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.

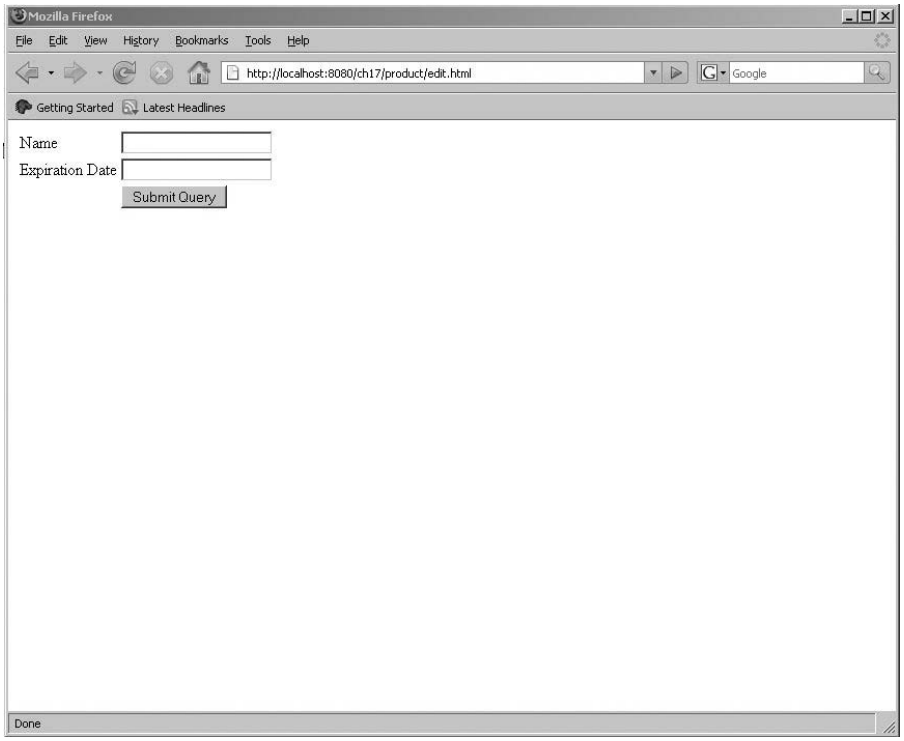

Figure 17-8. *The product editing form*

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.

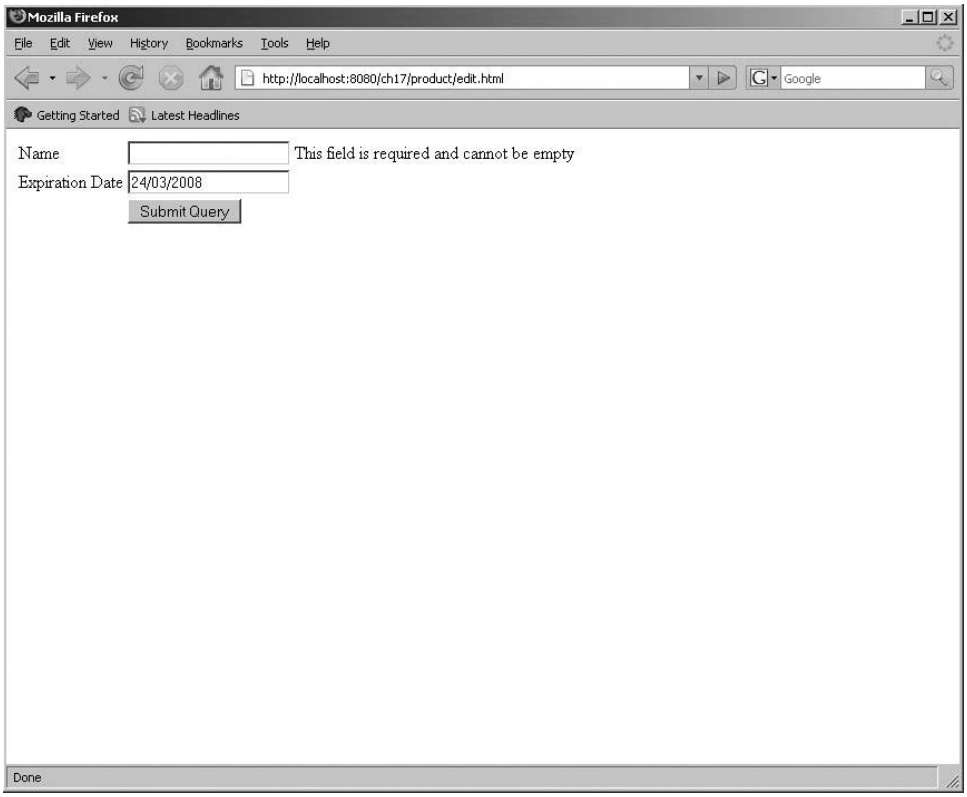

Figure 17-9. *The edit page with validation errors*

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.

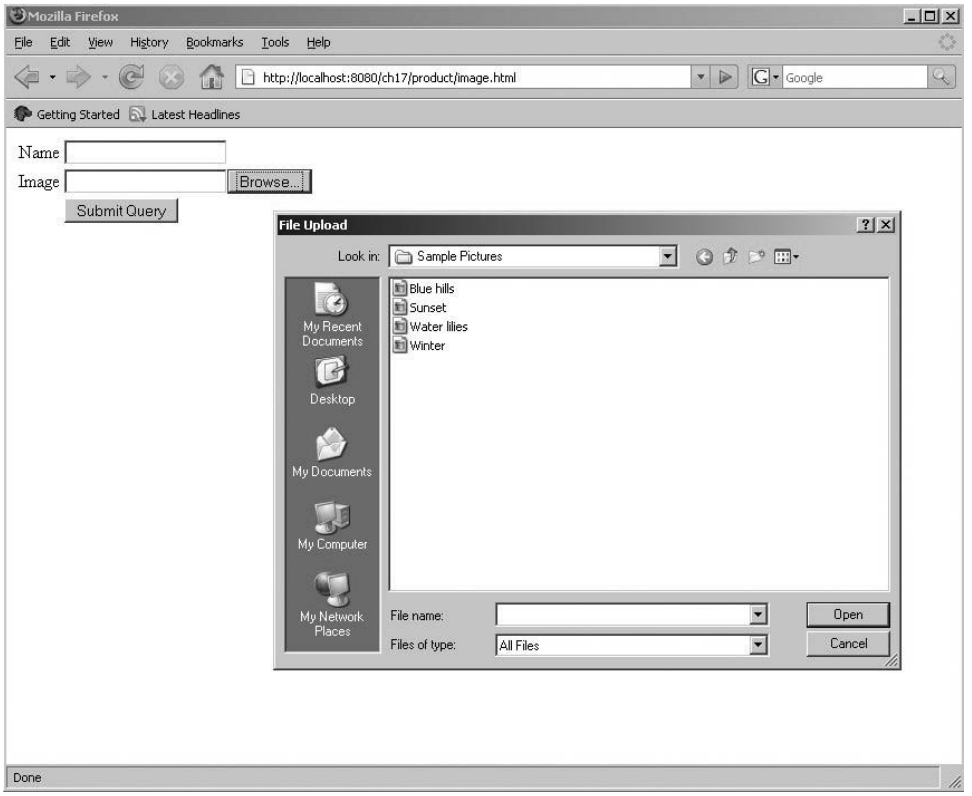

Figure 17-10. *File uploading*

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>
```