${spring.status.value?default("")}" /><br>
  <#list spring.status.errorMessages as error> <b>${error}</b> <br> </#list>
  <br>

  <input type="submit" value="submit"/>
</form>

</html>
```

Figure 17-15 shows how will this template look when rendered in the browser.

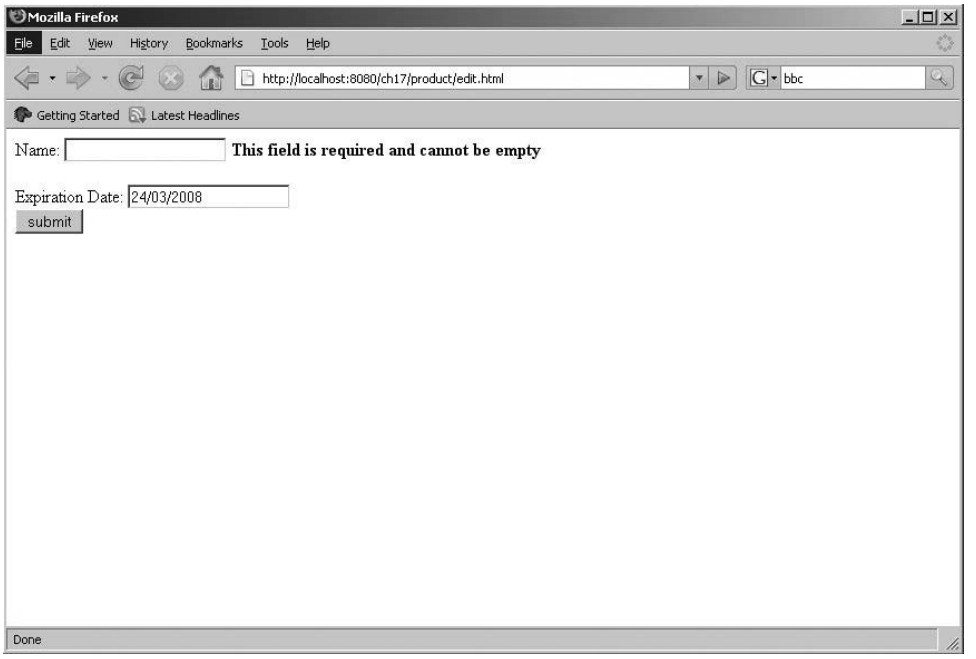

Figure 17-15. *A FreeMarker form*

The `Spring.ftl` FreeMarker macro library has additional convenience macros for HTML input field generation. These macros also include easy binding and validation errors display.

Let's write the example in Listing 17-96 by extending the form in Listing 17-95.

Listing 17-96. *A Spring FreeMarker Macro Library Example*

```
<#import "spring.ftl" as spring />

<!-- freemarker macros have to be imported into a namespace.  We strongly
recommend sticking to 'spring' -->
<#import "spring.ftl" as spring />
<html>

<form action="" method="POST">
  Name:
  <@spring.formInput "product.name" value="${product.name}"/>

  <br>
  Name:
    <@spring.formInput "product.expirationDate" value="${product.expirationDate}"/>

  <input type="submit" value="submit"/>
</form>

</html>
```

The `formInput` macro takes the path parameter (`product.name`) and an additional `attributes` parameter (empty in our example). The macro binds the field to provided path. The `attributes` parameter is usually used for style information or additional attributes required by HTML `input` field (like rows and columns of a text area).

The `showErrors` macro takes a separator parameter (the characters that will be used to separate multiple errors on the field) and accepts, as a second parameter, a class name or style attribute.

In FreeMarker, macros can have default parameters, so we can often call macro with fewer parameters than are defined for it (like in the preceding example, where we haven't passed a second parameter for the `formInput` macro).

Table 17-27 lists all of the macros available in the Spring FreeMarker macros library, with brief descriptions.

Table 17-27. *Spring FreeMarker Available Macros*

Macro	Description
`<@spring.message code/>`	Output a string from a resource bundle based on the code parameter.
`<@spring.messageText code, text/>`	Output a string from a resource bundle based on the code parameter, falling back to the value of the default parameter.
`<@spring.url relativeUrl/>`	Prefix a relative URL with the application's context root.
`<@spring.formInput path, attributes, fieldType/>`	This is the standard input field for gathering user input.
`<@spring.formHiddenInput path, attributes/>`	This hidden input field is for submitting nonuser input.

Continued

Table 17-27. *Continued*

Macro	Description
`<@spring.formPasswordInput path, attributes/>`	This standard input field gathers passwords. Note that no value will ever be populated in fields of this type.
`<@spring.formTextarea path, attributes/>`	This large text field gathers long, free-form text input.
`<@spring.formSingleSelect path, options, attributes/>`	This is a drop-down box of options allowing a single required value to be selected.
`<@spring.formMultiSelect path, options, attributes/>`	This list box of options allows the user to select zero or more values.
`<@spring.formRadioButtons path, options separator, attributes/>`	Create a set of radio buttons allowing a single selection to be made from the available choices.
`<@spring.formCheckboxes path, options, separator, attributes/>`	Create a set of check boxes allowing zero or more values to be selected.
`<@spring.showErrors separator, classOrStyle/>`	Simplify the display of validation errors for the bound field.

Selection field macros accept a `java.util.Map` attribute, containing the options of the select field. Map keys represent option values, and map values are displayed as the labels for those values. Let's see an example of the multiple `select` HTML fields:

```
<@spring.formMultiSelect "customer.categories", |categoriesMap />
```

If the web application expects category IDs as passed values, you can provide `categoriesMap` in the `referenceData()` method of your controller, as shown in Listing 17-97.

Listing 17-97. *referenceData() Method*

```
protected Map referenceData(HttpServletRequest request) throws Exception {
  Map categoriesMap = new LinkedHashMap();
  categoriesMap.put("1", "Books");
  categoriesMap.put("2", "CDs");
  categoriesMap.put("3", "MP3 Playes");

  Map m = new HashMap();
  m.put("categoriesMap", categoriesMap);
  return m;
}
```

FreeMarker offers good speed and some programming capabilities, but its main strength is great macro support that can improve reusability and customization of your web application.

Using XSLT Views

XSLT offers an elegant way to transform XML data into any other plain text format. If you already have an XML document, it may be worth using XSLT views to transform it to HTML output.

Let's create a `ProductsXsltView` that takes the `List` of `Product` objects, builds an XML document, and uses XSLT to transform the list to HTML. Listing 17-98 shows the implementation of the `ProductsXsltView` class.

Listing 17-98. *ProductsXsltView Implementation*

```
public class ProductsXsltView extends AbstractXsltView {

    protected Node createDomNode(Map model, String root,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {

        List products = (List)model.get("products");
        if (products == null) throw new IllegalArgumentException➥
("Products not in model");
        Document document = new Document();
        Element rootElement = new Element(root);
        document.setRootElement(rootElement);

        for (Product product : products) {
            Element pe = new Element("product");
            pe.setAttribute("productId", Integer.toString(product.getProductId()));
            pe.setAttribute("expirationDate", ➥
product.getExpirationDate().toString());
            pe.setText(product.getName());

            rootElement.addContent(pe);
        }

        return new DOMOutputter().output(document);
    }

}
```

Remember that it is not important how you create the `Node` object that the `createDomNode()` method returns; in this case, we have used JDOM (Java Document Object Model), because it is a bit easier to use than the W3C XML API.

The `AbstractXsltView` class allows you to add additional name/value pairs that you can pass as style sheet parameters. For each `<xsl:param name="param-name">param-value</xsl:param>`, you must add an entry to a `Map` returned from the `getParameters()` method.

To test our view, we are going to create an XSLT template that will transform the XML document to a very simple HTML page, as shown in Listing 17-99.

Listing 17-99. *An XSLT Template*

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <html>
            <head>
                <title>Pro Spring 2</title>
            </head>
            <body>
                <h1>Available Products</h1>
                <xsl:for-each select="products/product">
                    <xsl:value-of select="."/>
                    <br />
                </xsl:for-each>
            </body>
        </html>
    </xsl:template>
</xsl:stylesheet>
```

Just like with any other view, we need to declare it in the `views.properties` file.

Listing 17-100. *views.properties Declaration of ProductsXsltView*

```
product-index.class=com.apress.prospring2.ch17.web.views.ProductsXsltView
product-index.root=root
product-index.stylesheetLocation=/WEB-INF/views/product/index.xslt
```

When we deploy the application, we can see the product listing as a regular HTML page.

You should use XSLT views with caution as the processing involved is very complex, and in most cases, an XSLT view is the slowest to render view available. In our example, we have actually built the XML document and used XSLT to transform it to HTML—this is without any doubt the worst way to use XSLT views. However, if you already have XML document and all you need is to transform it to HTML, you can certainly benefit from implementing an XSLT view.

Using PDF Views

Even though the previous view technologies are powerful, they still depend on the browser to interpret the generated HTML code. Though basic HTML is rendered without any problems on all browsers, you cannot guarantee that a page with complex formatting is going to look the same on all browsers. Even more, there are situations where HTML is just not enough, if you need to print the output, for example. If this is the case, the best solution is to use a document format that will be rendered consistently on all clients.

You will see in this section how to use PDF files as the view technology. To use a PDF as a Spring `View`, you will need to include the iText (`http://www.lowagie.com/iText/`) and PDFBox (`http://www.pdfbox.org/`) JAR files; both are included in the Spring distribution.

Unfortunately, there is no PDF template language, so we have to implement the views ourselves. As you saw in this chapter's XSLT view example, implementing a custom view is not a difficult task, and Spring provides a convenience superclass, `AbstractPdfView`, which you can use in your PDF view implementation.

Implementing a PDF View

Generating a PDF from your Spring web application is not too difficult, even though there can be a lot of coding involved. The PDF view we are going to implement will display product details. We are not going to make any change to the `ProductController` whose `view()` method processes requests to view a product. We will need to make a change to the `views.properties` file and implement the `ProductPdfView` class to create the PDF output. Let's begin by looking at Listing 17-101, which shows the implementation of `ProductPdfView` class.

Listing 17-101. *ProductPdfView Implementation*

```
public class ProductPdfView extends AbstractPdfView {

    protected void buildPdfDocument(Map model, Document document,
        PdfWriter writer, HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        Product product = (Product) model.get("product");
        if (product == null) throw new
            IllegalArgumentException("Product not present in the model");
```

```
        Paragraph header = new Paragraph("Product details");
        header.font().setSize(20);
        document.add(header);

        Paragraph content = new Paragraph(product.getName());
        document.add(content);

        Paragraph footer = new Paragraph("Pro Spring Chapter 17");
        footer.setAlignment(Paragraph.ALIGN_BOTTOM);
        document.add(footer);
    }

}
```

This view simply extracts the `Product` domain object from the model and uses its data to add paragraphs to the document instance. Because `ProductPdfView` is a subclass of `AbstractPdfView`, we do not have to worry about setting the appropriate HTTP headers or performing any I/O—the superclass takes care of all that.

Before we can verify that the newly created `ProductPdfView` class works, we need to modify the `views.properties` file and make sure that the `product-view`'s class is set to `ProductPdfView` and that the `ProductController.view()` adds a `Product` instance to the model. Listing 17-102 shows the changes we need to make to `views.properties` file.

Listing 17-102. *views.properties File for the product-view*

```
product-view.class=com.apress.prospring2.ch17.web.views.ProductPdfView
```

You may notice that we do not need to set any additional properties on the `product-view`. Next, we need to make sure that the `ProductController.view()` method returns the correct instance of `ModelAndView`, as shown in Listing 17-103.

Listing 17-103. *ProductController.view Implementation*

```
public class ProductController extends MultiActionController {
    private Product createProduct(int productId, String name, Date expirationDate) {
        Product product = new Product();
        product.setProductId(productId);
        product.setName(name);
        product.setExpirationDate(expirationDate);

        return product;
    }

    public ModelAndView view(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        Product product = createProduct(1, "Pro Spring", new Date());
        return new ModelAndView("product-view", "product", product);
    }
}
```

When the application is rebuilt and deployed, the PDF document in Figure 17-16 should be returned for `/product/view.html`.

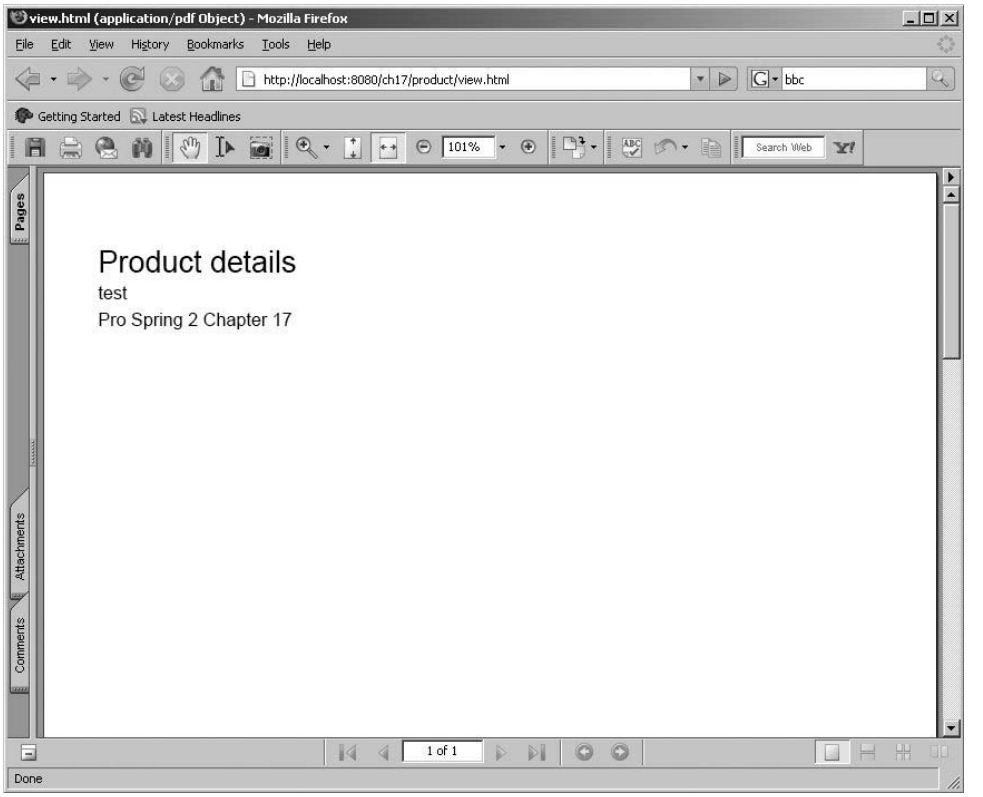

Figure 17-16. *ProductPdfView output*

Using Excel Views

If your application requires Excel output, you can use Spring to create an Excel view rather than directly writing the contents of an Excel file to the output stream. Just like `AbstractPdfView`, Spring provides `AbstractExcelView`, which you can subclass to further simplify development of new Excel views. You will need the Jakarta POI library (`http://jakarta.apache.org/poi/`) to perform the actual Excel I/O; the POI JAR file comes with the Spring distribution.

We are going to show a simple `ProductsExcelView` that renders the list of products into an Excel spreadsheet. We will implement `ProductsExcelView` and modify `views.properties`. The implementation of `ProductsExcelView` is going to extend `AbstractExcelView`, as shown in Listing 17-104.

Listing 17-104. *ProductsExcelView Implementation*

```
public class ProductsExcelView extends AbstractExcelView {

    private static final int COL_PRODUCT_ID = 0;
    private static final int COL_NAME = 1;
    private static final int COL_EXPIRATION_DATE = 2;

    protected void buildExcelDocument(Map model, HSSFWorkbook wb,
        HttpServletRequest request, HttpServletResponse response) throws Exception {
        List<Product> products = (List<Product>)model.get("products");
        HSSFSheet sheet = wb.createSheet("Products");
        int row = 0;
        getCell(sheet, row, COL_PRODUCT_ID).setCellValue("Id");
        getCell(sheet, row, COL_NAME).setCellValue("Name");
        getCell(sheet, row, COL_EXPIRATION_DATE).setCellValue("ExpirationDate");
        row++;
        for (Product product : products) {
            getCell(sheet, row, COL_PRODUCT_ID).setCellValue(
                product.getProductId());
            getCell(sheet, row, COL_NAME).setCellValue(
                product.getName());
            getCell(sheet, row, COL_EXPIRATION_DATE).setCellValue(
                product.getExpirationDate());
            row++;
        }
    }

}
```

The code for the `ProductsExcelView` class is quite simple: we get a `List` of `Product` objects and iterate over the list, adding a row to the Excel workbook in each iteration.

Next, we need to make sure that the `product-index` view declared in `views.properties` is referencing the newly created `ProductsExcelView` class, as shown in Listing 17-105.

Listing 17-105. *views.properties with the ProductsExcelView Definition*

```
product-index.class=com.apress.prospring.ch17.web.views.ProductsExcelView
```

Again, we do not need to set any additional properties for this view. The `ProductController.index()` method does not need any modification, as it already returns `ModelAndView ("product-index", . . .)`. When we make a request to `/product/index.html`, we will get an Excel spreadsheet, as shown in Figure 17-17.

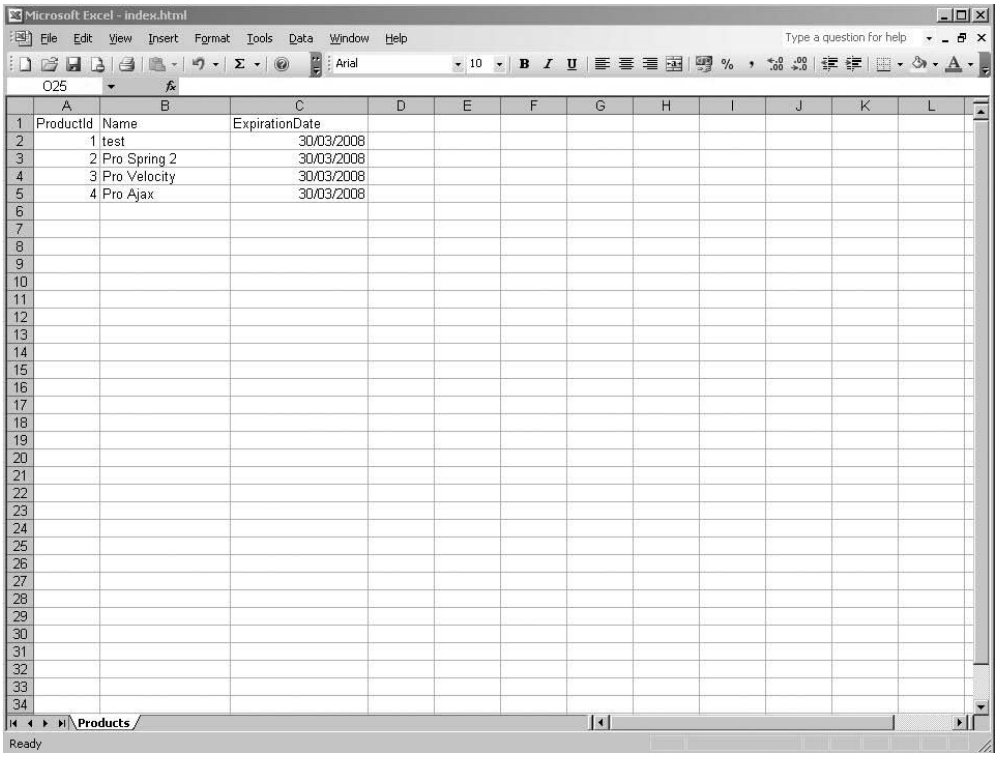

Figure 17-17. *Output of the ProductsExcelView*

As you can see, using `AbstractExcelView` is very similar to using `AbstractPdfView`: in both cases, we have to implement code that will create the content of the document, and we do not need to handle the I/O.

Using Tiles

Unlike all the previous view technologies, Tiles (`http://tiles.apache.org/`) is not actually used to generate output. Instead, it can combine the output from various views into a master view. Each individual tile can consist of a collection of tiles, a JSP page, or output directly written to the `HttpResponse` object in the `handle()` method of a Spring controller.

Tiles was a part of the Jakarta Struts project (`http://struts.apache.org/`). However, due to its growing popularity, it was split off into an independent project with a new version, known as Tiles 2. You will have to include the following dependencies in your project to be able to use Tiles: Tiles version 2.0.4 or higher, Commons BeanUtils, Commons Digester, and Commons Logging.

Integrating Tiles and Spring

Tiles requires some initialization before you can use it. In a Struts application, Tiles is initialized when loaded as a plug-in. In Spring, we need to create a `TilesConfigurer` bean that will load the tile definition XML files and configure the Tiles framework, as shown in Listing 17-106.

Listing 17-106. *TilesConfigurer Definition in the Application Context*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
...>
    <bean id="tilesConfigurer"
        class="org.springframework.web.servlet.view.tiles2.TilesConfigurer">
        <property name="definitions">
            <list>
                <value>/WEB-INF/tiles-layout.xml</value>
            </list>
        </property>
    </bean>
</beans>
```

This bean creates the `TilesDefinitionsFactory` instance using the definition files listed in the `definitions` property of the `TilesConfigurer` bean.

Before we can use Tiles in our application, we must create the `tiles-layout.xml` file; we will start with a very simple one. You can build and deploy your web application to check that the Tiles support is properly configured (if there is a problem in the configuration, the deployment will fail).

Listing 17-107. *Simple Tiles Definition File*

```
 <!DOCTYPE tiles-definitions PUBLIC
       "-//Apache Software Foundation//DTD Tiles Configuration 2.0//EN"
       "http://tiles.apache.org/dtds/tiles-config_2_0.dtd">

<tiles-definitions>
    <definition name=".dummy"/>
</tiles-definitions>
```

When we build and deploy the application, the application server's console will print debug messages from the `TilesConfigurer` class indicating that the Tiles support has been correctly configured; see Listing 17-108.

Listing 17-108. *TilesConfigurer Debug Messages*

```
21:07:58,001 INFO  [TilesConfigurer] TilesConfigurer: initializion started
21:07:58,011 INFO  [TilesConfigurer] TilesConfigurer: adding definitions ➥
[/WEB-INF/tiles-layout.xml]
21:07:58,211 INFO  [TilesConfigurer] TilesConfigurer: initialization completed
```

Now that we have verified that the Tiles support is working, we need to think about the tiles we want to use in our application. A typical web page layout might look like the one shown in Figure 17-18. We are going to call this page a root page; it is a JSP page that uses the Tiles tag library to insert the appropriate tiles according to their definitions in the tiles configuration files.

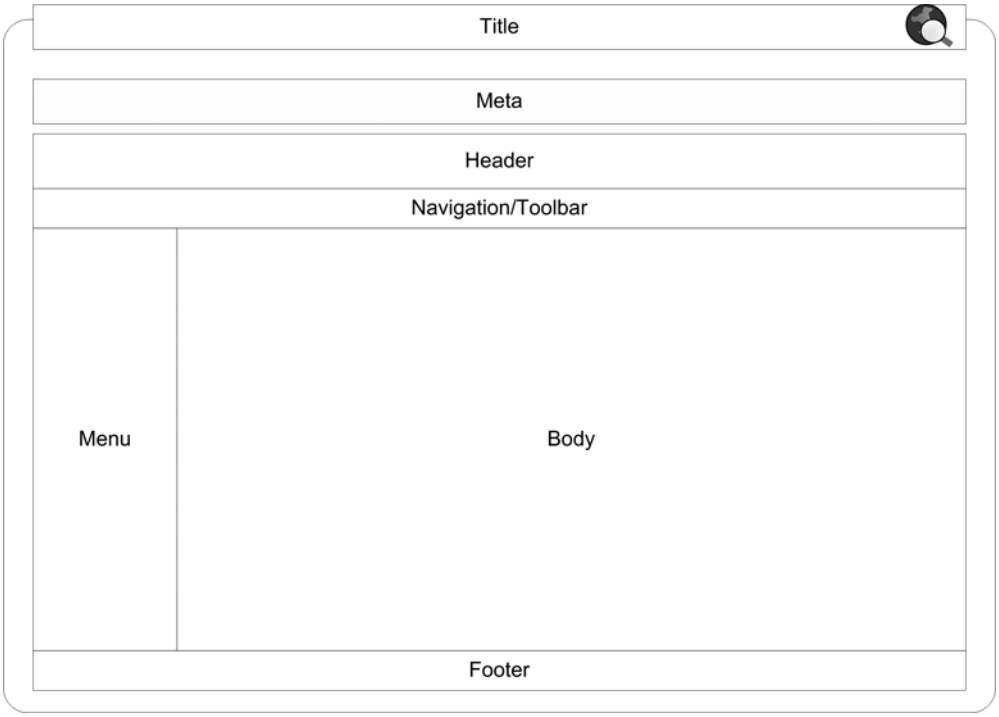

Figure 17-18. *Tiles layout*

Looking at this layout, you might think that we are going to create five tiles: one for each section of the final page layout. But we are going to add sixth tile, which is going to output the content of the `<meta>` tag in the root layout.

Let's begin by creating the `root.jsp` page that will place all the tiles into a HTML table, as shown in Listing 17-109.

Listing 17-109. *The root.jsp Page*

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

<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core_rt" %>

<html>
<head>
    <c:set var="css"><spring:theme code="css"/></c:set>
       <c:if test="${not empty css}">
            <link rel="stylesheet" href="<c:url value="${css}"/>" type="text/css" />
        </c:if>
    <tiles:insertAttribute name="meta"/>
    <title><tiles:getAsString name="title"/></title>
</head>

<table cellspacing="0" cellpadding="0" width="700px" align="center"
    bgcolor="#ffffff">
    <tr>
```

```
        <td colspan="2"><tiles:insertAttribute name="header"/></td>
    </tr>
    <tr>
        <td colspan="2"><tiles:insertAttribute name="toolbar"/></td>
    </tr>
    <tr height="400px">
        <td width="150px" valign="top"><tiles:insertAttribute name="menu"/></td>
        <td width="550px" valign="top"><tiles:insertAttribute name="body"/></td>
    </tr>
    <tr>
        <td colspan="2"><tiles:insertAttribute name="footer"/></td>
    </tr>
</table>

</body>
</html>
```

The root layout page is straightforward: we define the layout of the page, and we use Tiles tags to specify where Tiles should insert the appropriate pages. However, the Tiles framework still doesn't know what content to insert for all `<tiles:insertAttribute>` and `<tiles:getAsString>` tags. We need to create the Tiles definition file shown in Listing 17-110.

Listing 17-110. *Tiles Definition File*

```
 <!DOCTYPE tiles-definitions PUBLIC
        "-//Apache Software Foundation//DTD Tiles Configuration 2.0//EN"
        "http://tiles.apache.org/dtds/tiles-config_2_0.dtd">
<tiles-definitions>
    <!-- Abstract root definition -->
    <definition name=".root" template="/WEB-INF/views/en_GB/tiles/root.jsp">
        <put-attribute name="title" value="CHANGE-ME"/>
        <put-attribute name="meta" value="/WEB-INF/views/en_GB/tiles/meta.jsp"/>
        <put-attribute name="header" value="/WEB-INF/views/en_GB/tiles/header.jsp"/>
        <put-attribute name="menu" value="/WEB-INF/views/en_GB/tiles/menu.jsp"/>
        <put-attribute name="toolbar" value="/WEB-INF/views/➥
                en_GB/tiles/toolbar.jsp"/>
        <put-attribute name="footer" value="/WEB-INF/views/en_GB/tiles/footer.jsp"/>
    </definition>

    <!-- Index -->

    <definition name=".index" extends=".root">
                <put-attribute name="title" value="Main Page"/>
                <put-attribute name="body" value="/WEB-INF/views/en_GB/index.jsp"/>
            </definition>

    <definition name=".status" extends=".root">
        <put-attribute name="title" value="Status"/>
        <put-attribute name="body" value="/tile/status.tile"/>
    </definition>
</tiles-definitions>
```

This definition file introduces a number of Tiles concepts, so let's go through the features used line by line. The first definition element's attribute name is set to `.root`, and the element also includes the `path` attribute, which instructs Tiles to use the JSP page specified in the path attribute and instructs the values specified in the `put` elements to display their content.

However, the `.root` definition is missing the `body` attribute we are using in the `root.jsp` page. The definition we have created is an abstract definition, as it does not define all the attributes used. We must extend this abstract definition to ensure that all the necessary attributes are defined, as shown in the second definition element. Its `name` attribute is set to `.index`, and its `extends` attribute specifies that it should inherit all values in the `put` elements from the definition element whose name is `.root`. The `name` attribute overrides the `title` value and adds the `body` value—just as you would expect in Java code.

Before we can move ahead and configure the Spring views to use Tiles, we need to stress that the individual pages that are used as tiles must not include the standard HTML headers, but if you are using JSP pages, they must include all tag library references. The prohibition against using document HTML tags is obvious, as the document tags are already included in the root page. The taglibs must be included in the JSP pages, because Tiles will request each tile individually; the tiles are not actually aware of the fact that the output they are rendering is being collected by another layer and formatted using the root layout.

Now that you know the requirements for individual tiles, we must configure the Spring views to use the Tiles framework. To do this, we are going to modify the `views.properties` file as shown in Listing 17-111.

Listing 17-111. *views.properties Definition*

```
#index
index.class=org.springframework.web.servlet.view.tiles2.TilesView
index.url=.index
```

This file defines that the view name index is going to be created as an instance of `TilesView` and its URL is going to be `.index`.

■ **Note** You may be wondering why we are using dots (`.`) in the Tiles definition names and no dots in the view names. This is simply a practice we find useful when managing applications with a large number of views and Tiles definitions. By looking at the names, you can immediately identify Tiles and Spring views.

The last step we need to take before we can test our application is to make sure that the `IndexController`'s `handleRequestInternal()` method is returning the correct view; see Listing 17-112.

Listing 17-112. *The IndexController Class*

```
public class IndexController extends AbstractController {

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

        return new ModelAndView("index", model);
    }

}
```

When we now deploy the application and make a request to the `/index.html` page, the `TilesConfigurer` bean will parse the Tiles configuration, load the `.root` and `.index` definitions, call each JSP page and render its output, and finally take the JSP output and output it into the appropriate places in the `root.jsp` page, whose output will be returned to the client as shown in Figure 17-19.

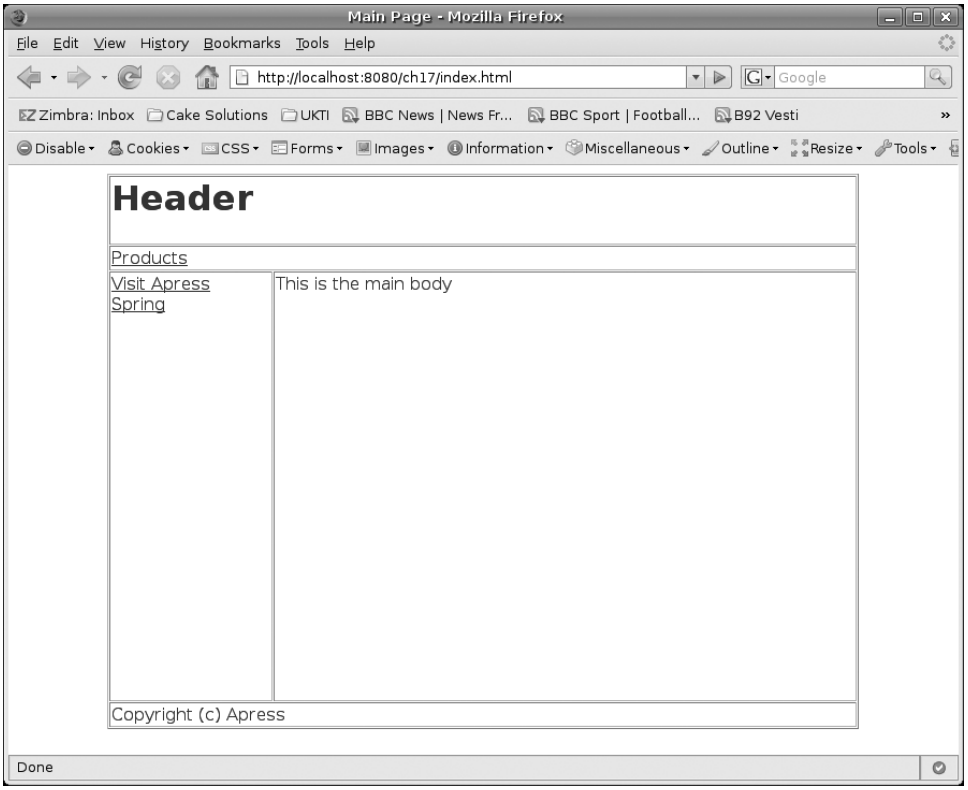

Figure 17-19. *The .root Tiles view*

Advanced Tiles Concepts

In the previous section, we showed you how to use Tiles in your Spring application in a way that is not too different from the `@include` JSP directive. The true power of Tiles comes from the fact that Tiles can take any output and paste it into the appropriate place. A tile can consist of other tiles, JSP pages, or even simple output from a `Controller`. We are going to start with a tile whose content is the output written to the response stream by a simple controller.

We are going to create a tile that prints out the memory usage information. It will print this information directly to the `HttpServletResponse`'s `Writer` object. We are going to create another servlet mapping in `web.xml` to map all `*.tile` requests to the `ch17` servlet. The only reason for this is to keep the request namespace clean of any ambiguous request URLs. The modified `web.xml` file is shown in Listing 17-113.

Listing 17-113. *web.xml Descriptor*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee ➥
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
         version="2.4">    <!-- omitted for clarity -->
    <servlet-mapping>
```

```
        <servlet-name>ch17</servlet-name>
        <url-pattern>*.tile</url-pattern>
    </servlet-mapping>
</web-app>
```

Next, we are going to create a `TileController`, which is going to be a subclass of `MultiActionController`. We are going to implement only one method in the `TileController`, the `handleStatus()`; it will print out the memory information. Listing 17-114 shows that the implementation is quite trivial.

Listing 17-114. *TileController.handleStatus Implementation*

```
public class TileController extends MultiActionController {

    private void writeMemoryPoolMXBean(MemoryPoolMXBean bean,
        PrintWriter writer) {
        writer.append("<pre><tt>");

        writer.append("Name: "); writer.append(bean.getName());
        writer.append("\n");
        writer.append("Type: "); writer.append(bean.getType().name ());
        writer.append("\n");
        writer.append("Usage: "); writer.append(bean.getUsage().toString());
        writer.append("\n");

        writer.append("</pre></tt>");
    }

    public ModelAndView handleStatus(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        List<MemoryPoolMXBean> beans =
                ManagementFactory.getMemoryPoolMXBeans();
        PrintWriter writer = response.getWriter();
        for (MemoryPoolMXBean bean : beans) {
            writeMemoryPoolMXBean(bean, writer);
        }
        return null;
    }

}
```

Notice that the `handleStatus()` method returns `null`, which means that Spring will not attempt to perform any view processing.

Next, we are going to declare the `tileController` bean in the application context file together with a `tileMethodNameResolver` bean and an entry in the `publicUrlMapping` bean, as shown in Listing 17-115.

Listing 17-115. *The tileController and tileMethodNameResolver Beans*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
...>
    <bean id="publicUrlMapping"
        class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <value>
                /index.html=indexController
                /product/index.html=productController
                /product/view.html=productController
                /product/edit.html=productFormController
                /product/image.html=productImageFormController

               /tile/*.tile=tileController
            </value>
        </property>
    </bean>

    <!-- Tile -->
    <bean id="tileController"
        class="com.apress.prospring2.ch17.web.tiles.TileController">
        <property name="methodNameResolver"
             ref="tileMethodNameResolver"/>
    </bean>
    <bean id="tileMethodNameResolver"
        class="org.springframework.web.servlet.mvc.multiaction.➥
PropertiesMethodNameResolver">
        <property name="mappings">
            <value>
                /tile/status.tile=handleStatus
            </value>
        </property>
    </bean>
</beans>
```

We can test that our `tileController` works by making a request to `/ch17/tile/status.tile`. This should print the JVM memory status information. Finally, we are going to create `StatusController` as a subclass of `AbstractController`, add it to the application context file, and add an entry to the `publicUrlMapping` bean to map `/status.html` URL to the `StatusController`, as shown in Listing 17-116.

Listing 17-116. *The statusController Bean Definition and the New Entry in the publicUrlMapping Bean*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
...>
    <bean id="publicUrlMapping"
        class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
```

```
            <value>
                /index.html=indexController
               /status.html=statusController
                /product/index.html=productController
                /product/view.html=productController
                /product/edit.html=productFormController
                /product/image.html=productImageFormController

                /tile/*.tile=tileController</prop>
            </value>
        </property>
    </bean>
    <bean id="statusController"
        class="com.apress.prospring2.ch17.web.StatusController"/>
</beans>
```

The code in the `StatusController.handleRequestInternal` method simply returns an instance of `ModelAndView ("status")`. This means that we need to add an entry to the `tiles-layout.xml` file and an entry to `views.properties`, as shown in Listing 17-117.

Listing 17-117. *Additions to tiles-layout.xml and views.properties*

```
//tiles-layout.xml:
 <!DOCTYPE tiles-definitions PUBLIC
       "-//Apache Software Foundation//DTD Tiles Configuration 2.0//EN"
       "http://tiles.apache.org/dtds/tiles-config_2_0.dtd">

<tiles-definitions>
    <!-- other definitions omitted -->

   <definition name=".status" extends=".root">
        <put-attribute name="title" value="Status"/>
        <put-attribute name="body" value="/tile/status.tile"/>
    </definition>
</tiles-definitions>

//views.properties:
status.class=org.springframework.web.servlet.view.tiles2.TilesView

status.url=.status
```

When we make a request to `/ch17/status.html`, Spring will instantiate the `status` view defined in `views.properties`. This view points to the `.status` tile definition, which specifies that the value for the body element should be taken from the output generated by `/ch17/tile/status.tile`. The final result should look something like the page shown in Figure 17-20.

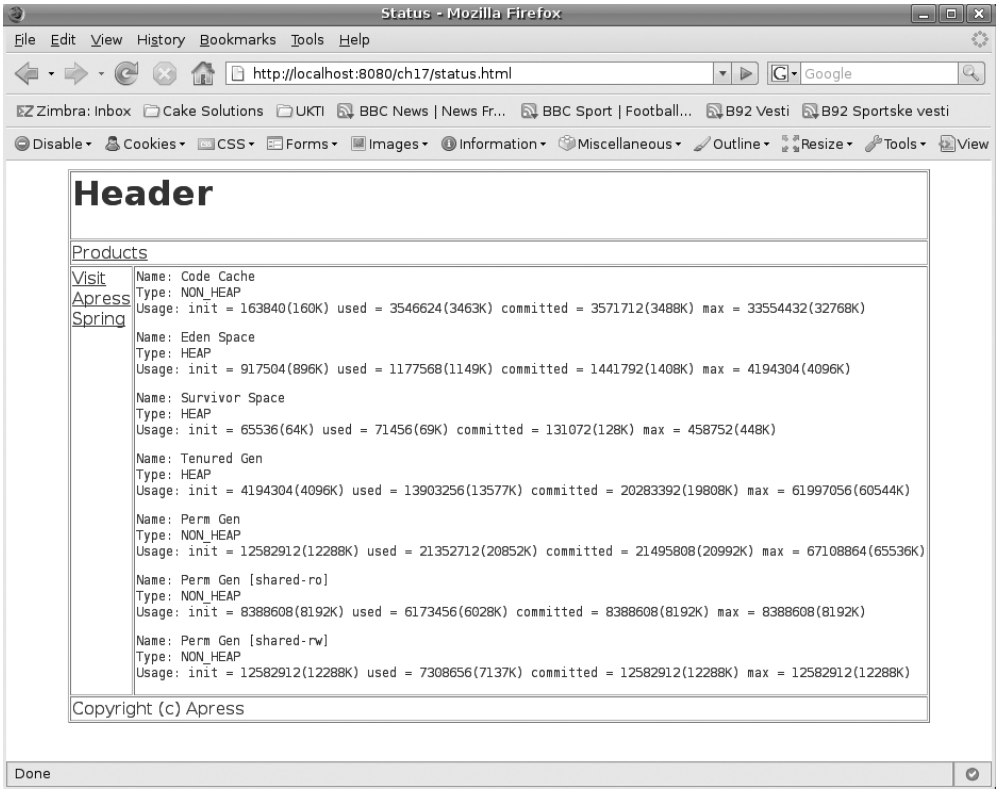

Figure 17-20. *Rendered /ch17/status.html page*

You would rarely use a controller that prints output directly to the output stream as our previous example does; in most cases, a controller returns a `ModelAndView` object that identifies a view and data to be rendered by that view. You can use this approach in a Tiles application as well. To demonstrate, we are going to add `handleMenu()` method to the `TileController`; this method reads the links from the configuration file and renders them using a JSP page.

First, we are going to modify the `TileController` as shown in Listing 17-118.

Listing 17-118. *Modified TileController*

```
public class TileController extends MultiActionController {
    private Map menu;
    public ModelAndView handleMenu(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        return new ModelAndView("tile-menu", "menu", menu);
    }

    public void setMenu(Map menu) {
        this.menu = menu;
    }
}
```

The handleMenu() method takes the menu property set in the application context file and forwards to the tile-menu view, which is defined as JstlView in views.properties, as shown in Listing 17-119.

Listing 17-119. *JstlView Definition in views.properties File*

```
#index
index.class=org.springframework.web.servlet.view.tiles2.TilesView

index.url=.index

#status
status.class=org.springframework.web.servlet.view.tiles2.TilesView

status.url=.status

#menu tile
tile-menu.class=org.springframework.web.servlet.view.JstlView

tile-menu.url=/WEB-INF/views/tiles/menu2.jsp
```

Listing 17-120 shows that the JSP page that displays the menu is quite trivial; it uses the core JSTL tags to iterate over all items in a map.

Listing 17-120. *The tile/menu2.jsp Page*

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core_rt" %>

<c:forEach items="${menu}" var="item">
    <a href="<c:out value="${item.value}"/>"><c:out value="${item.key}"/></a><br>
</c:forEach>
```

Finally, we will modify the tiles-layout.xml and ch17-servlet.xml files to use the newly created menu; see Listing 17-121.

Listing 17-121. *Updated tiles-layout.xml and ch17-servlet.xml Files*

tiles-layout.xml:

```
<tiles-definitions>
    <!-- Abstract root definition -->
     <definition name=".root" template="/WEB-INF/views/en_GB/tiles/root.jsp">
        <put-attribute name="title" value="CHANGE-ME"/>
        <put-attribute name="meta" value="/WEB-INF/views/en_GB/tiles/meta.jsp"/>
        <put-attribute name="header" value="/WEB-INF/views/en_GB/tiles/header.jsp"/>
        <put-attribute name="menu" value="/tile/menu.tile"/>
        <put-attribute name="toolbar" ➥
                 value="/WEB-INF/views/en_GB/tiles/toolbar.jsp"/>
        <put-attribute name="footer" value="/WEB-INF/views/en_GB/tiles/footer.jsp"/>
    </definition>
```

ch17-servlet.xml:

```
.......
    <bean id="tileController" class="com.apress.prospring2➥
               .ch17.web.tiles.TileController">
        <property name="methodNameResolver" ref ="tileMethodNameResolver"/>
        <property name="menu">
            <map>
                 <entry key="Apress" value="http://www.apress.com"/>
```

```
                <entry key="Spring" value="http://www.springframework.org"/>
                <entry key="Cake Solutions" ➥
                      value="http://www.cakesolutions.net"/>
            </map>
        </property>
    </bean>
    <bean id="tileMethodNameResolver"
        class="org.springframework.web.servlet.mvc.multiaction.➥
PropertiesMethodNameResolver">
        <property name="mappings">
            <value>
                /tile/status.tile=handleStatus
               /tile/menu.tile=handleMenu
            <value/>
        </property>
    </bean>
</beans>
```

Because we have modified the `.root` tile definition, our new menu is going to be used in all definitions that extend the `.root` definition, and the newly displayed menu is shown in Figure 17-21.

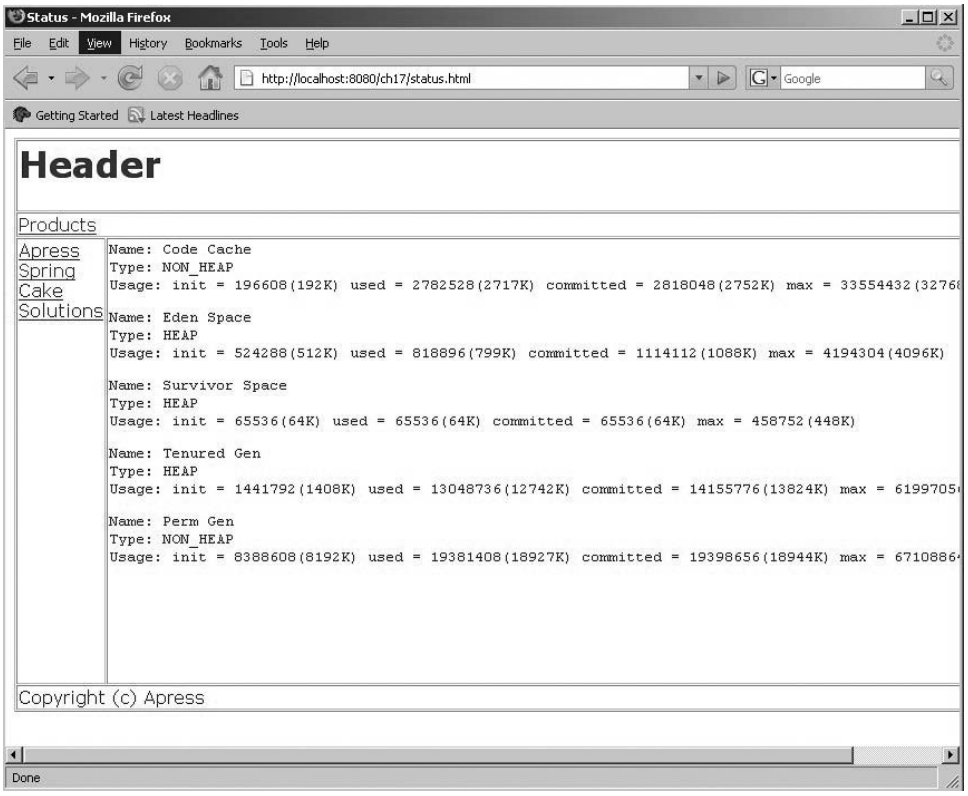

Figure 17-21. *Our new menu tile*

Tiles Best Practices

Tiles is a very powerful framework that can greatly simplify the development of web applications. With its power comes the complexity of configuration files and different request paths. Therefore, using a logical naming convention and enforcing this naming convention throughout the whole project is important. We feel that the best way to organize the URLs to be rendered as tiles is to use `*.html` for the final pages returned to the client, and `*.tile` for requests whose output is to be rendered as a tile. The view naming conventions follow the directory structure of the source files, so a Spring `JstlView` definition for a JSP page located in `tile/menu.jsp` will be named `tile-menu`. Finally, a tile definition that will be used to render the output of the `/product/index.html` page is best named `.product.index`.

■ **Note** You should take a look at the SiteMesh layout framework as well, which is based on the Decorator pattern. SiteMesh uses servlet filters instead of JSP includes, which means it intercepts request at lower level. So there is no other special or Spring-specific configuration; you just configure the SiteMesh filter in your `web.xml` file. This makes SiteMesh extremely easy to configure and maintain. It uses template files called detectors, which describe how the page will be rendered. Decorators can be written using different technologies, like JSP, FreeMarker, or Velocity.

SiteMesh is written in Java but can be used with all web technologies, whether they're Java based or not. The pages don't even know they are being decorated, unlike in Tiles, where each individual page has to be associated with a layout. However, SiteMesh doesn't allow inheritance of page definitions or overriding attributes. It also lacks debugging support, so you could have problem identifying the errors, especially if you use nested decorators. SiteMesh is available at `http://www.opensymphony.com/sitemesh/`.

JasperReports

JasperReports (`http://jasperreports.sourceforge.net`) is an open source reporting engine written entirely in Java. Its main goal is to deliver print-ready documents in any of the supported formats: CSV, Microsoft Excel, HTML, or PDF.

JasperReports uses one XML-based format for report design files. You can create the design files by yourself in any text editor, and you can change any report details at a low level. Thankfully, however, you do not need to hand-code all of your report files, because a wide range of graphical tools is available for working with JasperReports design files. You can find the complete list of these tools on the JasperReports web site.

Again, the design files are actually XML files, with the extension `.jrmxl`. Before you can use a report design file, it has to be compiled into a report file with the `.jasper` extension. JasperReports reports are distributed with Ant task for this compilation. However, when using JasperReports with Spring, you can let Spring compile your reports on the fly. You can supply either a `.jrxml` or `.jasper` file to Spring when generating your reports: if you supply a `.jrxml` design file, Spring compiles the report to `.jasper` format and caches the compiled file to serve future requests. You can supply already compiled `.jasper` file as well, which will Spring use directly, without compiling.

Let's design a simple report that will display the list of `Product` objects. Listing 17-122 shows our `Product` class.

Listing 17-122. *Product Java Object*

```
public class Product {
    private String name;
    private Date expirationDate;
```

```
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Date getExpirationDate() {
        return expirationDate;
    }

    public void setExpirationDate(Date expirationDate) {
        this.expirationDate = expirationDate;
    }
}
```

The next thing to do is to design the `report.jrxml` file. You can do so manually in any text editor or using a design tool. Listing 17-123 shows the XML content of the report file.

Listing 17-123. *The report.jrxml File*

```
<?xml version="1.0" encoding="UTF-8"  ?>
<!-- Created with iReport - A designer for JasperReports -->
<!DOCTYPE jasperReport PUBLIC "//JasperReports//DTD Report Design//EN" ➥
"http://jasperreports.sourceforge.net/dtds/jasperreport.dtd">
<jasperReport
        name="catalogue"
         columnCount="1"
         printOrder="Vertical"
         orientation="Portrait"
         pageWidth="595"
         pageHeight="842"
         columnWidth="555"
         columnSpacing="0"
         leftMargin="20"
         rightMargin="20"
         topMargin="30"
         bottomMargin="30"
         whenNoDataType="NoPages"
         isTitleNewPage="false"
         isSummaryNewPage="false">
    <property name="ireport.scriptlethandling" value="0" />
    <property name="ireport.encoding" value="UTF-8" />
    <import value="java.util.*" />
    <import value="net.sf.jasperreports.engine.*" />
    <import value="net.sf.jasperreports.engine.data.*" />

    <field name="name" class="java.lang.String"/>
    <field name="expirationDate" class="java.util.Date"/>

        <detail>
    <band height="20">
     <textField>
        <reportElement x="170" y="0" width="200" height="20"/>
       <textFieldExpression   class="java.lang.String">➥
           <![CDATA[$F{name}]]></textFieldExpression>
```

```
     </textField>
<textField>
        <reportElement x="170" y="0" width="200" height="20"/>
       <textFieldExpression   class="java.lang.String">➥
            <![CDATA[$F{expirationDate}]]></textFieldExpression>
     </textField>

    </band>
  </detail>

</jasperReport>
```

Configuring JasperReports

To be able to use JasperReports in your Spring application, you will have to download latest versions of JasperReports and the following dependencies: BeanShell, Commons BeanUtils, Commons Collections, Commons Digester, Commons Logging, iText, and POI.

Next, we need to configure the view resolver; see Listing 17-124. For JasperReports, you should use `ResourceBundleViewResolver` and map view names to `View` classes in the properties file.

Listing 17-124. *The viewResolver Definition*

```
<bean id="viewResolver" class="org.springframework.web.➥
servlet.view.ResourceBundleViewResolver">
    <property name="basename" value="views"/>
</bean>
```

JasperReports Views

An overview of Spring's `View` implementation for JasperReports is given in Table 17-28.

Table 17-28. *Spring's JasperReportsView Implementations*

Class Name	Description
`JasperReportsCsvView`	Generates reports as comma-separated values files.
`JasperReportsHtmlView`	Generates reports in HTML format.
`JasperReportsPdfView`	Generates reports in PDF format.
`JasperReportsXlsView`	Generates reports in Microsoft Excel format.
`JasperReportsMultiFormatView`	Wrapper class that allows the report format to be specified at runtime. Actual rendering of the report is delegated to one of the other JasperReports view classes.

Mapping these `View` implementations is simply the case of adding them to the `views.properties` file (see Listing 17-125).

Listing 17-125. *view.properties for a JasperReport View*

```
simpleReport.class=org.springframework.web.➥
servlet.view.jasperreports.JasperReportsPdfView
simpleReport.url=/WEB-INF/reports/report.jasper
```

`SimpleReport` view is now mapped to PDF format report of the `report.jasper` file.

Using JasperReportsMultiFormatView

`JasperReportMultiFromatView` allows us to determine the format of the report at runtime. If you specify this as your view class, you have to provide mode parameter with the name `format` and a value that determines the report format. Spring has default values for standard JasperReports views. Table 17-29 shows these values.

Table 17-29. *Default Key Values for JasperReportViews in JasperReportMultiFormatView*

View Class	Format
`JasperReportsCsvView`	`csv`
`JasperReportsHtmlView`	`html`
`JasperReportsPdfView`	`pdf`
`JasperReportsXlsView`	`xls`

Let's now edit the `views.properties` file to add a multiformat view for the report (see Listing 17-126).

Listing 17-126. *view.properties for JasperReport View Using JasperReportsMultiFormatView*

```
Products-report.class=org.springframework.web.➥
servlet.view.jasperreports.JasperReportsMultiFormatView
products-report.url=/WEB-INF/views/en_GB/jasper/report.jrxml
```

And finally, let's add a handler method to our `ProductController`, as shown in Listing 17-127.

Listing 17-127. *Controller Code for JasperReports*

```
public ModelAndView pdfReportHandler(HttpServletRequest request,
        HttpServletResponse response) {
            Map model = new HashMap();
            String format = request.getParameter("format");
            JRBeanCollectionDataSource source = ➥
                    new JRBeanCollectionDataSource(products);
            model.put("products", source);
            model.put("format", format"pdf");
            return new ModelAndView("products-report", model);
    }
...
```

In our controller code, we use `JRBeanCollectionDataSource` to pass the data source to the JasperReports engine. This simple implementation maps `java.util.Collection` to the JasperReports data source (you can find more about JasperReports on the project's web site).

Note that we are using request `"format"` parameter to specify the report's format.

After mapping `pdfReport.html` to `pdfReportHandler()` and `xlsReport.html` to `xlsReportHandler()` using any of the `MethodNameReolvers`, if we point the browser to `http://localhost:8080/ch17/product/report.html?fomat=pdf`, we can access the PDF report. Figure 17-22 displays the browser output.

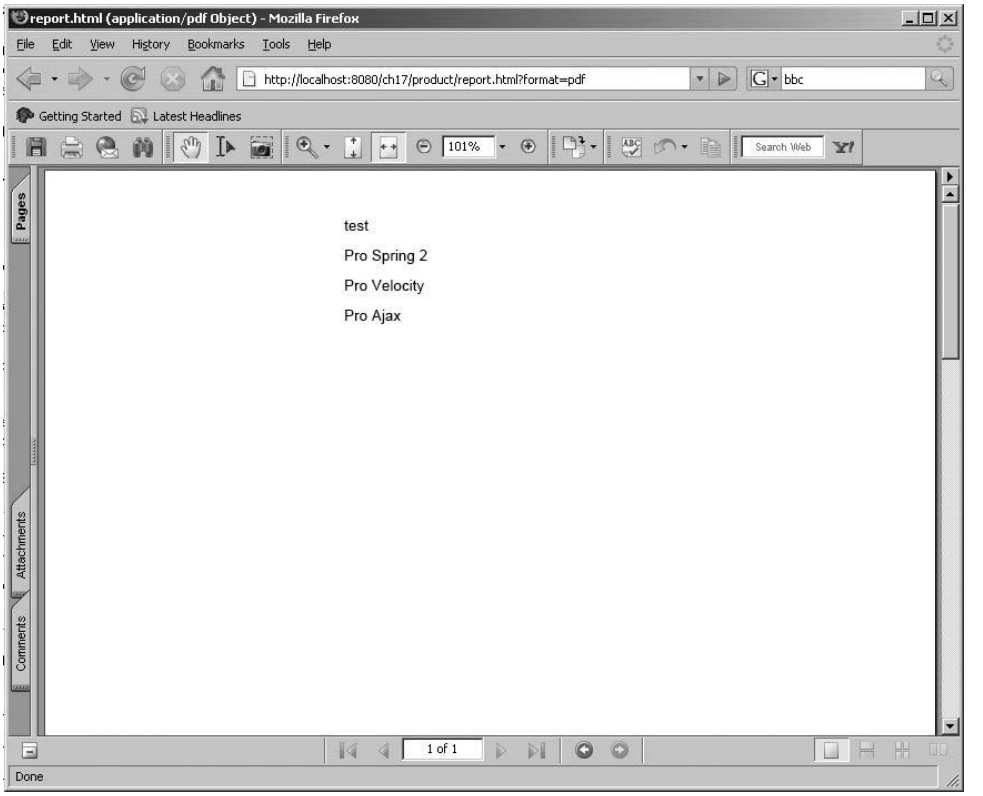

Figure 17-22. *JasperReportsPdfView browser output*

If we change format parameter to `xls`, we'll have `http://localhost:8080/ch17/product/report.html?format=xls`, as Figure 17-23 shows.

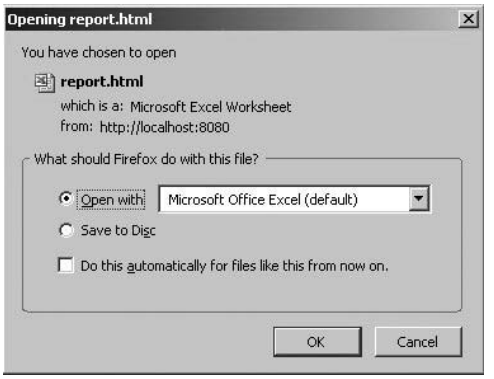

Figure 17-23. *JasperReportsExcelView browser output*

Configuring Export Parameters

Additional parameters are available in JasperReports to specify export options such as page size, headers, or footers. Using Spring, we can easily manage these parameters.

You can export additional parameters to your `JasperReportsView` using a simple declaration in the Spring configuration file. All you have to do is set the `exporterParameters` property of the view class. This property is actually a `Map`: the key is the fully qualified name of the static field from the `JasperReports` class, and the value should be the value you want to assign to the parameter (see Listing 17-128).

Listing 17-128. *ExporterParameters Declaration*

```
<bean id="report" class="org.springframework.web.➥
               servlet.view.jasperreports.JasperReportsHtmlView">
  <property name="url" value="/WEB-INF/reports/simpleReport.jrxml"/>
  <property name="exporterParameters">
    <map>
      <entry key="net.sf.jasperreports.engine.➥
                export.JRHtmlExporterParameter.HTML_FOOTER">
        <value>This is footer! </value>
      </entry>
    </map>
  </property>
</bean>
```

Here, you can see that the `JasperReportsHtmlView` is being configured with an export parameter for `net.sf.jasperreports.engine.export.JRHtmlExporterParameter.HTML_FOOTER`, which will output a footer in the resulting HTML.

Spring Conventions Over Configuration

For many smaller projects, configuring a Spring MVC model can be time consuming, even for fairly simple projects. Sometimes, we need to quickly develop a prototype, without worrying about all aspects of Spring MVC configuration. Imagine how much time you would save if you didn't have to configure all the handler mappings, view resolvers, model instances, views, and so on.

To that end, Spring developers came up with a convention-over-configuration setup to make programmers' lives easier. The convention separately supports core parts of MVC architecture, that is, the models, views, and controllers.

Controller Conventions

Spring 2.0 provides `ControllerClassHandlerMapping` class, which is basically an implementation of the `HandlerMapping` interface. This class uses the convention of mapping the requests' URL to controller classes based on names of controller classes in the Spring configuration file. Let's look at an example in Listing 17-129.

Listing 17-129. *ViewProductController Implementation*

```
public class ViewProductController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request, ➥
             HttpServletResponse response) {
       return new ModelAndView("/WEB-INF/views/product/viewProduct.jsp");
    }
}
```

Listing 17-130 shows the usual Spring context configuration for this controller.

Listing 17-130. *ViewProductController Spring Configuration*

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

<bean class="com.apress.ch17.web.ViewProductController" />
```

And that's it! If you point your browser to `/viewProduct.html`, you will get the content of the `/WEB-INF/views/product/viewProduct.jsp` file as implemented in `ViewProductController`. With this simple configuration, you have mapped the `viewProduct.html` URL to your `ViewProductController`—in fact, this convention maps all `/viewProduct*` URLs to the `ViewProductController`.

Behind the scenes, `ControllerClassNameHandlerMapping` looks for all `Controller` beans in the application context and defines its handler mappings by stripping the `"Controller"` string off its class name.

So, if we define another controller, lets say `IndexController`, it will automatically be mapped to the `/index*` request URL.

Note that `ViewProductController` (in camel case) is mapped to `/viewproduct*` (lowercase) request URLs.

MultiActionController Conventions

If we have a `MultiActionController` implementation, the convention is slightly different. Let's look at the examples in Listing 17-131 and Listing 17-132.

Listing 17-131. *AdminController Implementation*

```
public class AdminController extends MultiActionController {

    public ModelAndView usersHandler(HttpServletRequest request, ➥
            HttpServletResponse response) {
       //implementation not relevant
    }
public ModelAndView productsHandler(HttpServletRequest request, ➥
            HttpServletResponse response) {
       //implementation not relevant
    }
}
```

Listing 17-132. *AdminController Spring Configuration*

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

<bean id="internalPathMethodNameResolver"class="org.springframework.web.➥
            servlet.mvc.multiaction.InternalPathMethodNameResolver">
         <property name="suffix" value="Handler"/>
    </bean>
<bean id="adminController" class="com.apress.ch17.web.AdminController">
<property name="methodNameResolver" ref="internalPathMethodNameResolver"/>
</bean>
```

This configuration will map all request URLs like `/admin/*` to `AdminController`. The different methods inside `AdminController` will be mapped based on the `methodNameResolver` of the controller (in our example, `InternalPathMethodResolver`).

If you follow naming conventions for your `Controller` implementation (***`Controller`), which is the best practice in Spring, you can use this convention and have your web application up and running without having to maintain something like a `SimpleHandlerMapping` bean definition.

Model Conventions

Spring introduces the `ModelMap` class, which is actually a `Map` implementation that automatically generates the key for any object that is added to it. `ModelAndVew` implementation uses this `ModelMap`, allowing us to use it in Spring controllers. For scalar objects, the object key is generated from the short class name of the object added, with the first character in lowercase. Based on this, we have following rules:

- A `User` instance will have a `user key` generated for it.
- A `Product` instance will have a `product` key generated for it.
- A `java.util.HashMap` will have a `hashMap` key generated for it.
- Adding `null` will result in an `IllegalArgumentException` being thrown. If the object (or objects) that you are adding could potentially be `null`, you will also want to be explicit about the name.

If you're adding the object that is a `Set`, `List`, or array object, the map key is generated from the short class name of the first object contained in the `Set`, `List`, or array, followed by `List`. The general rules for key generation follow:

- A `User[]` array with one or more `User` elements added will have a `userList` key generated for it.
- An `x.y.User[]` array with one or more `x.y.Customer` elements added will have a `userList` key generated for it.
- A `java.util.ArrayList` with one or more `x.y.User` elements added will have a `userList` key generated for it.
- A `java.util.HashSet` with one or more `x.y.Customer` elements added will have a `customerList` key generated for it.
- An empty `java.util.ArrayList` will not be added at all (basically, it will be the no-op).

The `ModelMap` and the fact that it is part of `ModelAndView` class allow us to add objects to the model without the hassle of instantiating the `Map` and adding objects to it using invented keys. In Listing 17-133, you can see how easy it is to add objects to the model.

Listing 17-133. *Adding Objects to ModelMap in the Controller*

```
public class ViewOrderController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request, ➥
              HttpServletResponse response) {

        List orderItems = // get a List of OrderItem objects
        User user = // get the User making the order

        ModelAndView mav = new ModelAndView("viewOrder"); <-- the logical view name

        mav.addObject(orderItems);
        mav.addObject(user);

        return mav;
    }
}
```

This example will expose model that contains two objects, `orderItems` with the key `orderItemList` and `User` with key `user`.

View Conventions

In this section, we will show some conventions over configuration applied to the view. Let's take a look at the example in the Listing 17-134, which shows a `Controller` that returns `ModelAndView` without a view name being set.

Listing 17-134. *Controller Code Without a View Name*

```
public class RegistrationController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request, ➥
             HttpServletResponse response) {

        ModelAndView mav = new ModelAndView();

        return mav;
        // notice that no View or logical view name has been set
    }
}
```

At first glance, this code seems incorrect, because no `View` or view name has been set for the `ModelAndView` instance. But this code works!

The trick is in the `RequestToViewNameTranslator` interface introduced in Spring 2.0, which has one implementation, `DefaultRequestToViewNameTraslator`. To understand what's happening in Listing 17-134, we need to take a look at that example's Spring web application context in Listing 17-135.

Listing 17-135. *viewNameTranslator Configuration*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
...">
    <bean id="viewNameTranslator" class="org.springframework.web.➥
              servlet.view.DefaultRequestToViewNameTranslator"/>

    <bean class="com.apress.prospring2.ch17.web.ProductController">
    </bean>

    <bean class="org.springframework.web.➥
          servlet.mvc.support.ControllerClassNameHandlerMapping"/>

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

</beans>
```

If the `View` has not been set and no logical view name is passed to the `ModelAndView` instance, the `DefaultRequestToViewNameTraslator` will kick in. The `RegistrationController` in our example is used with `ControllerClassNameHandlerMapping`, which maps it to the `/registration*` request

URLs. The `/registration.html` URL will be mapped to the `RegistrationController`, which will in turn set default logical view to `registration` (generated by `DefaultRequestToViewNameTraslator`). This logical view will be resolved into `/WEB-INF/views/registration.jsp`, by the defined view resolver.

The `DefaultRequestToViewNameTraslator` strips the leading slash and file extension of the URL and returns the result as the logical view name.

Another example is `/admin/users.html`, which resolves to the logical view `admin/index`.

You can even skip the definition of the `viewNameTranslator` bean in the configuration. If no `viewNameTranslator` bean is explicitly defined, Spring `DispatchedServlet` will instantiate the `DefaultRequestToViewNameTraslator` itself. However, if you want to customize default settings of the `DefaultRequestToViewNameTraslator` or to implement your own `RequestToViewNameTraslator`, you will have to explicitly define the bean.

Using Annotations for Controller Configuration

Using annotations instead of XML-style configurations has become a trend in Java programming. Spring 2.5 introduces an annotation-based programming model for MVC controllers. 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. We will discuss Servlet MVC in this section.

@Controller

The `@Controller` annotation simplifies controller class declaration. There is no need to implement a `Controller` interface, extend any Spring controller class, or even reference the Servlet API. All you have to do is to add the annotation to your `Controller` class as shown in Listing 17-136.

Listing 17-136. *@Controller Annotation in IndexController*

```
@Controller
public class IndexController{

    public ModelAndView displayIndex(
  //omitted for clarity
    }

}
```

To be able to use this annotation, you have to add component scanning to your configuration files, as shown in Listing 17-137.

Listing 17-137. *@Controller Annotation Configuration*

```
<?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-2.5.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd">
```

```
    <context:component-scan base-package="com.apress.prospring2.ch17.web" />

    ...

</beans>
```

@RequestMapping

The `@RequestMapping` annotation is used to map URLs to the `Controller` class or a particular method (in `MultiActionControllers`, for example). Listing 17-138 provides an example.

Listing 17-138. *@RequestMapping Annotation in IndexController*

```
@Controller
@RequestMapping("/index.html")
public class IndexController{

    public ModelAndView displayIndex(
  //omitted for clarity
    }

}
@Controller

public class IndexController{

 @RequestMapping("/product/list.html")
    protected ModelAndView listProductsHandler(
  //omitted for clarity
    }
@RequestMapping("/product/view.html")
    protected ModelAndView viewProductHandler(
  //omitted for clarity
    }

}
```

If you want to use this mapping as a class-level annotation, you will have to configure `AnnotationMethodHandlerAdapter` for your servlet. If you want to use the `@RequestMapping` annotation at the method level, you need to configure `DefaultAnnotationHandlerMapping`. This is done by default for `DispatcherServlet`, but if you are implementing your own handler adapter, you will have to define `AnnotationMethodHandlerAdapter`—just as if you use custom handler mapping, `DefaultAnnotationHandlerMapping` must be defined in the Spring configuration files. Listing 17-139 shows an example of this configuration.

Listing 17-139. *@RequestMapping Configuration*

```
<?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-2.5.xsd
        http://www.springframework.org/schema/context
```

```
        http://www.springframework.org/schema/context/spring-context-2.5.xsd">
    <context:component-scan base-package="com.apress.prospring2.ch17.web" />
    <bean class="org.springframework.web.➥
        servlet.mvc.annotation.AnnotationMethodHandlerAdapter"/>
   <bean class="org.springframework.web.➥
        servlet.mvc.annotation.DefaultAnnotationHandlerMapping"/>

    ...

</beans>
```

@RequestParam

The `@RequestParam` annotation is used to bind request parameters to a method parameter in the controller; Listing 17-140 shows an example.

Listing 17-140. *@RequestParam Annotation*

```
@Controller
@RequestMapping("/product/edit.html")
public class EditProductController {

        @RequestMapping(type = "GET")
        public String setupForm(@RequestParam("productId") int productId, ➥
                      ModelMap model) {
                Product product = this.productManager.findProductById(productId);
                model.addAttribute("product", product);
                return "productForm";
        }
```

All parameters used with this annotation are mandatory by default. If you want to add an optional parameter, set the `@RequestParam` annotation's `required` attribute to `false`: `@RequestParam ("productId", required="false")`

@ModelAttribute

If your method returns the `Object` that is going to be used in your model, you can annotate it with the `@ModelAttribute` annotation. This annotation will populate the model attribute with the return value of the method, using its parameter, the model name, as shown in Listing 17-141.

Listing 17-141. *@ModelAttribute Annotation Example*

```
        @ModelAttribute("products")
        public Collection<Product> populateProducts() {
                return this.productManager.findAllProducts();
        }
```

This annotation can also be placed as a method parameter. In that case, the model attribute with a specified name is mapped to method parameter. This can be used to get the command object after filling in the HTML form (see Listing 17-142).

Listing 17-142. *Using @ModelAttribute to Get the Command Object in the Form Controller*

```
        public String processSubmit(@ModelAttribute("product") Product product, ➥
                       BindingResult result,SessionStatus status) {
               this.productManager.saveProduct(product);
```

```
            return getSuccessView();
        }
```

Using Annotations with the Command Controller

Let's now look at how can we use an annotation-based controller for functionality that was usually implemented using a command controller (like the simple form submission).

Listing 17-143 shows the annotation-based implementation of `ProductFormController`:

Listing 17-143. *Annotation-Based ProductFormController Implementation*

```
@Controller
@RequestMapping("/products/edit.html")
@SessionAttributes("product")
public class ProductFormController {

    @Autowired
    private ProductService productService;

    @ModelAttribute("types")
    public Collection<String> populateProductTypes() {
            List<String> types = new ArrayList<String>();
            types.add("Books");
            types.add("CDs");
            types.add("MP3 Players");

            return types;
    }

    @RequestMapping(method = RequestMethod.GET)
    public String showForm(@RequestParam("productId") int productId, ➥
ModelMap model) {
        Product product;
        if(productId == 0)
            product = new Product();
        else
            product = this.productService.findById(productId);
        model.addAttribute("product", product);
        return "products-edit";
        }

    @RequestMapping(method = RequestMethod.POST)
    public String submit(@ModelAttribute("product")Product product, ➥
BindingResult result,SessionStatus status) {

        this.productService.save(product);
        return "products-index";
    }

}
```

To the controller in the preceding listing, we have added three class-level annotations: `@Controller` to mark the class as `Controller`, `@RequestMapping` to set the URL that maps to this controller, and `@SessionAttribute` to set the name of our command object to be used in the view.

The method `public String showForm()` shows the actual form view. This method is annotated with `@RequestMapping(method = RequestMethod.GET)`, which tells Spring to call it only when the request method is `GET`. It takes the `productId` request parameter (as defined with `@RequestParam` annotation), which we use to create the command object named `product`. Since we have annotated the controller with `@SessionAttribute("product")`, the value of this attribute will be saved in the session. Listing 17-144 shows the actual form.

Listing 17-144. *The Product Editing Form*

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

<form:form commandName="product" action="editProduct.html">
    <table>
        <tr>
            <td>Product name:</td>
            <td><form:input path="name" /></td>
        </tr>
         <tr>
            <td>Product type:</td>
            <td><form:select path="type">
            <form:option value="-" label="--Please Select"/>
            <form:options items="${types}"/>
        </form:select></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form:form>
```

The method `public Collection<String> populateProductTypes()` populates the `productTypes List` with values that are later used as options in the form's select input field. The `@ModelAttribute (?QUOT?types?QUOT?)` method-level annotation tells Spring to include the return value of this method (that is, a `java.util.Collection` of product types) in the `ModelMap` for this model.

The `public String submit()` method is called whenever `POST` is the request method. It takes as a parameter a `Product` that is directly loaded from the model (annotated with `@ModelAttribute ("product")`). Now, we can save the product we got from the attribute and return the success view (in our case, the `products-index` view defined in the `views.properties` file introduced in Tiles section of this chapter).

Summary

In this chapter, you have learned how to use Spring MVC architecture to build flexible and powerful web applications. You know how to use Spring to configure your controllers; you know which controller to use for different usage scenarios. You also know how to validate the data the users enter on the forms, build applications that display the output in the user's language, and make the user's experience even better by providing themes. File upload is now quite a trivial task to implement.

In this chapter, you have also learned about the view technologies available for use in a Spring application and saw that there's a lot more to generating web output than using JSP pages. You can use Velocity if you need top performance or FreeMarker for macro support; you can control PDF or Excel file output if you need to ensure that the formatting is the same on all browsers or to generate output that users will print out. You also know how to set up Tiles to integrate all the different views

together. We have also introduced Spring 2.5 features like the Spring form tag library, convention-over-configuration methodology, and annotation-based controllers.

If you are interested in mastering Velocity, check out Rob Harrop's book *Pro Jakarta Velocity* (Apress, 2004), which covers the Velocity Template Language and its uses as well as all the internals of Velocity.