CHAPTER

4

Struts TagLibs and JSPs

TagLibs are the extension mechanism for JSPs. Struts takes advantage of this mechanism to provide a comprehensive set of TagLibs that ease the development of view components. Used correctly, they can help promote a cleaner separation of model, view, and controller components and simplify JSPs by eliminating the need for most scriptlet code. Pitfalls in using these TagLibs usually arise from a lack of specific knowledge of how the TagLibs that Struts provides are intended to be used, and from a lack of understanding of TagLibs in general.

TagLibs are architecturally part of the view of the application and are best used as such. Due to TagLibs' inherent flexibility however, developers somtimes introduce model or controller code into their JSPs. When TagLibs are used to blur the lines between the model, view, and controller components, it's a good sign that some pitfall-infected code is creeping into your JSP. This chapter examines these and other pitfalls developers may encounter in coding JSPs using the Struts TagLibs.

Pitfall 4.1: Hard-Coded Strings in JSPs describes the habit of many developers to hard-code string values directly into their JSPs. The pitfall shows its teeth when *name* shows up more than once, resulting in a dual maintenance point.

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 198

Pitfall 4.2: Hard-Coded Keys in JSPs is related to Pitfall 4.1 because developers will often turn from hard-coded strings to this pitfall and hard-code the names of the properties into their JSPs. This usually turns into a maintenance hassle over time.

Pitfall 4.3: Not Using Struts Tags for Error Messaging describes the habit of developers to manage error messages themselves, resulting in developers doing manually what is already provided (that is, tested and well designed) in Struts.

Pitfall 4.4: Calculating Derived Values in JSPs examines the consequences of using scriptlets to perform calculations in JSPs.

Pitfall 4.5: Performing Business Logic in JSPs involves an even more serious form of blurring the lines between the models, views, and controllers in a Struts application. If business logic is in a JSP, then each point in the user interface that needs that processing will have to have a copy of the code—definitely not recommended.

Pitfall 4.6: Hard-Coded Options in HTML Select Lists describes the problems developers often encounter with HTML select lists in Struts. Many developers find the Struts html:options tag confusing when they initially encounter it, and they simply find it expedient to code the values that they need directly in their JSP using the html:option tag instead. This leads to a lot more code than is required, thereby potentially increasing maintenance costs.

Pitfall 4.7: Not Checking for Duplicate Form Submissions addresses a common scenario involving the lack of synchronization between the interface presented in a browser and the state of the model on the server. This has the potential to corrupt the underlying data store by creating duplicate records, overwriting current data with stale values, and more.

Pitfall 4.1: Hard-Coded Strings in JSPs

Struts provides a tag for rendering string values by first looking them up in a resource bundle. Many developers, though, hard-code the strings directly into their JSPs. When the JSPs are first being written it may not be apparent that anything else could or should happen—if the string “Name” appears on only one form in a Web application, then why would we want to have to look up that string to get it onto the Web page? The pitfall shows its teeth when “Name” shows up more than once. If that string is hard-coded in both places and if it ever needs to be changed (for example, to “First Name”) there is a dual maintenance point. Whenever a dual maintenance point arises you can be fairly confident that a pitfall is lurking somewhere, making your life more difficult than it needs to be.

When the same instructional text, field label, or other string value appears to vary depending on which page the user is looking at, you are experiencing this pitfall. You will also notice that it is often painful and time-consuming to make simple changes and that far too many defect tickets are written against trivial JSP coding errors during system test.

This pitfall will be much more obvious if your development team faces language localization requirements. If your JSPs are trapped in this pitfall, each one will have to be translated individually. Usually the local language version of the user interface has already been coded by the time support for additional languages is addressed. If you have to translate your pages, don’t brute-force the translation—apply the solution first.

Generally, this pitfall affects those newer to Struts who are usually distracted by other, more pressing issues during the earlier stages of development. The typical developer new to Struts is usually just trying to keep his or her head above water while riding the learning curve for Struts’ major architectural aspects, much less trying to learn many of the seemingly unimportant tags in the extensive TagLib. Here’s a simple example of a hard-coded string that might appear in a JSP.

```
...
<h1>Modify Invoice</h1>
...
```

At first pass, this might appear to be the right solution; the heading is Modify Invoice, and that is what it will remain for the foreseeable future. Why would you want to look it up? Well, let’s go back to the example of translating the application into multiple languages and suppose that Modify Invoice appears on several pages. If localization is a requirement for your application, each page that the string appears on will have to be translated. Modifying all the pages will be an expensive proposition.

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 200

Example

Listing 4.1 provides an example of a JSP caught in this pitfall. Notice all the hard-coded strings.

The first hard-coded string is somewhat innocuous because it is probably unique to the current JSP. Unique strings can sometimes be the exception to the rule on hard-coding strings in JSPs (for apps that *don't* require localization) because they appear only once.

The next hard-coded string in the example is instructional text that may be used in several similar pages in the application. If the requirements for the text to be displayed change, every JSP the string appeared in will have to be updated. If the same text appears in more than a few places, it would be easy to miss an instance or mangle it, potentially resulting in bug reports being written against the affected page during system test. That, of course, means that you'll have to rummage through those same pages to make the same edits again.

Finally, we see a couple of hard-coded field labels. These same labels may occur on many other pages, sometimes in more than one place on a given page. Finding all occurrences of a given label and changing them all by hand can be quite tedious and has the potential to introduce errors.

Some content may be truly unique to a given page, and in that case there is little to be gained by abstracting it away (unless there are language localization requirements). Applications with a small page count are also less vulnerable to the downsides of this pitfall, particularly if there is little overlap between the content of the various pages.

```
...
<h1>Modify Invoice</h1> <%-- HARDCODED --%>
<p>
Modify one or more of the values below and press Submit<%-- HARDCODED --
%>
<p>
<html:form  action="/saveInvoice" method="post">
    <table>
        <tr>
            <td>Invoice Number: <td> <%-- HARDCODED --%>
            <td><html:text property="invoiceNumber"/></td>
        </tr>
        <tr>
            <td>Invoice Date: <td> <%-- HARDCODED --%>
            <td><html:text property="billingDate"/></td>
        </tr>
...
```

Listing 4.1 Hard-coded strings in a JSP.

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 201

To avoid this pitfall altogether, spend some time early in the project thinking through how strings will be presented in your JSP pages and setting guidelines for using resource bundles.

Solving Pitfall 4.1: Move Common Strings to Resource Bundles

In solving this pitfall, our goal is to allow for sharing of common string values across multiple JSPs. This solution is essential if your application has language localization requirements, but it is helpful in any case because it provides a centralized location for the storage of UI strings so that they can be shared by multiple JSPs, rather than be copied into each of the JSPs.

Step-by-Step

1. Create a .properties file.
   a. Place the .properties file somewhere in the path at runtime. It is typical to place the .properties file alongside the struts-config.xml file in the WAR file.
2. Begin with a single JSP.
   a. Start with the simplest JSP.
3. Search for hard-coded strings.
   a. Any string that will appear in the final HTML on the user's browser is a candidate to be a hard-coded string.
4. Create an entry in the .properties file for each unique string.
5. Replace the hard-coded strings with bean:message tags.
6. Deploy and test the changes.
7. Repeat for each JSP.

Example

In this example, we will walk through the steps laid out earlier to solve this pitfall. The first step is to create a messages.properties file to hold the strings. The next step is to choose a simple JSP to start with. We will begin with Listing 4.1. The next step is to identify all the hard-coded strings and put them into the properties file that we created. Here is a listing of the properties that are in the file.

```
...
example.title=Jakarta Pitfalls Example
invoice.modify.heading=Modify Invoice
label.invoice.number=Invoice Number:
label.billing.date=Billing Date:
instruction.modify=Modify one or more of the values below and press
Submit
...
```

The next step for this JSP is to replace the hard-coded values in the JSP with bean:message tags. Listing 4.2 is a partial listing of the code for the JSP with the strings replaced with bean:message tags.

Notice that the newly improved JSP does not have any display strings hard-coded in it. If localization ever becomes an issue for this page, then it will be easy to translate because all the user visible strings are in one file. In this listing the display strings are no longer hard-coded, but the names of the keys in the properties file are. Pitfall 4.2 addresses this issue.

```
...
<head>
    <title><bean:message key="example.title"/></title>
</head>

<body bgcolor="white">
<h1><bean:message key="modify.heading"/></h1>
<p>
<bean:message key="instruction.modify"/>
<p>
<html:form  action="/saveInvoice" method="post">
    <table>
        <tr>
            <td><bean:message key="label.invoice.number"/><td>
            <td><html:text property="invoiceNumber"/></td>
        </tr>
        <tr>
            <td><bean:message key="label.billing.date"/><td>
            <td><html:text property="billingDate"/></td>
        </tr>
...
```

Listing 4.2 JSP with bean:message tags.

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 203

Pitfall 4.2: Hard-Coded Keys in JSPs

This pitfall is related to Pitfall 4.1 because what usually happens (as we saw in the solution to Pitfall 4.1) is that a developer will turn from hard-coded strings to this pitfall and hard-code the names of the properties into his or her JSPs. This usually turns into a maintenance hassle over time. Suppose for example that the initial naming scheme for properties is outgrown, and a new naming scheme is introduced. The new naming scheme will require the old properties to be renamed. Searching through each JSP looking for each instance of the old property name becomes very tedious very quickly.

As an application grows in size and complexity, the burden required to maintain the hard-coded strings grows as well. Another issue is that the hard-coded values don't scale up to large teams well. Even though this pitfall seems rather harmless, it can lead to countless hours of tedious work to rename a single property.

Example

Here is the solved code from Listing 4.2. It is stuck in this pitfall because the names of the properties are hard-coded in the JSP.

```
...
<head>
    <title><bean:message key="example.title"/></title>
</head>

<body bgcolor="white">
<h1><bean:message key="modify.heading"/></h1>
<p>
<bean:message key="instruction.modify"/>
<p>
<html:form  action="/saveInvoice" method="post">
    <table>
        <tr>
            <td><bean:message key="label.invoice.number"/><td>
            <td><html:text property="invoiceNumber"/></td>
        </tr>
        <tr>
            <td><bean:message key="label.billing.date"/><td>
            <td><html:text property="billingDate"/></td>
        </tr>
...
```

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 204

Over time, we might decide that we want the properties used for labels in our forms to be named in a consistent three-part naming scheme where the first part of the name is an action of the string, the second part identifies a part of the model, and the third part of the name identifies the kind of string. Under this scheme the properties names would look like this:

```
...
modify.invoice.heading=Modify Invoice
label.invoice.number=Invoice Number:
label.billing.date=Billing Date:
modify.invoice.instruction =Modify one or more of the values below and
press Submit
...
```

If each of these names were spread out over several JSPs, we'd have to go to each JSP and update each occurrence of the property names. If you find yourself doing similar things, you are stuck in this pitfall.

Solving Pitfall 4.2: Replace Hard-Coded Keys with Constants

To solve this pitfall, the constant Strings should be captured in a Java constants class. One really nice thing about this solution is that it allows developers to take advantage of the syntax-checking facilities of IDEs in order to avoid runtime errors caused by invalid keys. The IDE can validate that Constants.MY_PROPERTY_NAME is a real attribute on the Constants class, but the IDE cannot validate my.property.name. Applying this solution also makes maintaining property definitions easier. If a key is changed in the properties file, only the constants class must be updated. Instead of performing a global search and replace on the key name in all of the JSPs, one simple value change is all that is needed.

Something to consider as you are applying the solution to this pitfall is not to go overboard on putting all your constants into one Java class. On a medium-sized project, you can end up with hundreds of properties; if each of those properties ends up in one Constants class, the class will then itself become a pitfall. Group the constants into related properties, and put the groups into a class together.

I've been on projects where all the message constants are jumbled into one Constants file that also contains constants for request/session attributes and every other imaginable use for a constant. I think it's much more manageable to have a MessageConstants.java file that contains only message key constants.

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 205

Step-by-Step

1. Create a Constants class.
   a. Depending on which approach is taken to the replacement of constant keys in the JSP, you might also need to create accessors for each property constant.
2. Begin with a single, simple JSP.
   a. Keep in mind that you will be repeating these steps, so it is best to start simple and work your way up.
   b. Another equally valid approach is to go through your home page, then each page that is accessible from there and so on until you have covered all the pages.
3. Search for hard-coded keys.
4. Create a new constant in the Java class for each unique key.
   a. Keep in mind that if there are a lot of constants, say more than 30 or 40 as a rule of thumb, the constants should be grouped and split into separate classes.
5. Replace each hard-coded key with a reference to the appropriate constant.
   a. Or use an expression referring to the property.
6. Test the changes.
7. Continue with the next JSP, until all keys have been replaced with the constant.

Example

In this example, we will apply the solution to the same JSP that we looked at earlier in the pitfall (Listing 4.2). Here is the code for the JSP that is stuck in the pitfall for reference.

```
...
<head>
    <title><bean:message key="example.title"/></title>
</head>

<body bgcolor="white">
<h1><bean:message key="modify.heading"/></h1>
<p>
<bean:message key="instruction.modify"/>
```

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 206


```
<p>
<html:form  action="/saveInvoice" method="post">
    <table>
        <tr>
            <td><bean:message key="label.invoice.number"/><td>
            <td><html:text property="invoiceNumber"/></td>
        </tr>
        <tr>
            <td><bean:message key="label.billing.date"/><td>
            <td><html:text property="billingDate"/></td>
        </tr>
...
```

The first step is to create a Constants class that will contain the strings. We will skip grouping the constants for brevity, but keep that in mind on larger projects. You have several choices for how to group the properties. For example, the properties can be grouped in functional areas; in an online store, for example, the areas would include a shopping cart section, a catalog section, an account management section, and probably others. Another equally valid way is to group the constants along functional lines, like shopping, purchasing, and shipping. Just about any way you organize the constants is fine as long as the method is well documented so that others can follow it. Just remember that the idea is to keep from building a single class with 400 static variables in it.

Now that we have chosen the JSP to work with first, the next step is to find the list of hard-coded property names. Here is the list of properties that are referred to in the JSP.

```
example.title
modify.heading
instruction.modify
label.invoice.number
label.billing.date
```

The next step is to put the constants into the Java class that we created. Here is the code for that class.

```
...
public abstract class Constants {
    /** The key to the string displayed in the browser's title bar */
    public final static String BROWSER_TITLE_KEY =
        "example.title";
    /** The key to the heading for the Modify Invoice page */
    public final static String MODIFY_INVOICE_HEADING_KEY =
        "modify.heading";
    /** The key to instructional text for forms users can modify */
```

```
    public final static String MODIFY_INSTRUCTION_KEY =
        "instruction.modify";
    /** The key to the label for the <b>invoice number</b> field */
    public final static String INVOICE_NUMBER_LABEL_KEY =
        "label.invoice.number";
    /** The key to the label for the <b>billing date</b> field */
    public final static String BILLING_DATE_LABEL_KEY =
        "label.billing.date";
}
```

The next step is to replace the hard-coded key names in the JSPs with the constants we just created. There are two ways to do this. The first involves scriptlets; the second involves the JSP expression language. If you are able to use the expression language, then you should do so because it provides a cleaner separation of Java code from the JSP. If you are unable to use the expression language, then use scriptlet code in your JSP.

Using scriptlets involves first adding an *import* statement to the JSP. That import should look like this:

```
<%@ page import="com.aboutobjects.pitfalls.Constants" %>
```

Next, the hard-coded strings are replaced with references to the constants defined in Constants.java:

```
<head>
    <title><bean:message
key="<%=Constants.BROWSER_TITLE_KEY%>"/></title>
</head>

<body bgcolor="white">

<h1><bean:message key="<%=Constants.MODIFY_INVOICE_HEADING_KEY%>"/></h1>
<p>
<bean:message key="<%=Constants.MODIFY_INSTRUCTION_KEY%>"/>
<p>
<html:form  action="/saveInvoice" method="post">
    <table>
        <tr>
            <td><bean:message
                     key="<%=Constants.INVOICE_NUMBER_LABEL_KEY%>"/><td>
            <td><html:text property="invoiceNumber"/></td>
        </tr>
        <tr>
            <td><bean:message
                     key="<%=Constants.BILLING_DATE_LABEL_KEY%>"/><td>
            <td><html:text property="billingDate"/></td>
        </tr>
...
```

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 208

The final step is to deploy and test the newly modified JSP. It is always a good idea to test each individual JSP before moving on to the next. The final step is to start over and do it all again for the next JSP.

If you are able to use the JSP expression language, the approach is very similar. Instead of using a scriptlet, you write a JSP expression to access the constant value. The scriptlet engine works against JavaBeans, so we must add a bean façade over the Constants class that we already created. Here is the additional code that must be added to the Constants class.

```
public abstract class Constants {
    public Constants() {
    }
    public String getBrowserTitleKey() {
        return BROWSER_TITLE_KEY;
    }
    public String getModifyInvoiceHeadingKey() {
        return MODIFY_INVOICE_HEADING_KEY;
    }
    // Continue on with this approach for each key added to the
    // constants file
    ...
}
```

Now we'll modify the JSP to use the expression language:

```
<jsp:useBean id="constants"
             class="com.aboutobjects.pitfalls.Constants"/>
<head>
    <title><bean:message key="${constants.browserTitleKey}"/></title>
</head>

<body bgcolor="white">

<h1><bean:message key="${constants.modifyInvoiceHeadingKey}"/></h1>
```

You can now continue through the JSP, replacing scriptlet code with corresponding expressions. It should be pointed out that the choice between scriptlets and expression language for this particular usage is not terribly meaningful. Keep in mind, though, that scriptlets represent a maintenance issue for non-Java programmers. Web developers know the Web-based languages (XHTML, JavaScript, etc.) but do not usually know Java. Having Java embedded in the JSP sometimes forces the Web developer to wait for a Java developer to consult before he or she can make a change. If it is possible to use the expression language, you are better off doing that.

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 209

Pitfall 4.3: Not Using Struts Tags for Error Messaging

Struts provides a fairly comprehensive set of facilities for managing and rendering error (and other) messages. Developers, however, may have already acquired the habit of managing these messages themselves through working in other JSP environments, or they simply may not have had sufficient time to understand how to use the Struts messaging facilities. This results in the developers doing manually what is already provided (that is, tested and well designed) in Struts. It is generally better to use an existing piece of well-tested software than to build your own solution.

The symptoms of this pitfall typically are inconsistent error messaging, extra copy/paste code to format values to be displayed as part of the message, and difficulty maintaining scattered, hard-coded messages.

This pitfall typically arises because the Struts error-messaging mechanism is a bit difficult to grasp at first and the documentation is sparse. Developers often find themselves wrestling with more pressing design issues, so error messaging may get short shrift early in the life of the project.

This pitfall usually shows up in two places. First, the JSP has a logic:present tag that looks for a marker saying that an error occurred. Second, the Action class notices error conditions and sets the marker so that the JSP will find and display it. Here is some typical code you'd find in an application trapped in this pitfall.

```
<logic:present name="errorMessage">
    <bean:write name="errorMessage"/>
</logic:present>
try {
    ...
}
catch (SomeException e) {
    request.setAttribute("errorMessage", "Some error message");
    ...
}
```

Keep in mind that this is just one of several forms that this pitfall may take. In general, any time you find yourself managing error messages in your application and you are not using the Struts error tags, you are probably stuck in this pitfall.

Example

Here is another example of an Action class and a JSP that are stuck in the pitfall. The try/catch block from the FindInvoiceAction is putting the error messages directly in the request:

```
try {
    InvoiceDelegate delegate = InvoiceDelegate.getInstance();
    invoice = delegate.findInvoice(bean);
}
catch (InvoiceDelegate.NotFoundException e) {
    request.setAttribute("errorMessage",
        "Unable to find invoice number " + invoiceNumber);
    return mapping.findForward("findInvoicePage");
}
```

And the ModifyInvoice.jsp is looking for the request value and placing it into the output if it is found:

```
<logic:present name="errorMessage">
    <bean:write name="errorMessage"/>
</logic:present>
```

This works perfectly well (as long as you don't have any localization requirements), but it is difficult to maintain as messages have to be copy/pasted in numerous places, along with any formatting code. Of course, you could pull the strings out into a properties file and use the java.text.MessageFormat class to allow for parameterization and formatting of values, but then you'd have to embark on the path of re-implementing existing Struts functionality, which would obviously be counterproductive.

To avoid this pitfall use Struts ActionError and ActionMessage classes to generate user interface messages, and reference them with html:errors and html:messages tags in your JSPs.

Solving Pitfall 4.3: Replace Custom Messaging with Struts Messaging

The Struts messaging functionality allows us to maintain our message strings in properties files, and it automatically supports parameterized messages, so we can supply values that will be plugged into the strings when necessary. Instead of coding this ourselves in Action classes and JSPs, we should use the functionality that comes with Struts. Using the error mechanism also neatly handles keeping the strings in one place for easy maintenance and localization (if we need localization). At the same time, this solution will allow us to simplify the code in our Actions.

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 211

Step-by-Step

1. Start with the simplest Action containing custom messaging code.
2. Move the error strings to the appropriate properties file.
   a. Add placeholders for dynamic values as necessary.
3. Add a constant to your constants class for the new property key.
   a. Don't forget to add the simple accessor method so that the property can be used in a JSP expression statement.
4. Replace messaging implementation with code to generate ActionError or ActionMessage instances, and place them in the appropriate scope.
5. Identify associated JSPs.
6. Modify rendering code in the JSPs to display the Struts messages.
7. Test the new implementation.
8. Search for additional Actions that contain custom messaging code and repeat the steps as necessary.

Example

The first step is to identify the simplest Action class to start with. For this example, we will solve the example presented in the pitfall. Here's the code snippet from the Action:

```
try {
    InvoiceDelegate delegate = InvoiceDelegate.getInstance();
    invoice = delegate.findInvoice(bean);
}
catch (InvoiceDelegate.NotFoundException e) {
    request.setAttribute("errorMessage",
        "Unable to find invoice number " + invoiceNumber);
    return mapping.findForward("findInvoicePage");
}
```

The next step is to move the message string to ApplicationResources. properties:

```
error.find=Unable to find invoice with invoice number {0,number,#}.
```

Note that we have added an expression (not to be confused with JSP expression language) at the end of the line to act as a placeholder for the invoice number.

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 212

FORMATTING MESSAGES

Struts uses the class java.text.MessageFormat to perform substitution on bracketed expressions in message strings. The ActionMessage class and its subclass ActionError allow you to provide an array of arguments that will be used by MessageFormat to replace the bracketed expressions. There are also convenience methods that take one to four explicit arguments.

In the preceding example, we used a couple of optional features of MessageFormat to control how one of the values was formatted. Note that the only value MessageFormat requires inside the curly braces is a number specifying by position the parameter to be applied. The additional values in the example represent the type of formatting to apply (for example, *number* in this case), as well as a formatting pattern to use. We specified #, which will prevent MessageFormat from inserting commas as thousands separators. (For more details, see the javadoc for java.text.MessageFormat.)

The next step is to add a line to our Constants class to refer to the key. If you're using expression language in your JSP, you will also need to add an accessor method.

```
/** The key to the <i>not found</i> error message */
public final static String FIND_ERROR = "error.find";
...
public String getFindError() {
    return FIND_ERROR;
}
```

The next step is to rewrite the catch block to generate an ActionError instead of putting the error string into the response.

```
catch (InvoiceDelegate.NotFoundException e) {
    ActionError error =
        new ActionError(Constants.FIND_ERROR, invoiceNumber);
    ActionErrors errors = new ActionErrors();
    errors.add(ActionErrors.GLOBAL_ERROR, error);
    saveErrors(request, errors);
    return mapping.findForward("findInvoicePage");
}
```

You may recall that in a previous solution we created a convenience method on our Action base class to perform most of this work, so we can actually rewrite the previous code as follows:

```
catch (InvoiceDelegate.NotFoundException e) {
    postGlobalError(Constants.FIND_ERROR, invoiceNumber, request);
```

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 213


```
    return mapping.findForward("findInvoicePage");
}
```

Next we need to find the associated JSPs and replace any custom error presentation code, as in the following snippet:

```
<logic:present name="myMessage">
    <bean:write name="myMessage"/>
</logic:present>
```

with a Struts errors tag:

```
<html:errors/>
```

Note that if we were posting an ActionMessage rather than (or in addition to) an ActionError, we would need to add the following code to our JSP:

```
<html:messages id="messageId" message="true"/>
    <logic:present name="messageId">
        <h4><font color="red"><bean:write name="messageId"/></font></h4>
    </logic:present>
</html:messages>
```

The final step is to deploy and test the new implementation. When it works, you can continue on to fix any additional Actions and JSPs that are stuck in this pitfall.

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 214

Pitfall 4.4: Calculating Derived Values in JSPs

This pitfall examines the consequences of using scriptlets to perform calculations in JSPs. Developers are often tempted to take shortcuts, and one of the easiest to take is inserting a bit of scriptlet code in a JSP to calculate a value derived from other fields displayed in the same JSP.

The MVC model states that there are specific layers to which certain types of activities should be confined. One of the key benefits of using Struts is achieving a cleaner separation between these layers. Implementing calculations in JSP scriptlet code begins to erode this separation, though, coupling the view to the model, which leads to more difficult maintenance and harder-to-find bugs.

The main symptom of this pitfall is increased complexity of JSP code. This extra complexity makes the JSPs more difficult to maintain. Also, because there's no compile-time checking of the scriptlet code, the JSP code will be more vulnerable to runtime errors. Anyone who has had to track down an error in a JSP page knows how difficult it can be—especially compared to how straightforward tracking down exceptions in the regular Java code that makes up the Model and Controller classes usually is.

Usually this pitfall is seen as a scriplet that performs a calculation in a JSP. Here is a typical piece of scriptlet code that adds the employees' bonuses to their base salaries.

```
<%
    ...
    BigDecimal compensation = salary.add(bonus);
    ...
%>
```

Example

In this example, we will see some of the responsibilies of the earlier InvoiceDO (from Chapters 2 and 3) implemented in scriptlet code in a JSP. Suppose our InvoiceDO had a Balance field that represented the balance carried forward from the previous billing cycle. We could then present a Balance field on the Modify Invoice page as well as an Amount Due field. The amount due is derived by subtracting the prior balance from the invoice amount.

One way to accomplish this is to calculate the difference directly in the JSP. First we need to add some imports.

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 215


```
<%@ page import="com.aboutobjects.pitfalls.Constants,
                java.math.BigDecimal,
                com.aboutobjects.pitfalls.InvoiceForm,
                com.aboutobjects.pitfalls.Formatter" %>
```

Next we need a scriptlet to perform the calculation. Note that we need to use Formatter to unformat and format the values.

```
<%
Formatter formatter = Formatter.getFormatter(BigDecimal.class);
InvoiceForm form = (InvoiceForm) session.getAttribute("invoiceForm");
BigDecimal amount = (BigDecimal) formatter.unformat(form.getAmount());
BigDecimal balance = (BigDecimal) formatter.unformat(form.getBalance());
String amountDue;
if (amount == null)
    amountDue = "";
else if (balance == null)
    amountDue = formatter.format(amount);
else {
    BigDecimal difference = amount.add(balance.negate());
    amountDue = formatter.format(difference);
}
%>
```

Now we can add code to display the resulting value.

```
<tr>
    <td>Amount Due: </td>
    <td><%= amountDue%></td>
</tr>
```

If one or more additional pages need to display the same value, we will have to resort to copying and pasting this code to the new pages. A page containing numerous derived values could get quite complicated.

Pitfall 4.5: Performing Business Logic in JSPs is similar to this pitfall. Any time you have scriptlet code in your JSP, you should make sure that the code could not possibly belong somewhere else, and that it is confined to rendering logic. In addition to blurring the lines between the model, view, and controller, scriptlet code tends to increase maintenance costs and almost always leads to copying and pasting code from one JSP to another.

Solving Pitfall 4.4: Move Calculations to Value Object

The goal of this solution is to calculate the derived values in only one place, rather than copying a scriptlet implementation into several JSPs. The benefit to solving this pitfall is that the derived value calculation can be shared

not only between Web pages, but also with the business tier. It should also allow us to used typed values in performing the calculations, eliminating the need to perform extra type conversions.

The code to calculate the derived values can be in the value object or in the Session Bean, whichever is more convenient. The value object generally makes more sense if the values in question pertain only to the user interface, but if the derived value is used in further business logic or processing, the Session Bean probably makes more sense.

Step-by-Step

1. Begin with the simplest JSP that contains one or more calculations.
2. Add a getter method in the corresponding value object that performs the calculation and returns the derived value.
   a. As noted earlier, the right place to put the code could also be the Session Bean, depending on your implementation.
3. Add a property, including a getter and a setter, for a string version of the value in the corresponding ActionForm.
4. Modify the rendering code in the JSP to display the new form property.
   a. If you are using the dynamic mapping infrastructure from Chapter 2, add the property's name to the outbound keysToSkip array.
5. Test the new implementation.
6. Continue with the next JSP.

Example

The first step is to choose the JSP to start with; remember to keep it simple the first time you apply this solution to your existing code. We will start with the JSP that was described in the pitfall example. Here is the original scriptlet code.

```
<%
Formatter formatter = Formatter.getFormatter(BigDecimal.class);
InvoiceForm form = (InvoiceForm) session.getAttribute("invoiceForm");
BigDecimal amount = (BigDecimal) formatter.unformat(form.getAmount());
BigDecimal balance = (BigDecimal) formatter.unformat(form.getBalance());
String amountDue;
if (amount == null)
    amountDue = "";
else if (balance == null)
    amountDue = formatter.format(amount);
```

```
else {
    BigDecimal difference = amount.add(balance.negate());
    amountDue = formatter.format(difference);
}
%>
```

The next step is to add a getter method to InvoiceDO that performs the same calculation being done in the scriplet. Note that, in our current scenario, the code for our getter method will be quite a bit simpler because we're taking advantage of the typed values in InvoiceDO as well as the dynamic formatting infrastructure:

```
public BigDecimal getAmountDue() {
    if (amount == null)  return null;
    if (balance == null) return amount;
    return amount.add(balance.negate());
}
```

Now we need to add a property to the InvoiceForm to hold the formatted string version of the amountDue value. Here is the new code for InvoiceForm.

```
private String amountDue;
...
public String getAmountDue() { return amountDue; }
public void setAmountDue(String amountDue) {
    this.amountDue = amountDue;
}
```

Now we can modify our JSP to display the new property:

```
<tr>
    <td>Amount Due: </td>
    <td><bean:write name="invoiceForm" property="amountDue"/></td>
</tr>
```

Finally, before we start testing, we need to make sure that the mapping infrastructure doesn't try to transfer the amountDue value to the InvoiceDO because this is a derived value and therefore cannot be set. We need to add the corresponding key to the keysToSkip array in InvoiceForm, as follows:

```
protected ArrayList keysToSkip(int mode) {
    ArrayList keysToSkip = super.keysToSkip(mode);
    if (mode == TO_OBJECT) {
        ...
        keysToSkip.add("amountDue");
    }
    return keysToSkip;
}
```

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 218

Pitfall 4.5: Performing Business Logic in JSPs

This pitfall doesn't just blur the lines between the models, views, and controllers in a Struts application—it obliterates them. Coding business logic in the presentation tier is problematic because the implementation cannot be shared with business-tier objects that may require access to the same functionality. If the presentation tier and the business tier each have to have their own separate implementations of the same piece of logic, it becomes difficult to ensure consistent behavior. Placing the code in JSPs rather than Java classes only makes matters worse, making it impossible to share the logic even among presentation-tier classes. (Admittedly, you could use JSP include statements to source in a file containing scriptlet code in order to share business logic between JSPs, but other presentation-tier objects, such as Actions and ActionForms would still be unable to access the functionality.)

This pitfall is similar to Pitfall 4.4: Calculating Derived Values in JSPs, although the consequences are potentially even more serious, as it affects the critical business behavior of the application.

A couple of paths lead developers into this pitfall. One is simply lack of experience with applying the Model-View-Controller design pattern to developing Web applications. The other is the understandable tendency to take shortcuts under deadline pressure. Unfortunately, this shortcut will almost certainly cost the developer more time than it saves in the end.

The general form of this pitfall is a chunk of scriptlet code that implements business logic. In the following example, a bit of the business logic that properly belongs in the InvoiceDO has been implemented in a scriptlet in one of the JSPs.

```
<%
    InvoiceDO invoice = (InvoiceDO)
        request.getSession().getAttribute("invoice");
    GregorianCalendar dueDateCal = new GregorianCalendar();
    GregorianCalendar invoiceCal = new GregorianCalendar();
    invoiceCal.setTime(invoice.getBillingDate());
    dueDateCal.add(Calendar.DATE, -30);
    boolean thirtyDaysPastDue = invoiceCal.before(dueDateCal);
%>
```

Another block of code in the JSP renders the resulting value.

```
<% if (thirtyDaysPastDue) { %>
    <bean:message key="invoice.past.due"
                  arg0="<%= invoice.getInvoiceNumber().toString()%>"/>
<% } %>
```

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 219

How will this calculation of the past-due date be handled in other JSPs that need to present the same information? Unfortunately, the answer in this case is that the implementation will have to be duplicated in each JSP that requires the derived value.

Example

Below is a slightly more elaborate example involving a past-due invoice calculation. Calculated past-due dates may need to be presented in several different places in the user interface, but with the code presented here, we will have to copy and paste the scriptlet implementation in each JSP that displays these values.

```
<%
    InvoiceDO invoice = (InvoiceDO)
        request.getSession().getAttribute("invoice");
    GregorianCalendar dueDateCal = new GregorianCalendar();
    GregorianCalendar invoiceCal = new GregorianCalendar();
    invoiceCal.setTime(invoice.getBillingDate());
    int interval;
    switch (invoice.getPaymentTerm().intValue()) {
        case InvoiceDO.PAYMENT_TERM_30: interval = -30; break;
        case InvoiceDO.PAYMENT_TERM_60: interval = -60; break;
        default:                        interval = 0;
    }
    dueDateCal.add(Calendar.DATE, interval - 30);
    boolean thirtyDaysPastDue = invoiceCal.before(dueDateCal);
%>
```

Note that the first step in the scriptlet is to extract an InvoiceDO from the session. This in itself could be considered a pitfall, in that placing value objects directly in the session unnecessarily exposes them to being accidentally modified elsewhere. The more objects we have floating around in the session, the more difficult our application's code is to understand, as the reader of a given JSP generally has no clue about how the values got there and how or when they may be modified.

Of course, in real life, the logic could be quite a bit more complex, so the scriptlet could get considerably more convoluted. A great question to ask yourself as you check if any of your scriptlets have caused your JSPs to be trapped in this pitfall is, should this code be shared? If the answer is yes, then you are probably stuck in this pitfall and you should look at the solution. Your application is almost certainly going to need to apply the same logic elsewhere.

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 220

The next problem is that the container compiles this code, so you're at its mercy regarding compiler warnings and error messages, and container compiler error messages are notoriously hard to follow. If you're not precompiling your JSPs, compilation errors won't occur until you load the page in the Web container.

In addition, we still need code to implement the user interface logic associated with the above scriptlet. Let's suppose we simply wish to display a message regarding the lateness of the invoice. Here is the message as it is defined in the properties file:

```
invoice.past.due=Invoice {0} is {1} or more days past due
```

Here's the code we would need to render the message:

```
<% if (thirtyDaysPastDue) { %>
    <bean:message key="invoice.past.due"
                  arg0="<%= invoice.getInvoiceNumber().toString()%>"
                  arg1="thirty"/>
<% } %>
```

Note that the arg*N* parameters of the bean:message tag require arguments of type String, so that we can no longer use the formatting capabilities of java.util.MessageFormat to format the supplied value. In other words, a specification like this:

```
invoice.past.due=Invoice {0,number,#} is {1} or more days past due
```

would fail at runtime because the specification {0, number, #} requires an argument of type Number.

Therefore, any formatting would also have to be done in scriptlet code inside the JSP. Again, such formatting code could not be easily shared and thus would likely be copy/pasted around, with all its inherent evils. To avoid this pitfall, you can use helper classes to contain business logic so that the logic can be easily shared across the layers of your application.

Solving Pitfall 4.5: Move Business Logic to a Helper Class

Business logic can easily wind up being duplicated throughout the layers of your application. Obviously, that makes it difficult to maintain the code, and it usually frustrates attempts to reuse the implementations.

By creating a helper class to contain our business logic, we will be able to access the behavior easily anywhere in our application that it may be needed. This of course applies equally to business logic coded in Actions and ActionForms, though for the sake of this example we will focus on business logic coded in JSPs.

Step-by-Step

1. Start with the JSP with the simplest business logic scriptlet.
   a. Make sure to identify any copies of the scriptlet so that they can be deleted in favor of an expression to invoke the new implementation in the helper class.
2. Move the scriptlet code to a helper class.
3. Modify the implementation as necessary so that it can be shared as needed.
4. Modify rendering code in the JSP to take advantage of the new implementation.
5. Add code to the related Action to invoke the business logic.
6. Test the new implementation.
   a. After you have tested the new implementation, you can go back to the other JSPs that had the scriptlet copied into them and apply the same steps. It is a good idea not to move on to the next scriptlet until you have removed all the copies of the current scriptlet.
7. Continue with the next JSP that has business logic captured in scriptlets.

Example

In this example, we will apply the steps of this solution to the example we looked at in the pitfall and see how we can get the JSP and Action out of this pitfall. Making this choice completes the first step of the solution. Remember that you should start your solution with the simplest scriptlet to get the hang of applying this solution. Here is the scriptlet for reference.

```
<%
    InvoiceDO invoice = (InvoiceDO)
        request.getSession().getAttribute("invoice");
    GregorianCalendar dueDateCal = new GregorianCalendar();
    GregorianCalendar invoiceCal = new GregorianCalendar();
    invoiceCal.setTime(invoice.getBillingDate());
    int interval;
    switch (invoice.getPaymentTerm().intValue()) {
        case InvoiceDO.PAYMENT_TERM_30: interval = -30; break;
        case InvoiceDO.PAYMENT_TERM_60: interval = -60; break;
        default:                        interval = 0;
    }
    dueDateCal.add(Calendar.DATE, interval - 30);
    boolean thirtyDaysPastDue = invoiceCal.before(dueDateCal);
%>
```

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 222

The next step in our solution is to move the preceding implementation to a helper class. Let's call this class BusinessRules. The initial implementation is found in Listing 4.3.

```
package com.aboutobjects.pitfalls.rule;

import com.aboutobjects.pitfalls.InvoiceDO;
import java.util.Calendar;
import java.util.GregorianCalendar;

/**
 * The business rules for our application.
 */
public class BusinessRules {

    /**
     * Determines whether the provided invoice is past due by the
     * given number of days.
     * @param invoice The invoice to be checked
     * @param days The number of days before the invoice is considered
     *             past due
     * @return <code>true</code> if the invoice is past due;
     *         <code>false</code> otherwise
     */
    public static boolean isPastDue(InvoiceDO invoice, int days) {
        GregorianCalendar dueDateCal = new GregorianCalendar();
        GregorianCalendar invoiceCal = new GregorianCalendar();
        invoiceCal.setTime(invoice.getBillingDate());
        int interval;
        switch (invoice.getPaymentTerm().intValue()) {
            case InvoiceDO.PAYMENT_TERM_30: interval = -30; break;
            case InvoiceDO.PAYMENT_TERM_60: interval = -60; break;
            default:                        interval = 0;
        }
        dueDateCal.add(Calendar.DATE, interval - days);
        return invoiceCal.before(dueDateCal);
    }

    /**
     * Determines whether the provided invoice is thirty or more days
     * past due.
     * @param invoice The invoice to be checked.
     * @return <code>true</code> if the invoice is thirty or more days
     *         past due; <code>false</code> otherwise
     */
    public static boolean isThirtyDaysPastDue(InvoiceDO invoice) {
        return isPastDue(invoice, 30);
    }
}
```

Listing 4.3 BusinessRules.java.

Notice that we have made the basic implementation a bit more general by allowing the caller to pass the number of days past due. We have also added a convenience method isThirtyDaysPastDue() to invoke the generic method with a well-known value. This code can now easily be made available to business-tier classes as well as to all levels of the presentation tier.

Accordingly, we can now invoke these methods from our Actions. In this case, we simply need to add the following code to our FindInvoiceAction:

```
if (BusinessRules.isThirtyDaysPastDue(invoice)) {
    postGlobalMessage(Constants.INVOICE_PAST_DUE, invoiceNumber,
                      "thirty", request);
}
```

The next step is to update the JSP so that it takes advantage of the new implementation. First, we will need to modify the string specification in our properties file to prevent java.util.MessageFormat from inserting comma separators in the invoice number during formatting:

```
invoice.past.due=Invoice {0,number,#} is {1} or more days past due
```

Now we can clean up the rendering logic in the JSP by replacing this code:

```
<% if (thirtyDaysPastDue) { %>
    <bean:message key="<%= Constants.INVOICE_PAST_DUE%>"
                  arg0="<%= invoice.getInvoiceNumber().toString()%>"
                  arg1="thirty"/>
<% } %>
```

with this obviously much more generic code:

```
<html:messages id="messageId" message="true"/>
    <logic:present name="messageId">
        bean:write name="messageId"/>
    </logic:present>
</html:messages>
```

This makes our JSP more adaptable to future changes. Since the invocation of the business logic now takes place in the Action, we can now delete the business logic scriptlet from our JSP as well.

The next step is to deploy and test the new code to make sure that everything works as it should. Once the new implementation is working correctly we can search for any other JSPs that had copies of the scriptlet and clean them up as well. The final step is to repeat this whole process for each JSP that is trapped in this pitfall.

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 224

Pitfall 4.6: Hard-Coded Options in HTML Select Lists

This pitfall deals with problems developers often encounter with HTML select lists in Struts. One of the trickier issues developers face when coding their JSPs is how to populate the value lists displayed in the drop-down menus rendered by the HTML select element. Struts provides an html:options tag that allows developers to specify a bean containing a collection that provides the labels to display in the UI, and another collection for their associated values, if different. Unfortunately, many developers find the Struts html:options tag confusing when they initially encounter it, and they simply find it expedient to code the values that they need directly in their JSP using the html:option tag instead. This leads to a lot more code than is required, thereby increasing maintenance costs. The cause of this problem is twofold: First, populating HTML select lists is one of the less straightforward mappings developers need to work out, and second, the associated Struts tags are correspondingly less intuitive. In addition, documentation on this feature of Struts has historically been sparse.

Example

Here's an example of a JSP that is stuck in this pitfall. The code uses hard-coded values to populate the labels and values of an html:select tag.

```
<html:form  action="/saveInvoice" method="post">
    ...
    <html:select property="paymentTerm">
         <html:option value="0">None</html:option>
         <html:option value="1">Net 30 Days</html:option>
         <html:option value="2">Net 60 Days</html:option>
    </html:select>
    ...
</html:form>
```

And here is the HTML it generates:

```
<form name="invoiceForm"
      method="post"
      action="/pitfalls/saveInvoice.do">
    ...
    <select name="paymentTerm">
        <option value="0">None</option>
        <option value="1" selected="selected">Net 30 Days</option>
        <option value="2">Net 60 Days</option>
    </select>
</form>
```

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 225

The html:select tag binds the value of its property element, in this case paymentTerm, to the value of the underlying bean. In our example, the bean will be the ActionForm bound to the path /saveInvoice (the html:form's action) in struts-config.xml. The tag will automatically preselect a matching value (if one is found) from the options list during rendering, and the user-selected value will automatically be bound to the ActionForm's paymentTerm property when the form is submitted.

The problem with this approach is that we will be forced to copy/paste this code to other JSPs that need to render the same select list. Any changes would then have to be applied by hand, opening the door to inconsistency and bugs. For example, suppose a developer accidentally switched two of the value's settings. The user interface would display the user's selection correctly, but the underlying bean value would be incorrect. This type of bug is often hard to diagnose. Clearly, it would be best if we could maintain a single instance of the list and share it among various JSPs.

Another potential problem is that some applications may require the flexibility of having the values lists determined dynamically. For example they might be fetched from a database table, which would allow new values to be added at runtime. Obviously if the values are hard-coded in a JSP, the application would lack this kind of flexibility.

You can avoid this pitfall by using a helper class to provide the label/value mappings for select lists, along with the Struts html:options tag to populate html select lists. See the solution for details on how to do this.

Solving Pitfall 4.6: Move Options Values to a Helper Class

Moving the label/value mappings from JSP options lists to one or more helper classes will allow us to share the implementation across multiple JSPs and conceivably other code that may be able to use them. It also provides a central location for viewing the mappings, which makes it easier to verify that they are correct and to modify them when necessary.

Step-by-Step

1. Start with any JSP containing an options list with hard-coded values.
2. Add constants to the associated value object for each of the allowable bean values.
3. Create a method in a helper class that returns a Collection corresponding to the label/value pairs in the options list.
   a. You might have to create a new helper class if there is no logical place to put the method.

4. Add a method to your base ActionForm class to provide access to an instance of the helper class.
5. Replace the hard-coded options with an html:options tag.
6. Test the implementation.
7. Repeat for additional options lists until all have been refactored.

Example

The first step of the solution is to choose the JSP to start with. We will start with the example from the pitfall. Here is the options list for reference.

```
...
<html:select property="paymentTerm">
     <html:option value="0">None</html:option>
     <html:option value="1">Net 30 Days</html:option>
     <html:option value="2">Net 60 Days</html:option>
</html:select>
...
```

First, we'll define constants in InvoiceDO or in Constants to represent the allowable bean values. (Note: it's generally best to place constant definitions in the class with which they are most closely associated whenever possible and to avoid dumping everything into the Constants class.)

```
public final static int PAYMENT_TERM_NONE = 0;
public final static int PAYMENT_TERM_30 = 1;
public final static int PAYMENT_TERM_60 = 2;
```

Next, we'll create a helper class to provide the label/value mappings. Struts offers us some assistance in the form of the LabelValueBean class, which provides predefined label and value properties that we can use to contain the values, as seen in Listing 4.4.

```
package com.aboutobjects.pitfalls;

import java.util.ArrayList;
import java.util.List;
import java.io.Serializable;
import org.apache.struts.util.LabelValueBean;

/**
 * The label/value mappings used in options lists.
 */
```

Listing 4.4 Options.java.

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 227


```
public class Options {
    private static Options uniqueInstance = new Options();

    // Don't allow direct instantiation.
    private Options() { }

    public static Options getInstance() { return uniqueInstance; }

    public List getPaymentTerms() {
        ArrayList paymentTerms = new ArrayList();
        String none = Integer.toString(InvoiceDO.PAYMENT_TERM_NONE);
        String net30 = Integer.toString(InvoiceDO.PAYMENT_TERM_30);
        String net60 = Integer.toString(InvoiceDO.PAYMENT_TERM_60);
        paymentTerms.add(new LabelValueBean("None", none));
        paymentTerms.add(new LabelValueBean("Net 30 Days", net30));
        paymentTerms.add(new LabelValueBean("Net 60 Days", net60));
        return paymentTerms;
    }
    ...
}
```

Listing 4.4 *(continued)*

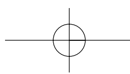
Now we can make the Options instance visible to the JSPs by adding an accessor method to the ActionForm base class:

```
    /**
     * Returns the unique instance of the Options class.
     * @return the Options instance
     */
    public Options getOptions() { return Options.getInstance(); }
```

The next step is to replace the hard-coded options list in the JSP with a bit of slightly tricky Struts tag code, as seen here:

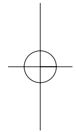
```
<html:select property="paymentTerm">
    <bean:define id="values"
                 name="invoiceForm"
                 property="options.paymentTerms"
                 type="java.util.ArrayList"/>
    <html:options collection="values"
                  property="value"
                  labelProperty="label"/>
</html:select>
```

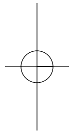
e 449156 Ch04.qxd 6/16/03 8:49 AM Page 228

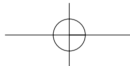


The bean:define tag puts the List returned by getPaymentTerms() in scope so that the html:options tag can access it, using the arbitrary identifier values. The html:options tag will then iterate the collection, rendering an html option tag for each entry.

The next step is to deploy and test the cleaned-up code. After testing to make sure that this fix has been completed successfully, you can proceed to other hard-coded option lists in your other JSPs.







e 449156 Ch04.qxd 6/16/03 8:49 AM Page 229

Pitfall 4.7: Not Checking for Duplicate Form Submissions

A side effect of the stateless nature of the HTTP protocol is that the interface presented in the browser can easily get out of sync with the state of the model on the server. There is a common pitfall involving this lack of synchronization that must be dealt with in nearly every application. This occurs when a user submits values using a form on a given page and then, at some later point, backtracks to the cached page, edits the now stale values, and resubmits the form. This has the potential to corrupt the underlying data store by creating duplicate records, overwriting current data with stale values, and so on. In the best case, it creates a bad transaction that is rejected by the data store.

This pitfall potentially affects nearly every JSP containing a form tag. In most projects, the problems show up during system test. Fortunately for developers, system testers are generally wise enough to include scenarios in their test scripts that expose this type of bug. Developers, on the other hand, rarely test for these conditions.

Example

The code that follows represents a typical naive form implementation that is trapped in this pitfall. If we don't do anything special to prevent it, the user can use the HTML generated by this JSP to submit the same form multiple times.

```
<html:form  action="/saveInvoice" method="post">
   <table>
       <tr>
           <td><bean:message
                   key="<%=Constants.INVOICE_NUMBER_LABEL_KEY%>"/><td>
           <td><html:text property="invoiceNumber"/></td>
       </tr>
       <tr>
           <td><bean:message
                   key="<%=Constants.BILLING_DATE_LABEL_KEY%>"/><td>
           <td><html:text property="billingDate"/></td>
       </tr>
       <tr>
           <td><bean:message
                   key="<%=Constants.AMOUNT_KEY%>"/><td>
           <td><html:text property="amount"/></td>
       </tr>
```

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 230


```
            <td colspan=3 align="right"><html:submit/></td>
        </tr>
    </table>
</html:form>
```

One of the common mechanisms to control this pitfall, and the one that will be explored in the accompanying solution, is to have forms submit tokens that the application can check to ensure that the forms are not submitted more than once.

Solving Pitfall 4.7: Add Tokens to Generated JSP

To solve this pitfall, we will make use of one of the more obscure features of Struts. The token management facility is provided by the Action and FormTag classes, comprising a small group of Action methods that provide for token generation and checking, and largely undocumented code in the FormTag to render the generated tokens automatically. Taken together, these two features make it quite easy to check for duplicate submissions.

Our goal here is to ensure that forms are submitted only when they contain fresh data. In essence, we're trying to exert some control over the flow of the application. To do that, we can call on Struts to place tokens in our forms as necessary, so that we can check the submitted token against a copy cached on the server.

Step-by-Step

1. Choose a form.
2. Identify the Action that navigates to the form.
   a. Add code to generate a token.
3. Identify the Action that handles submission of the form.
   a. Add code to check the generated token and to generate a new one if the Action is successful.
4. Test the implementation.
5. Continue with other Actions until all form-submission scenarios have been covered.

Example

The first step in the solution is to choose a form. We will begin this example by choosing the FindInvoice form. The next step is to identify the

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 231

Action that navigates to the form and add code to generate the token. The Action that navigates to the form is the FindInvoiceAction. This is the code for that action with the token generation added.

```
public class FindInvoiceAction extends BaseAction
{
    public ActionForward execute(ActionMapping mapping,
                                 ActionForm form,
                                 HttpServletRequest request,
                                 HttpServletResponse response)
        throws IOException, ServletException
    {
        ...
        saveToken(request); // This is all we need to add

        return mapping.findForward("saveInvoicePage");
    }
}
```

The FormTag class takes care of inserting the token for you if a value for it has been set, as we did above with saveToken in the FindInvoiceAction.

The next step is to identify the action that handles submission of the form and add code to that form that will make sure that the tokens match and then generate a new token if the form submission was successful. Listing 4.5 is the code for the SaveInvoiceAction that has been modified to check the token.

```
public class SaveInvoiceAction extends BaseAction
{
    public ActionForward execute(ActionMapping mapping,
                                 ActionForm form,
                                 HttpServletRequest request,
                                 HttpServletResponse response)
        throws Exception {

        // Check the token. If it doesn't match the token currently in
        // the session, it's stale.
        if (!isTokenValid(request)) {
            postGlobalError(Constants.DUPLICATE_FORM_ERROR, request);
            return mapping.findForward("saveInvoicePage");
        }

        ...

        // if we got this far, everything worked okay, so generate
```

Listing 4.5 Checking the token in SaveInvoiceAction. *(continues)*

e 449156 Ch04.qxd 6/16/03 8:49 AM Page 232


```
        // a new token.
        saveToken(request);

        return mapping.findForward("confirmInvoicePage");
    }
}
```

Listing 4.5 *(continued)*

The final step is to deploy and test the code to make sure it functions properly. Once you have a working implementation, you can add token generation and checking wherever else it is needed. Extra care should be taken in testing the solution to ensure that forms cannot be resubmitted under any odd combinations of circumstances.