Pro Jakarta Struts, Second Edition

JOHN CARNELL AND ROB HARROP

Apress™

Pro Jakarta Struts, Second Edition

ISBN (pbk): 1-59059-228-X

Printed and bound in the United States of America 12345678910

Technical Reviewer: Jan Machacek

Editorial Board: Steve Anglin, Dan Appleman, Gary Cornell, James Cox, Tony Davis, John Franklin, Chris Mills, Steven Rycroft, Dominic Shakeshaft, Julian Skinner, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Assistant Publisher: Grace Wong

Project Manager: Beth Christmas

Copy Editor: Ami Knox

Production Manager: Kari Brooks

Production Editor: Laura Cheu

Proofreader: Thistle Hill Publishing Services, LLC

Compositor: Kinetic Publishing Services, LLC

Indexer: Michael Brinkman

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 3

Form Presentation and Validation with Struts

In the previous chapter, all of our Struts examples were built around very simple screens that were populated with data retrieved from the JavaEdge application. However, most web applications require a high degree of interaction, with end users often submitting the data via HTML forms.

This chapter is going to look at how to simplify the construction of HTML forms and form-handling code using the Struts development framework. We are going to discuss, from both a conceptual and implementation point of view, how the Struts framework can provide a configurable and consistent mechanism for building web forms. This chapter is going to cover the following topics:

- Validating HTML form data using the `ActionForm` class. Some of the topics that will be covered include
  - How the `validate()` method of the `ActionForm` class is used to validate data against the user
  - Error handling when a validation rule is violated
- Prepopulating an HTML form with data
- Configuring Struts for processing HTML form data
- Simplifying the development of HTML form pages using the Struts HTML tag libraries
- Using `Map`-backed `ActionForms` to build flexibility into your application
- Best practices associated with using the Struts `ActionForm` class

Problems with Form Validation

Most web development teams do not have a consistent strategy for collecting data from the end user, validating it, and returning any error messages that need

to be displayed. They use a hodgepodge of different means of collecting and processing the user's data. Two commonly used validation mechanisms include embedding JavaScript in the HTML or JSP page rendering the form, and/or mixing the validation logic for the screen with the business logic in the business tier of the application.

This inconsistency in how data is collected and validated often results in the following:

- Customers, whether they are internal (such as employees) or external (such as purchasers), having a disjointed experience while using the web-based applications of the organization. Each application requires the customer to have a different set of skills and an understanding of how the application works and how to respond to errors. In larger applications, this inconsistency can exist even between different pages in the same application.

- Validation logic is strewn through the different layers of the application. This increases the amount of time required to perform application maintenance. The maintenance developer, who is rarely the same as the code developer, often has to hunt for the location of the validation logic and know multiple development languages (JavaScript for validation rules enforced in the browser, Java for validation logic in the middle tier, and a stored procedure language for validation logic in the database).

- Validation logic is used differently across different browsers. JavaScript, though "standardized," is implemented differently across browsers. Developers of a web browser take great liberties in the way in which they implement the JavaScript European Computer Manufacturers Association (ECMA) standard. Often, they provide their own browser-specific extensions, which make cross-browser viewing (and portability) of the application difficult.

- The application code is difficult to reuse. With validation logic strewn throughout the tiers of an application, it is difficult to pick up that validation code and reuse it in another application. The developer has to take care of the dependencies being present before reusing the validation code, because there is no clean separation between the validation and business logic.

All the problems just identified are the symptoms of the Validation Confusion antipattern. Recollecting the discussion in Chapter 1, the Validation Confusion antipattern occurs due to one of the following reasons:

- No clear distinction between the validation logic of the form data and the business logic that processes the user's request
- Lack of a pluggable interface, which allows the developer to easily modify the validation logic for a particular screen
- No standardized mechanism for identifying the validation violations and notifying the end user of them

Fortunately, the Struts framework provides a rich set of software services for building and managing the form data. These services allow a developer to handle the form validation in a consistent fashion. Much of the logic, normally associated with capturing and presenting the errors, becomes the responsibility of the Struts framework and not of the application developer.

Using Struts for Form Validation

To build an HTML form, in Struts, you need to have the following pieces in place:

- A Struts `ActionForm` class, which is used to hold the data collected from the end user and perform any validations on that data. This class provides a simple-to-use "wrapper" that eliminates the need for developers to pull the submitted data out of the `HttpServletRequest` object, associated with the end user's request.
- A Struts `Action` class to carry out the user's request. In the process of carrying out the user's request, any business rules that should be enforced will be executed, and any database insert, updates, or deletes will be performed.
- A JSP page that uses the Struts HTML tag libraries to render the form elements that are going to appear on the page.

Tying all of these pieces together is the struts-config.xml file. This file will have entries in it for defining the Struts `ActionForm` classes used in the application, which `ActionForm` classes are going to be used with which action, and whether an `ActionForm` class is going enforce the validation against the submitted data. Each Struts action processing the form data must have its corresponding `<action>` tag modified, to indicate which `ActionForm` class will be used by the action.

Let's discuss what happens when the user submits the data in an HTML form. Figure 3-1 shows what happens when the user submits the form data to a Struts-based application.

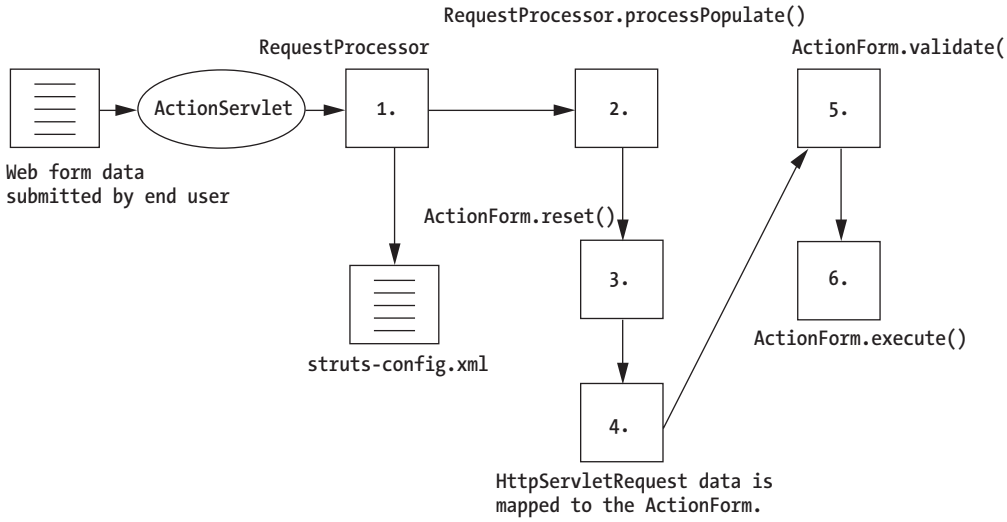

Figure 3-1. The flow of a user request through Struts

1. The `ActionServlet` will examine the incoming URL mapping or extension to determine what action the user submitting the data wants to take. The `ActionServlet` will create an instance of a `org.apache.struts.action.RequestProcessor` class and hand over responsibility for processing the user's request to it.

   The `RequestProcessor` will look at the data for the action requested in the struts-config.xml file. It will then determine whether or not an `ActionForm` has been defined for the action and if so, what scope the `ActionForm` resides in.

   Once a scope has been determined for an `ActionForm`, the `RequestProcessor` will check to see if the `ActionForm` already exists in that scope. If the desired `ActionForm` class does exist in the defined scope, the `RequestProcessor` will retrieve it and pass it to the `RequestProcessor.processPopulate()` method.

   If the desired `ActionForm` does not exist in scope, the `RequestProcessor` will create an instance of it, put it into the scope defined inside of the `<action>` tag in the struts-config.xml file, and then pass the created `ActionForm` instance to the `processPopulate()` method.

2. The `processPopulate()` method is responsible for mapping the form data passed into it via the `HttpServletRequest` object to the Struts `ActionForm` defined for the action being processed. It does this by first calling the `reset()` method on the `ActionForm` and then populating the `ActionForm` with data from the `HttpServletRequest` object.

An `ActionForm` simplifies the form processing, but it is not required to access the form data submitted by the end user. An `Action` class can still access the submitted form data by calling the `getParameter()` method on the `request` object passed into its `execute()` method. However, overreliance on the `getParameter()` method can bypass much of the validation and error-handling support in Struts.

3. Before the data submitted can be validated, the `RequestProcessor` will call the `ActionForm`'s `reset()` method. The `reset()` method is a method that can be overridden by Struts developers to "initialize" or "override" individual properties on an `ActionForm`.

   This method is most commonly used to properly handle a web form's radio checkbox fields when the form has been submitted multiple times. The `reset()` method will be covered in greater detail later on in the chapter.

4. Once the `reset()` method has been called, the `RequestProcessor` will populate the `ActionForm` with data from the request by using the `org.apache.struts.util.RequestUtil`'s `populate()` method.

5. Once the form data has been mapped from the `HttpServletRequest` object to the `ActionForm`, the submitted data will be validated. The `RequestProcessor` will validate the form data by calling the `ActionForm`'s `validate()` method.

   If a validation error occurs, the `RequestProcessor` will inform the `ActionServlet` to redirect users back to the screen where data was submitted. Users will then have to correct the validation violations before they can continue. We will be covering how Struts is notified of a validation error in the section called "Validating the Form Data."

6. If the data contained in the `ActionForm` successfully passes all validation rules defined in the `validate()` method, the `RequestProcessor` will invoke the `execute()` method on the `Action` class. The `execute()` method contains the business logic needed to carry out the end user's request.

Remember that the Java class, which carries out the end user's request, is defined via the `type` attribute in the `<action>` element. We suggest you to refer to Chapter 2 to understand how to configure a Struts action before continuing.

Implementing Form Validation with Struts

Let's begin the discussion of form handling by Struts by looking at how an HTML form is processed by Struts when a user submits it. We are going to use the Post a Story page from the JavaEdge application as our example.

This page can be accessed by either clicking the Post a Story link in the menu bar at the top of every JavaEdge page or pointing your browser to `http://localhost:8080/javaedge/execute/postStorySetup`. The Post a Story page is used by a JavaEdge user to submit a story, which the other users visiting this page can read.

If you have successfully reached this page, you will see the screen in Figure 3-2.

Figure 3-2. The Post a Story page

Let's begin by looking at how to set up the `struts-config.xml` class to use `ActionForm` objects.

The struts-config.xml File

To use an `ActionForm` class to validate the data collected from a user form, the struts-config.xml file for the application must be modified. These modifications include

- Adding a `<form-beans>` tag, which will define each of the `ActionForm` classes used in the application

- Modifying the `<action>` tag processing the user's request to indicate that before the user's request is processed, it must be validated by an `ActionForm` class

The `<form-beans>` tag holds one or more `<form-bean>` tags within it. This tag appears at the top of the struts-config.xml file. Each `<form-bean>` tag corresponds to only one `ActionForm` class in the application. For the JavaEdge application, the `<form-beans>` tag looks as shown here:

```
<form-beans>
  <form-bean name="postStoryForm"
             type="com.apress.javaedge.struts.poststory.PostStoryForm"/>
  ...
  <form-bean> //More form-bean definitions.
</form-beans>
```

The `<form-bean>` element has two attributes:

- `name`: A unique name for the form bean being defined. Later on, this name will be used to associate this form bean with an `<action>` element. This attribute is a required field.

- `type`: The fully qualified class name of the `ActionForm` class that the form bean represents. This attribute is also a required field.

The `<form-bean>` actually has a third optional attribute called `className`. This attribute is used to specify what configuration bean to use for the defined form bean. If the `className` attribute is omitted, Struts will default to the `org.apache.struts.config.FormBeanConfig` class.

Once a `<form-bean>` has been defined, you can use it in an `<action>` element to perform validation of the form data. To add the validation to a `<form-bean>`, you must supply the following four additional attributes in an `<action>` element:

Attribute Name	Attribute Description
name	Maps to the name of the <form-bean> that will be used to process the user's data.
scope	Defines whether or not the ActionForm class will be created in the user's request or session context. The scope attribute can be used only when the name attribute is defined in the <action> tag. If the name attribute is present, the scope attribute is an optional tag. The default value for the scope attribute is request.
validate	A Boolean attribute that indicates whether or not the submitted form data will be validated. If it's true, the validate() method in the ActionForm class and the execute() method in the Action class will be invoked. If it's false, then the validate() method will not be invoked, but the execute() method in the Action class defined in the tag will be executed. The validate attribute is used only when the name attribute has been defined in the tag. The default value for the validate attribute is true.
input	Used to define where the user should be redirected, if a validation error occurs. Usually, the user is redirected back to the JSP page where the data was submitted. It is not required if the name attribute is not present.

The /postStory action processing the data entered by the user in the postStory.jsp page is shown here:

```
<action path="/postStory"
         input="/WEB-INF/jsp/postStory.jsp"
         name="postStoryForm"
         scope="request"
         validate="true"
         type="com.apress.javaedge.struts.poststory.PostStory">
  <forward name="poststory.success" path="/execute/homePageSetup"/>
</action>
```

Struts ActionForm Class

The Struts ActionForm class is used to hold the entire form data submitted by the end user. It is a helper class that is used by the ActionServlet to hold the form data that it has pulled from the end user's request object. The application developer can then use the ActionForm to access the form through get() and set() method calls.

The `ActionForm` class not only provides a convenient wrapper for the request data but also validates the data submitted by the user. However, an `Action` class is not required to have an `ActionForm` class. An `Action` class can still access the form data submitted by the end user by calling the `getParameter()` method in the `request` object passed into its `execute()` method.

To build an `ActionForm` class, the developer needs to extend the base Struts `ActionForm` class and override two methods in it, `reset()` and `validate()`.

Just to review, the `reset()` method is overridden by the developer when an `ActionForm` class for an action is to be stored in the user's session context. The `reset()` method clears the individual attributes in the `ActionForm` class to ensure that the `ActionForm` class is properly initialized before it is populated with the user's form data. The `validate()` method is overridden by the developer. This method will contain all of the validation logic used in validating the data entered by the end user.

In addition, the application developer needs to define all the form elements that are going to be collected by the `ActionForm` class and stored as private properties in the class. For each defined property, there must be corresponding `get()` and `set()` methods that follow the standard JavaBean naming conventions.

NOTE *You must implement a* `get()` *and* `set()` *method for each form element captured off the web form. These* `get()` *and* `set()` *methods should follow the standard JavaBean naming conventions. The first letter of the word after* `get()`/`set()` *should be capitalized along with the first letter of each word in the method thereafter. All other letters in the method name should be lowercase. The Struts framework uses Java reflection to read the data from and write data to the* `ActionForm` *class. An exception will be raised if the* `get()` *or* `set()` *method is not present for a piece of data submitted.*

For the Post a Story page, you are going to write a Struts `ActionForm` class called `PostStoryForm.java`. This class will hold the story title, the story intro, and the body of the story. In addition, it will contain the validation code for the data being submitted by the user.

The class diagram shown in Figure 3-3 illustrates the class relationships, methods, and attributes for the Struts `ActionForm` class and the `PostStoryForm` class.

Figure 3-3. PostStoryForm's relationship to the ActionForm class

It is very easy to fall into the mind-set that there must be one `ActionForm` class for each HTML form from which the data is collected. In small-to-medium size applications, there is nothing wrong in using a single `ActionForm` placed in the user's session. All the forms in the application will use this `ActionForm` to hold the data collected from the user.

This simplifies the collection of the data because your application has only one `ActionForm` instance that you have to work with. By using a single `ActionForm` class and placing it in the user's session, you can very easily implement a wizard-based application that will remember each piece of user information entered. As the user steps back and forth through the wizard, the data can easily be retrieved from the single `ActionForm` class.

The problem with using a single `ActionForm` class in the user's session is that the application will not scale as well. Remember, the objects placed in the user's session have a held reference until the session times out and the objects are garbage collected.

Do not place `ActionForm` objects in the session merely as a convenience. The other problem with this method occurs if the users are carrying out a long-lived transaction. If the users lose their connection or close their browser, any of the data entered till then will be lost.

To ensure that as much of the user's data is captured and persisted as possible, break the application into smaller transactions. Use an `ActionForm` class for each application screen and persist the data in the `ActionForm` class as soon as

the users submit their data. Place the `ActionForm` class into the request so that server resources are not unnecessarily used.

The code for the `PostStoryForm` class is shown next. However, the `reset()` and `validate()` methods for this class are not displayed. They will be discussed in the sections "Using the reset() Method" and "Validating the Form Data," respectively.

```
package com.apress.javaedge.struts.poststory;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionServlet;
import org.apache.struts.util.MessageResources;
import com.apress.javaedge.common.VulgarityFilter;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class PostStoryForm extends ActionForm {

  String storyTitle = "";    //defined as empty string
  String storyIntro = "";
  String storyBody  = "";

 /**
  *  Validates all data posted from the Post Story page.
  */
  public ActionErrors validate(ActionMapping mapping,
                               HttpServletRequest request) {
    ...
  }

 /**
  *  Used to clear out the values stored in a PostStoryForm classes
  *     attributes.
  */
  public void reset(ActionMapping mapping, HttpServletRequest request) {
    ...
  }
```

```
  public java.lang.String getStoryTitle() {
    return storyTitle;
  }

  public void setStoryTitle(java.lang.String storyTitle) {
    this.storyTitle = storyTitle;
  }

  public java.lang.String getStoryIntro() {
    return storyIntro;
  }

  public void setStoryIntro(java.lang.String storyIntro) {
    this.storyIntro = storyIntro;
  }

  public java.lang.String getStoryBody() {
    return storyBody;
  }

  public void setStoryBody(java.lang.String storyBody) {
    this.storyBody = storyBody;
  }
}
```

Using the reset() Method

The `reset()` method is used to ensure that an `ActionForm` class is always put in a "clean" state before the `ActionServlet` populates it with the form data submitted in the user's request. In the struts-config.xml file, the developer can choose to place an `ActionForm` for a specific Struts action in either the user's session or request.

The `reset()` method was originally implemented to allow developers to deal with one of the more annoying HTML form controls: checkboxes. When a form is submitted with unchecked checkboxes, no data values are submitted for the checkbox control in the HTTP request.

Thus, if an `ActionForm` is sitting in the user's session and the user changes a checkbox value for the `ActionForm` from `true` to `false`, the `ActionForm` will not get updated because the value for the checkbox will not be submitted. Remember, the HTML `<input>` tag does not send a value of `false` on an unchecked checkbox.

The `reset()` method can be used to initialize a form bean property to a predetermined value. In the case of a form bean property that represents a checkbox, the `reset()` method can be used to set the property value always to `false`.

Since the `reset()` method is called before the form is populated with data from the `HttpServletRequest` object, it can be used to ensure that a checkbox is set to `false`. Then if the user has checked a checkbox, the `false` value set in the `reset()` method can be overridden with the value submitted by the end user.

The Struts development team typically recommends the `reset()` method only be used for the preceding purpose. However, as you will see in the next section, the `reset()` method can be useful for prepopulating a JSP page with data.

A Word on the reset() Method

Among Struts developers the use of the `reset()` method to prepopulate form data can be the cause of rigorous debate. The Struts JavaDoc advises to not use the `reset()` method. The main reason the Struts development team gives is that the `reset()` method maybe deprecated at some point in the future (even though this has yet to be even mentioned anywhere).

In the next several sections, we will be demonstrating how to prepopulate a web page by using the `reset()` method and a "setup" action. We give our reason for using both methods and have seen both methods work rather successfully in production-level systems. That being said, please do not deluge our mailboxes with angry e-mails.

Implementing the `reset()` method for the `PostStoryForm` will set all its properties to an empty string. The `reset()` method for the `PostStoryForm` class is shown here:

```
public void reset(ActionMapping mapping,
                  HttpServletRequest request) {
  storyTitle = "";
  storyIntro = "";
  storyBody  = "";
}
```

Prepopulating an ActionForm with Data

So far, we have talked about using the `reset()` method to ensure that the contents of an `ActionForm` class are cleared before the `ActionServlet` places data in it from the user request. However, an `ActionForm` class can also be used to prepopulate an

HTML form with data. The data populating the form might be text information retrieved from a properties file or a database.

To prepopulate an HTML form with data, you need to have the following Struts elements in place:

- A Struts Setup action that will be called before a user is redirected to a JSP page, displaying an HTML form prepopulated with the data. The concept of Setup actions is discussed in Chapter 2.

- An `ActionForm` class whose `reset()` method will prepopulate the form fields with data retrieved from the ApplicationResources.properties file. The ApplicationResources.properties file is discussed in Chapter 2.

- A JSP page that uses the Struts HTML tag libraries to retrieve the data from the `ActionForm` class.

For example, you can prepopulate the HTML form for the Post a Story page with some simple instructions on what data is supposed to go in each field. For this example, you are going to use the following files:

- PostStoryForm.java

- PostStorySetupAction.java

- postStoryContent.jsp

We are only going to show you the `PostStoryForm` and the `PostStorySetupAction` Java classes. The postStoryContent.jsp file will use the Struts HTML tag library to read the values out of the `PostStoryForm` object stored in the request and display them in each field. The postStoryContent.jsp file and Struts HTML tag library are discussed in the section "The Struts HTML Tag Library."

PostStoryForm.java

Writing the `reset()` method for a `PostStoryForm` to prepopulate the `ActionForm` with the instructions for each field in the form is a straightforward task:

```
public void reset(ActionMapping mapping,
                  HttpServletRequest request) {
  ActionServlet servlet = this.getServlet();
  MessageResources messageResources = servlet.getResources();
```

```
    storyTitle =
      messageResources.getMessage("javaedge.poststory.title.instructions");
    storyIntro =
      messageResources.getMessage("javaedge.poststory.intro.instructions");
    storyBody =
      messageResources.getMessage("javaedge.poststory.body.instructions");
}
```

The `reset()` method just shown reads values from the ApplicationResources.properties file and uses them to populate the properties of the `PostStoryForm` object.

NOTE *In the preceding* `reset()` *method, the error messages being looked up by the call to* `getMessage()` *have a string literal being passed in as a parameter. This string literal is the name of the message being looked up from the resource bundle used for the JavaEdge application (that is, the ApplicationResources.properties file).*

This was done for clarity in reading the code. A more maintainable solution would be to replace the individual string literals with corresponding static final constant values.

The Struts development framework provides an easy-to-use wrapper class, called `MessageResources`, for directly accessing the data in the ApplicationResources.properties file.

NOTE *We use the name ApplicationResources.properties for the name of the message resource bundle used in the JavaEdge application because this is traditionally what this file has been called in the Struts application. However, the name of the file used as the message resource bundle can be set in the parameter attribute of the* `<message-resources>` *tag contained within the struts-config.xml file. For a review of the* `<message-resources>` *tag, please review Chapter 2.*

To retrieve an instance of the `MessageResources` class, you first need to get a reference to the `ActionServlet` that is currently processing the `ActionForm` object. Fortunately, the `ActionForm` class provides a `getServlet()` method that will retrieve an instance of the `ActionServlet`:

```
ActionServlet servlet = this.getServlet();
```

Once an instance of the `ActionServlet` is retrieved, a call to its `getResources()` method will retrieve a `MessageResources` object that wraps all the values stored in the ApplicationResources.properties file:

```
MessageResources messageResources = servlet.getResources();
```

After getting an instance of a `MessageResources` object, you can pass the message key of the item that you want to retrieve to `getMessage()`. The `getMessage()` method will retrieve the desired value.

```
messageResources.getMessage("javaedge.poststory.title.instructions");
```

If the key passed to the `getMessage()` method cannot be found, a value of `null` will be returned. The following are the name-value pairs from the ApplicationResources.properties file used to prepopulate the `PostStoryForm`:

```
javaedge.poststory.title.instructions=Enter a title here.
javaedge.poststory.intro.instructions=
Enter the story introduction here. Please be concise.
javaedge.poststory.body.instructions=Enter the full story here. Please be nice.
```

The `PostStoryForm.reset()` method is a very simple example of how to prepopulate a form with the data contained in an `ActionForm` class. In reality, many applications retrieve their data from an underlying relational database rather than from a properties file. How the `reset()` method on the `PostStoryForm` is invoked is yet to be explored.

NOTE *A common mistake by beginning Struts and JSP developers is to try to use the* `ActionForm` *class to manage Struts form data without using the Struts HTML tag libraries.*

It is important to note that all of the techniques shown for prepopulating a web form will only work with the Struts HTML JSP tag libraries.

Let's take a look at the PostStorySetupAction.java file and see how we can trigger the `reset()` method.

PostStorySetupAction.java

Triggering the `PostStoryForm.reset()` method does not require any coding in the PostStorySetupAction.java file. All that the `PostStorySetupAction` class is going to do is to forward the user's request to the postStoryContent.jsp file. So what role does PostStorySetupAction.java play, if its `execute()` method just forwards the user on to a JSP page? How is the `reset()` method in the `PostStoryForm` class called?

If you set a Struts `<action>` tag in the struts-config.xml file to use an `ActionForm` and tell the `ActionServlet` to put the `PostStoryForm` in the user's request, the `reset()` method in the `PostStoryForm` class will be invoked.

When users click the Post a Story link in the JavaEdge header, they are asking the `ActionServlet` to invoke the `/postStorySetup` action. This action is configured to use the `ActionForm` class of `PostStoryForm`. The `PostStoryForm` is going to be put in the users' request context by the `ActionServlet`.

Since the `ActionForm` class for the `/postStorySetup` action is the `PostStoryForm` class and the `PostStoryForm` class is going to be placed into the users' request context, the `reset()` method in the `PostStoryForm` class will be invoked. The `reset()` method is going to initialize each of the attributes in the `PostStoryForm` class to hold a set of simple instructions pulled from the ApplicationResources.properties file.

After the `reset()` method has been invoked, the `ActionServlet` will place any submitted form data in the `PostStoryForm` instance. Since the user has not actually submitted any data, the `PostStoryForm` class will still hold all of the values read from the ApplicationResources.properties file. The `ActionServlet` will then invoke the `execute()` method in the `PostStorySetupAction` class, which will forward the user to the postStoryContent.jsp page. This page will display a form, prepopulated with instructions.

In summary, to prepopulate the form, you need to perform the following two steps:

1. Write a Struts `Action` class called `PostStorySetupAction`. The `execute()` method of this class will pass the user on to postStoryContent.jsp.

2. Set up an action called `/postStorySetup` in the struts-config.xml file. This action will use the `PostStoryForm` class.

The code for PostStorySetupAction.java is shown here:

```
package com.apress.javaedge.struts.poststory;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class PostStorySetupAction extends Action {
  public ActionForward execute(ActionMapping mapping,
                               ActionForm form,
                               HttpServletRequest request,
                               HttpServletResponse response){
```

```
    return (mapping.findForward("poststory.success"));
  }
}
```

The `execute()` method just forwards the user to the postStoryContent.jsp page by returning an `ActionForward` mapped to this page:

```
return (mapping.findForward("poststory.success"));
```

The `poststory.success` mapping corresponds to the `<forward>` element, defined for the following `<action>` tag of `/postStorySetup`:

```
<action path="/postStorySetup"
        type="com.apress.javaedge.struts.poststory.PostStorySetupAction"
        name="postStoryForm"
        scope="request"
        validate="false">
  <forward name="poststory.success" path="/WEB-INF/jsp/postStory.jsp"/>
</action>
```

The `name` attribute shown here tells the `ActionServlet` to use an instance of `PostStoryForm` whenever the user invokes the `/postStorySetup` action.

```
name="postStoryForm"
```

Remember, the value of the `name` attribute must refer to a `<form-bean>` tag defined at the beginning of the struts-config.xml file.

The `scope` attribute tells the `ActionServlet` to place the `PostStoryForm` as an attribute in the `HttpServletRequest` object.

```
scope="request"
```

Setting the `validate` attribute to `false` in the preceding tag will cause the `ActionServlet` not to invoke the `validate()` method of the `PostStoryForm`. This means the `reset()` method in the `PostStoryForm` object is going to be invoked and placed in the user's request, but no data validation will take place.

Since no data validation takes place, the `execute()` method of `PostStorySetupAction` will be invoked. Remember, the `Action` class that carries out the end user's request is defined in the `type` attribute.

```
type="com.apress.javaedge.struts.poststory.PostStorySetupAction"
```

Another Technique for Prepopulation

Another technique exists for prepopulating an ActionForm with data. It is discussed here because implementing your Struts application using this technique can cause long-term maintenance headaches.

In the PostStorySetupAction.java file, you could implement the execute() method so that it creates an instance of PostStoryForm and invokes its reset() method directly. After the reset() method is invoked, the PostStoryForm can then be set as an attribute in the request object passed in the execute() method.

The following code demonstrates this technique:

```
public class PostStorySetupAction extends Action {
  public ActionForward execute(ActionMapping mapping,
                               ActionForm form,
                               HttpServletRequest request,
                               HttpServletResponse response){

    PostStoryForm postStoryForm = new PostStoryForm();
    postStoryForm.setServlet(this.getServlet());
    postStoryForm.reset(mapping, request);
    request.setAttribute("postStoryForm", postStoryForm);

    return (mapping.findForward("poststory.success"));
  }
}
```

This technique does not require you to provide any additional configuration information in the <action> tag. The preceding code carries out all the actions that the ActionServlet/RequestProcess would carry out.

However, using this approach has two long-term architectural consequences. First, the preceding approach has tightly coupled the PostStoryForm class to the PostStorySetupAction class. In the future, if the development team wants /postStorySetup to use something other than the PostStoryForm class for the prepopulation or form validation, it must rewrite the execute() method.

This becomes a tedious task if the PostStoryForm class is present throughout multiple applications and a developer needs to switch it with another ActionForm class. If the developer had used the first technique and associated PostStoryForm and PostStorySetup by declaring their usage in the struts-config.xml file, a few small changes to the file could have easily switched the ActionForm class populating the postStoryContent.jsp page.

The second problem is that this approach takes control of the ActionForm away from the ActionServlet. The responsibility for managing the ActionForm shifts from the ActionServlet to the application developer.

In addition, the problem with invoking the `reset()` method directly is that if the Struts development team changes how the `ActionForm` class is processed by the `ActionServlet`, the developers run the risk of having their application code break when they try to upgrade to the next release of Struts.

NOTE *If you find yourself working around the application framework, consider redesigning the task you are trying to execute. Stepping outside the application framework, as in the example shown previously, can lead to long-term maintenance and upgrade issues. The Struts architecture tries to remain very declarative, and controlling the application flow programmatically breaks one of Struts' fundamental tenets.*

Prepopulating a Form the Correct Way

If you are going to use a setup action and not the `reset()` method on an `ActionForm` to prepopulate a form with data, then you should do all of the work directly in the setup action. The code that follows demonstrates this:

```
public class PostStorySetupAction extends Action {
  public ActionForward execute(ActionMapping mapping,
                               ActionForm form,
                               HttpServletRequest request,
                               HttpServletResponse response){

    ActionServlet servlet = this.getServlet();
    PostStoryForm postStoryForm = new PostStoryForm();
    postStoryForm.setServlet(this.getServlet());

   MessageResources messageResources = servlet.getResources();

  storyTitle =
    messageResources.getMessage("javaedge.poststory.title.instructions");
  storyIntro =
    messageResources.getMessage("javaedge.poststory.intro.instructions");
  storyBody =
    messageResources.getMessage("javaedge.poststory.body.instructions");
  request.setAttribute("postStoryForm", postStoryForm);

    return (mapping.findForward("poststory.success"));
  }
}
```

If you look at this code, you will notice that you can directly retrieve and set Struts `ActionForm` classes in the user's `request` or `session` context.

```
storyTitle =
    messageResources.getMessage("javaedge.poststory.title.instructions");
  storyIntro =
    messageResources.getMessage("javaedge.poststory.intro.instructions");
  storyBody =
    messageResources.getMessage("javaedge.poststory.body.instructions");
  request.setAttribute("postStoryForm", postStoryForm);
```

At some point as a Struts developer you will need to retrieve, create, or manipulate an `ActionForm` manually. Keep the following in mind:

NOTE *The Struts framework always uses the value stored in the* `name` *attribute of an* `<action>` *element as the key to storing the* `ActionForm` *class as the user's* `request` *or* `session`.

Validating the Form Data

As discussed earlier, a common mistake in web application development is for no clear distinction to exist between the application's business logic and validation logic. The `ActionForm` class helps the developers to solve this problem by allowing them to enforce lightweight validation rules against the data entered by a user. By encapsulating these validation rules in the `ActionForm` class, the developer can clearly separate the validation rules from the business logic that actually carries out the request. The business logic is placed in the corresponding `Action` class for the end user's request.

Web developers can override the `validate()` method and provide their own validation rules for the submitted data, while writing their own `ActionForm` class. If the developers do not override the `validate()` method, none of the data submitted will have any validation logic run against it.

The `validate()` method for the `PostStoryForm` class is going to enforce three validation rules:

- The users must enter a story title, story introduction, and story body. If they leave any field blank, they will receive an error message indicating that they must enter the data.
- The users are not allowed to put vulgarity in their application. The `validate()` method will check the data entered by the user for any inappropriate phrases.
- Each field in the Post a Story page is not allowed to exceed a certain length; otherwise, the user will get an error message.

It is important to note that in all the cases, the users will not be allowed to continue until they correct the validation violation(s).

The `validate()` method for the `PostStoryForm` class is as shown here:

```
public ActionErrors validate(ActionMapping mapping,
                             HttpServletRequest request) {
  ActionErrors errors = new ActionErrors();

  checkForEmpty("Story Title", "error.storytitle.empty",
                getStoryTitle(), errors);
  checkForEmpty("Story Intro", "error.storyintro.empty",
                getStoryIntro(), errors);
  checkForEmpty("Story Body",  "error.storybody.empty",
                getStoryBody(), errors);

  checkForVulgarities("Story Title", "error.storytitle.vulgarity",
                      getStoryTitle(), errors);
  checkForVulgarities("Story Intro", "error.storyintro.vulgarity",
                      getStoryIntro(), errors);
  checkForVulgarities("Story Body",  "error.storybody.vulgarity",
                      getStoryBody(), errors);

  checkForLength("Story Title", "error.storytitle.length", getStoryTitle(),
                 100, errors);
  checkForLength("Story Intro", "error.storyintro.length", getStoryIntro(),
                 2048, errors);
  checkForLength("Story Body", "error.storybody.length", getStoryBody(),
                 2048, errors);

  return errors;
}
```

The first step in the `validate()` method is to instantiate an instance, called `errors`, of the `ActionErrors` class.

```
ActionErrors errors = new ActionErrors();
```

The `ActionErrors` class is a Struts class that holds one or more instances of an `ActionError` class. An `ActionError` class represents a single violation of one of the validation rules being enforced in the `ActionForm` class.

NOTE *The Struts framework's* `ActionError` *class is used throughout all of the code examples in this book. As of Struts 1.2.1, the* `ActionError` *class will be deprecated and replaced by the* `ActionMessage` *class.*

However, the release of Struts 1.0.2 to Struts 1.1 took over a year to release to production. The next release of Struts 1.2 is supposed to be minor bug fixes with no major functionality included. As such, new and existing applications using Struts application have a while before they need to be upgraded.

If a form element submitted by an end user violates a validation rule, an `ActionError` will be added to the `errors` object.

When the `validate()` method completes, the `errors` object will be returned to the `ActionServlet`:

```
return errors;
```

If the `errors` object is `null` or contains no `ActionErrors`, the `ActionServlet` will allow the business logic to be carried out, based on the end user's request. This is done by invoking the `execute()` method in the `Action` class associated with the request.

Let's look at the `checkForVulgarities()` method to see how an `ActionError` class is actually created when a validation rule is violated. The `checkForEmpty()` and `checkForLength()` methods will not be discussed in detail, but the code for these methods is shown here:

```
private void checkForEmpty(String fieldName, String fieldKey, String value,
                           ActionErrors errors) {
  if (value.trim().length() == 0) {
    ActionError error = new ActionError("error.poststory.field.null",
                                        fieldName);
    errors.add(fieldKey, error);
  }
}


private void checkForLength(String fieldName, String fieldKey, String value,
                            int maxLength, ActionErrors errors){
  if (value.trim().length() > maxLength){
    ActionError error = new ActionError("error.poststory.field.length",
                                        fieldName);
    errors.add(fieldKey, error);
  }
```

Creating an ActionError

The `checkForVulgarities()` method is as shown here:

```
private void checkForVulgarities(String fieldName, String fieldKey,
                                 String value, ActionErrors errors) {

  VulgarityFilter filter = VulgarityFilter.getInstance();

  if (filter.isOffensive(value)){
    ActionError error = new ActionError("error.poststory.field.vulgar",
                                        fieldName);
    errors.add(fieldKey, error);
  }
}
```

The first line in this method retrieves an instance of the `VulgarityFilter` into a variable called `filter`.

```
VulgarityFilter filter = VulgarityFilter.getInstance();
```

The `VulgarityFilter` class is implemented using a Singleton design pattern and wraps a collection of words that are considered to be offensive. The code for the class is shown here:

```
package com.apress.javaedge.common;

public class VulgarityFilter {

  private static VulgarityFilter filter = null;

  private static String[] badWords = {"Stupid", "Idiot", "Moron", "Dummy",
                                      "Flippin", "Ninny"};

  static {
    filter = new VulgarityFilter();
  }

  public static VulgarityFilter getInstance(){
    return filter;
  }

  public boolean isOffensive(String valueToCheck){
    String currentWord = "";
```

```
    for (int x = 0; x <= badWords.length - 1; x++){
      if (valueToCheck.toLowerCase().indexOf(badWords[x].toLowerCase())
          != -1) {
        return true;
      }
    }

  return false;
  }
}
```

The `VulgarityFilter` class has a single method called `isOffensive()`, which checks if the text passed in is offensive. A value of `true` returned by this method indicates the user has entered data that contains offensive text:

```
if (filter.isOffensive(value))
```

When a vulgarity is found, a new `ActionError` is created and added to the `errors` object passed to the `checkForVulgarity()` method:

```
ActionError error = new ActionError("error.poststory.field.vulgar",
                                    fieldname);
errors.add(fieldKey, error);
```

There are five constructors that can be used to instantiate an `ActionError` class. The first parameter of each of these constructors is a lookup key that Struts uses to find the text of the error message displayed to the end user. Struts will look for all error messages in the ApplicationResources.properties file associated with the application. The error messages for the Post a Story page are shown here:

```
error.poststory.field.null=

 The following field: {0} is a required field. Please provide a value for {0}.<br/>
error.poststory.field.vulgar=
 You have put a vulgarity in your {0} field. Please refer to our
 <a href="/javaedge/policy.html">terms of use policy.</a><br/>
error.poststory.field.length=Your {0} field is too long.<br/>
```

When the user violates the vulgarity validation rule and the `checkForVulgarity()` method creates an `ActionError`, the lookup key `error.poststory.field.vulgar` will be used to return the following error message:

```
The following field: {0} is a required field. Please provide a value for {0}.<br/>
```

The error message can contain at most four distinct parameter values. The parameter values are referenced by using the notation {number}, where number is between zero and three. In the preceding example, only one parameter is inserted into the error message. A summary of the five constructors in the ActionError class is given here:

ActionError Constructor	Description
ActionError(String lookupKey)	Retrieves the error message from the ApplicationResources.properties file
ActionError(String lookupKey, String param0)	Retrieves the error message from the ApplicationResources.properties file and passes in one parameter
ActionError(String lookupKey, String param0, String param1)	Retrieves the error message from the ApplicationResources.properties file and passes in two parameters
ActionError(String lookupKey, String param0, String param1, String param2)	Retrieves the error message from the ApplicationResources.properties file and passes in three parameters
ActionError(String lookupKey, String param0, String param1, String param2, String param3)	Retrieves the error message from the ApplicationResources.properties file and passes in four parameters

After the error object has been created, it is later added to the errors object by calling the add() method in errors:

```
errors.add(fieldKey, error);
```

The add() method takes two parameters:

- A key that uniquely identifies the added error within the ActionErrors class. This key must be unique and can be used to look up a specific error in the ActionErrors class.
- An ActionError object containing the error message.

Viewing the Errors

The Struts ActionServlet checks if there are any errors in the returned ActionErrors object to determine if a validation error was returned by the validate() method. If

the value returned from the `validate()` method is `null` or contains no `ActionError` objects, no validation errors were found.

If the Struts `ActionServlet` finds that there are errors present in the `ActionError` object, it will redirect the user to the path set in the `input` attribute for the action.

NOTE *Remember, the* `input` *attribute on the* `<action>` *tag is a required field if the* `name` *attribute is also defined on the tag. The* `name` *attribute is used to define the name of the* `ActionForm` *that will hold the form data being submitted.*

Failure to include an `input` *attribute when using an* `ActionForm` *will result in an exception being thrown.*

Most of the time, the value in this `input` tag is the JSP page where the data was entered. The `ActionForm` object holding the user's data will still be in the request. Thus, any data entered by the user in the form will appear prepopulated in the form. How does Struts present the user with all the errors raised in the `validate()` method? It does this using the `<html:errors/>` tag. This tag is found in the Struts HTML custom JSP tag library. (Several other form-related custom tags are contained in the HTML tag library. We will be discussing the full HTML tag library in the section "The Struts HTML Tag Libraries.") There are two ways of using this tag:

- To write each error message stored within the `ActionErrors` class to the JSP `out` `PrintWriter` class
- To retrieve a specific error from the `ActionErrors` class and place it next to the specific fields

Writing All Error Messages to the JSP Page

To perform the first action, you must import the Struts HTML tag library and place the `<html:errors/>` tag where you want the errors to appear. For instance, in postStoryContent.jsp, you use this tag in the following manner:

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>

<BR/><BR/>
<H1><bean:message key="javaedge.poststory.text.header"/></H1>

<html:errors/>
```

This code will write all the errors in the `ActionErrors` class returned by `validate()` method of the `PostStoryForm` immediately below the header of the page. The following example shows the type of error messages that can be presented to the end user:

```
You have put a vulgarity in your Story Title field.
Please refer to our <a href="/javaedge/policy.html">terms
of use policy.</a><br/>
The following field: Story Intro is a required field.
Please provide a value for Story Intro.<br/>
The following field: Story Body is a required field.
Please provide a value for Story Body.<br/>
```

It is extremely important to note that the `<html:errors/>` tag will write the error text exactly as it has been defined in the ApplicationResources.properties file. This means that the developer must provide HTML tags to format the appearance of the error message. This also includes putting any `<br/>` tags for the appropriate line breaks between the error messages. The `<html:errors/>` tag allows the application developer to define a header and footer for a collection of error messages. Headers and footers are defined by including an `errors.header` property and `errors.footer` property in the ApplicationResources.properties file. These two properties can contain text (and HTML code) that will appear immediately before and after the errors written by the `<html:errors/>` tag. The following snippet shows these properties for the JavaEdge application:

```
errors.header=<h3><font color="red">Important Message</font></h3><ul>
errors.footer=</ul><hr>
```

The `<html:errors/>` tag provides a very simple and consistent error handling mechanism. Front-end screen developers only need to know that they have to put an `<html:errors/>` tag in their JSP form pages to display any validation errors. The job of the server-side developers is simplified because they can easily validate the form data submitted by the end user and communicate the errors back to the user by populating an `ActionErrors` object.

Keeping in mind all the discussion that we had so far, when the end users violate a validation rule on the Post a Story page, they will see the output shown in Figure 3-4.

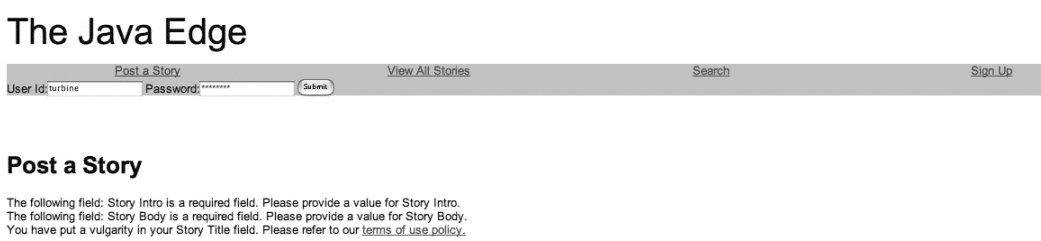

Figure 3-4. The end result of a validation rule violation

Retrieving a Single Error Message

The `<html:errors/>` tag by itself is somewhat inflexible, because you have to present all the validation errors caused by the end user at a single spot on the screen. Many application developers like to break the validation errors apart and put them next to the field that contains the invalid data.

Fortunately, the `<html:errors/>` tag allows you to pull a single error message from an `ActionErrors` object. It has an attribute called `property`. This attribute will let you retrieve an error message, using the key value that was used while adding the error message to the `ActionErrors` object. For example, when a user enters a word that violates the vulgarity filter, you add that validation error to the `errors` object by calling

```
errors.add(fieldKey, error);
```

The `fieldKey` variable passed to the `errors.add()` method is the name we have chosen to represent that particular error. For example, if the user typed the word *dummy* into the Story Title field, this would violate the vulgarity validation rule and a new `ActionError` class would be instantiated. The new `ActionError` would be added to the `errors` class and would have a `fieldKey` value of `error.storytitle.vulgarity`.

If you wanted to put that specific error message directly above the Story Title field label, you could rewrite postStoryContent.jsp with the following code:

```
<TR>
  <TD>
    <font size="1" color="red">
      <html:errors property="error.storytitle.vulgarity"/>
    </font>
    <bean:message key="javaedge.poststory.form.titlelabel"/>
    <html:text name="postStoryForm" property="storyTitle"/>
  </TD>
</TR>
```

By using the <html:errors/> tag in the manner shown, you can cause postStoryContent.jsp to generate an error message that may look like the one shown in Figure 3-5.

Figure 3-5. Displaying a single validation error message

If you want to automatically format the individual error messages, you need to use `error.prefix` and `error.suffix` rather than the `error.header` and `error.footer` properties in the ApplicationResources.properties file.

```
error.prefix=<font size="1" color="red">
error.suffix=</font>
```

Error Handling and Prepopulation

After discussing how HTML errors are handled in Struts, you might be a little bit confused. Why does the form show up with all of the fields prepopulated with the data that the user just entered? Why doesn't the `reset()` method in the `ActionForm` class reset all the values?

The reason is simple. When the `validate()` method is invoked and if there are validation errors, the `ActionServlet` is going to look at the value of the `input` attribute in the `<action>` tag. The `input` attribute almost invariably points back to the JSP where the user entered the data. Remember, the `reset()` method gets called only when an action is invoked. Redirecting the user back to a JSP page will not invoke the `reset()` method. If the JSP page to which the user is redirected uses the Struts HTML tag library and an `ActionForm` in the user's request or session, it will pull the data out of the `ActionForm` and prepopulate the form elements with that data. Thus, when a validation error occurs, the user sees the validation errors and a prepopulated form.

If you want to force the reset of all the elements in a form, after the validation occurs, you need to point the `input` attribute in the `<action>` element to a setup action that will repopulate the data.

On Validations and Validation Libraries

One of the biggest complaints that we have heard from development teams using Struts is that it seems wrong to separate the validation logic from the actual business logic. After all, the same validation logic is going to be applied regardless of where the actual business logic is being executed. For example, a parameter that is required by a piece of business logic to be non-null is going to have the same requirement regardless of which application is executing the business logic.

The strength of the `ActionForm` class's `validate()` method is that it provides a clean mechanism for performing validation *and* handling errors that are found during validation. The examples in this chapter have shown the validation rules for the code embedded directly in the `ActionForm` class doing the validation. This has been to simplify the reading of code and allow the reader to follow the examples without having to wade through multiple layers of abstraction and generalization.

The problem with embedding the validation logic inside the `ActionForm` class is that it ties the code to a Struts-specific class and makes the embedded validation code difficult to reuse in a non-Struts application.

Oftentimes, development teams will leverage a number of approaches to help generalize validation and avoid tying it to a Struts `ActionForm` class. These include

- Separating all of the validation logic used in an application into a set of validation "helper" classes that can be reused across multiple applications.

- Moving the validation code into the value objects being used to move data back and forth across the application tiers. The base value object class extended by all of the value objects in the application has a `validate()` method that can be overridden to contain validation code. If you are not familiar with the Value Object pattern, please refer to Chapter 5.

- Moving all of the validation code into the business logic layer. Each object in the business logic layer has a private `validate()` method that is called before the actual business logic is processed.

- Using a Validation framework, like the Validator framework in Struts, to externalize the validation logic from the business logic and make them as declarative as possible.

Each of the items listed have their advantages and disadvantages. Moving all of the validation logic to a set of "helper" classes is simple, but oftentimes leads to the development team creating a cumbersome set of low-level data validation calls that they must maintain and support. There are already plenty of open source libraries and frameworks that do this type of low-level validation. The question becomes, Why waste time on something others have already done?

Moving the validation logic to the Value Object pattern has the advantage of putting the validation logic very close to the data. The same validation logic for data can be applied over and over again by simply invoking the `validate()` method on the value object. The problem with this approach is that the value objects are supposed to be lightweight "wrappers" for data being passed across the different application boundaries (presentation, business, and data). At any given time there can be a large number of value objects being used with only a small fraction of them actually being validated. This is a lot of extra overhead for nothing.

Moving the validation logic to the business layer and embedding it inside of a business object makes sense. After all, one of the first rules of object-oriented programming is that all data and the code that acts on that data should be self-contained within a single discrete class. Oftentimes when validation rules are built into a business layer class, nonbusiness layer details that deal with error handling and error presentation are also embedded in the class. This results in tight dependencies being created on the business object and violates another tenet of OO programming, the concept of Single Responsibility.

Classes and the methods contained within them should have a discrete set of responsibilities that reflect on the domain being modeled by the class. When other pieces of nondomain-specific logic start creeping into the class, it becomes bloated and difficult to maintain. This is one of the principal reasons why the Struts `ActionForm` class is useful: It allows a developer to write validation logic without getting the business logic used in the class too tightly tied to the application.

The last option is our favorite. If you can use a framework that specifically is built for validation, you can save a lot of time. The Struts `ActionForm` class's `validate()` method is only meant to be a plug-in point from which validation logic is called. However, if you start from the premise that validation logic is lightweight and will consist of no more than a handful of standard checks, using a declarative validation framework where you have to write little to no code for performing validation is the best approach. The Struts 1.1 framework now integrates with the Validator framework. This framework lets you declare a set of validation rules that can be processed against data contained within an `ActionForm` class.

The validation rules in the Validator framework cover most of the validation rules a developer is going to need while building an application. In addition, the Validator framework is extensible enough to allow you to build your own validation rules. The Validator framework will be covered in greater detail in Chapter 7.

The Struts HTML Tag Library

As we have seen earlier in this chapter, Struts provides the `ActionForm` and the `Action` classes as the means of validating and processing the data submitted by the end user. The Struts development framework also provides a JSP tag library,

called the HTML tag library, that significantly simplifies the development of HTML-based forms. The HTML tag library allows the developer to write JSP pages that tightly integrate with an `ActionForm` class.

The Struts HTML tag library can be used to generate HTML form controls and read data out of an `ActionForm` class in the user's session or request. It also helps developers avoid writing significant amounts of scriptlet code to pull the user data out of JavaBeans (that is, the `ActionForm` objects) in the user's request and/or session. When combined with the other Struts tag libraries, as discussed in Chapter 2 (see the section called "Building the homepage.jsp"), a developer can write very dynamic and data-driven web pages without ever having to write a single line of JSP scriptlet code.

The Struts HTML tag library contains a wide variety of tags for generating HTML form controls. We are not going to cover every tag in the Struts HTML tag library. Instead, we are going to go through the most commonly used tags and explore their usage. For a full list of the different tags available in the Struts HTML tag library, you can visit `http://jakarta.apache.org/struts`. The tags discussed in this chapter include the following:

Tag Name	Tag Description
`<html:form>`	Renders an HTML `<form>` tag
`<html:submit>`	Renders a submit button
`<html:cancel>`	Renders a cancel button
`<html:text>`	Renders a text field
`<html:textarea>`	Renders a textarea field
`<html:select>`	Renders an HTML `<select>` tag for creating drop-down boxes
`<html:option>`	Renders an HTML `<option>` control that represents a single option in a drop-down box
`<html:checkbox>`	Renders an HTML checkbox
`<html:radio>`	Renders an HTML radio control

Let's begin the discussion of the Struts HTML tag library by looking at the postStoryContent.jsp page:

```
<%@ page language="java" %>
<%@ taglib uri="/taglibs/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/taglibs/struts-html.tld" prefix="html" %>
<%@ taglib uri="/taglibs/struts-logic.tld" prefix="logic" %>
<%@ taglib uri="/taglibs/struts-tiles.tld" prefix="tiles" %>
```

```
<BR/>
<BR/>
<H1>
  <bean:message key="javaedge.poststory.text.header"/>
</H1>

<html:errors/>

<html:form action="postStory">
  <TABLE>
    <TR>
      <TD>
        <bean:message key="javaedge.poststory.text.intro"/>
        <logic:present scope="session" name="memberVO">
          <B>
            <bean:write name="memberVO" scope="session"
                        property="firstName"/> 
            <bean:write name="memberVO" scope="session"
                        property="lastName"/>
          </B><BR/>
        </logic:present>

        <logic:notPresent scope="session" name="memberVO">
          <B>Anonymous</B>
        </logic:notPresent>

      </TD>
      <TD>
        <BR/><BR/> 
      </TD>
    </TR>

    <TR>
      <TD>
        <bean:message key="javaedge.poststory.form.titlelabel"/>
        <html:text property="storyTitle"/>
      </TD>
    </TR>

    <TR>
      <TD>
        <bean:message key="javaedge.poststory.form.introlabel"/>
        <html:textarea property="storyIntro" cols="80" rows="5"/>
      </TD>
    </TR>
```

```
    <TR>
     <TD>
       <bean:message key="javaedge.poststory.form.bodylabel"/>
       <html:textarea property="storyBody" cols="80"
                              rows="10"/>
     </TD>
   </TR>

    <TR>
      <TD align="center"><html:submit property="submitButton"
          value="Submit"/>
      </TD>
    </TR>
  </TABLE>
</html:form>
```

Setting Up a Struts HTML Form

Before using the individual Struts HTML tag within a JSP page, three steps must be undertaken:

1. Import the Struts HTML Tag Library Definitions (TLDs).

2. Define an `<html:form>` tag, within the page that will map to an `<action>` tag defined in the struts-config.xml file.

3. Define an `<html:submit>` button to allow the user to submit the entered data.

The Struts HTML TLD is imported as shown here:

```
<%@ taglib uri="/taglibs/struts-html.tld" prefix="html" %>
```

Next, you use the Struts HTML tags. Just as in a static HTML page, you need to define a `<form>` tag that will encapsulate all the HTML form controls on the page. This is done by using the `<html:form>` tag.

```
<html:form action="postStory">
    ...
</html:form>
```

The <html:form> tag has a number of different attributes associated with it. However, we will not be discussing every <html:form> attribute in detail. Some of the <html:form> attributes are given here:

Attribute Name	Attribute Description
action	Maps to the <action> tag that will carry out the user's request when the form data is submitted. This is a required field.
method	Determines whether the form will be sent as a GET or POST. This is not a mandatory field and if not specified, it will generate the <form> tag to use a POST method.
name	The name of the JavaBean that will be used to prepopulate the form controls. The <html:form> tag will check if this bean is present in the user's session or request. The scope attribute defines whether to look into the user's session or request. If no JavaBean is found in the context defined in the scope attribute, the <html:form> tag will create a new instance of the bean and place it into the scope defined by the scope attribute. The class type of the created JavaBean is determined by the type attribute.
scope	Determines whether the tag should look in the user's session or request for the JavaBean named in the name attribute. The value for this attribute can be either "session" or "request".
type	Fully qualified Java class name for the JavaBean being used to populate the form.
onsubmit	Lets the developer define a JavaScript onSubmit() event handler for the generated form.
onreset	Lets the developer define a JavaScript onReset() event handler for the generated form.
focus	Name of the field that will have focus when the form is rendered.

The most important of these attributes is the action attribute. It maps to an <action> element defined in the struts-config.xml file. If no name, scope, or type attribute is specified in the <html:form> tag, the ActionForm that will be used to populate the form, its fully qualified Java name, and the scope in which it resides will be pulled from the <action> tag in the struts-config.xml file.

In the <html:form> tag used in the postStoryContent.jsp, all the ActionForm information would be retrieved by the ActionServlet, by looking at the name attribute in the <action> tag of the postStory action in the struts-config.xml file.

```
<action path="/postStory"
  input="/WEB-INF/jsp/postStory.jsp"
  name="postStoryForm"
  scope="request"
  validate="true"
  type="com.apress.javaedge.struts.poststory.PostStory">
  <forward name="poststory.success" path="/execute/homePageSetup"/>
</action>
```

Since the value of `name` (`postStoryForm`) is defined as a `<form-bean>` element in the struts-config.xml file, the `ActionServlet` can figure out its fully qualified Java class name and instantiate an instance of that class.

NOTE *It is a good practice to use the* `action` *attribute rather than the* `name`, `scope`, *and* `type` *attributes to define the JavaBean that will populate the form. Using this attribute gives you more flexibility by allowing you to change the* `ActionForm` *class in one location (struts-config.xml) rather than searching multiple JSP pages.*

Let's look at the HTML generated by the `<html:form>` tag shown earlier:

```
<form name="postStoryForm" method="POST"
      action="/javaedge/execute/postStory">
```

The `name` attribute generated tells the `ActionServlet` of Struts that the `postStoryForm` bean, defined in the `<form-beans>` tag of the struts-config.xml, is going to be used to hold all the data posted by the user. The default method of the form (since you did not define one in the `<html:form>` tag) is going to be a POST method. The `action` attribute contains the URL to which the form data is going to be submitted. Since the `action` of the `<html:form>` tag was `postStory`, the `<html:form>` generated the `action` attribute (for the corresponding `<form>` tag) as `/javaedge/execute/postStory`.

The last step in setting up an HTML form is using the Struts `<html:submit>` tag to generate an HTML submit button:

```
<html:submit property="submitButton" value="Submit"/>
```

In addition to the `<html:submit>` tag, the Struts tag library has HTML tags for creating cancel buttons. When an `<html:cancel>` tag is used, an HTML button will be rendered, which when clicked will cause the `ActionServlet` to bypass the `validate()` method in the `ActionForm` that is associated with the form.

Even though the validate() method is bypassed, the execute() method for the Action class (in this case PostStory.java) linked with the form will be invoked. This means if you want to use an <html:cancel> button in your page, the execute() method must detect when the cancel button is invoked and act accordingly. For instance, let's say the following <html:cancel> tag was added to the postStoryContent.jsp file:

```
<html:cancel value="Cancel"/>
```

The validate() method in the PostStoryForm class would not be called. However, the execute() method on the PostStory class will be invoked. The execute() method taken from the PostStory class could be written in the following manner:

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response){

  if (this.isCancelled(request)){
    System.out.println("*****The user pressed cancel!!!");
    return (mapping.findForward("poststory.success"));
  }

  //Add the story data to the database.
  ...
  return (mapping.findForward("poststory.success"));
}
```

If you did not want the code in the execute() method to be executed, you will have to use a method called isCancelled() to detect if the user pressed a cancel button. The isCancelled() method is inherited from the base Struts Action class. This method looks for a parameter in the user's request, called org.apache.struts.taglib.html.CANCEL. If it finds this parameter, it will return true, indicating to the developer writing the execute() method code that the user clicked the cancel button.

The parameter name, org.apache.struts.taglib.html.CANCEL, maps to the name attribute in the <input> tag generated by the <html:cancel> button. The HTML button generated by the <html:cancel> tag shown earlier looks like this:

```
<input type="submit" name="org.apache.struts.taglib.html.CANCEL"
        value="Cancel">
```

Unlike the <html:submit> tag, the property attribute on the <html:cancel> tag is rarely set.

NOTE *If you set the* `property` *attribute in the* `<html:cancel>` *button, it will override the default value generated, and you will not be able to use the* `isCancelled()` *method to determine if the user wants to cancel the action.*

Using Text and TextArea Input Fields

The postStoryContent.jsp files use text `<input>` and textarea tags to collect the data from the end user. The `<html:text>` and `<html:textarea>` tags are used to generate the `text` and `textarea` `<input>` tags, respectively. For instance, the `postContent.jsp` page uses the `<html:text>` tag to generate an HTML text `<input>` tag by using the following:

```
<html:text property="storyTitle"/>
```

The `<html:text>` tag has a number of attributes, but the most important are `name` and `property`. The `name` attribute defines the name of the `ActionForm` bean that the input field is going to map to. The `property` attribute defines the property in the `ActionForm` bean that is going to map to this input field. You should keep in mind two things while working with the `property` attribute:

- The `property` attribute will map to a `get()` and `set()` method in the `ActionForm` bean. This means that value must match the standard JavaBean naming conventions. For instance, the value `storyTitle` is going to be used by the `ActionServlet` to call the `getStoryTitle()` and `setStoryTitle()` methods in the `ActionForm`.

- The value in a `property` attribute can be nested by using a "`.`" notation. Let's assume that the `ActionForm` method had a `property` called `member` that mapped to a `MemberVO` object containing the user data. The developer could set the value of the `property` attribute to be `member.firstName`. This would translate into a call to the `getMember().getFirstName()` and `getMember().setFirstName()` methods of the `PostStoryForm` class.

NOTE *If you refer to the Struts documentation on the Jakarta web site, you will notice that almost every Struts HTML tag has a* `name` *attribute in it. This attribute is the name of the JavaBean that the HTML tag will read and write data to. You do not have to supply a* `name` *attribute for the HTML form attributes we are describing in the following sections. If you do not supply a* `name` *attribute and if the* `<html:*>` *control is inside an* `<html:form>` *tag, the* `<html:*>` *control will automatically use the* `ActionForm` *associated with the* `<html:form>` *tag.*

The <html:textarea> input tag behaves in a similar fashion to the <html:text> tag. The <html:textarea> tag uses the cols and rows attributes to define the width and length of the textarea the user can type in:

```
<html:textarea name="postStoryForm" property="storyIntro" cols="80" rows="5"/>
```

The preceding tag will generate an <textarea> tag called storyIntro that will be 80 columns wide and five rows long.

Drop-Down Lists, Checkboxes, and Radio Buttons

Most HTML forms are more than just a collection of the simple text field controls. They use drop-down lists, checkboxes, and radio buttons to collect a wide variety of information. While the postStoryContent.jsp file did not contain any of these controls, it is important to understand how the Struts framework renders these controls using the HTML tag library. Let's begin the discussion by the looking at drop-down lists.

Drop-Down Lists

An HTML drop-down list control provides a list of options that a user can select from. However, the user sees only the item that has been selected. All of the other items are hidden until the user clicks the drop-down box. On clicking the box, the rest of the options will be displayed and the user will be able to make a new choice.

Since the Post a Story page does not have a drop-down box, we will have to step away from it briefly. Using the Struts HTML tag library, there are two ways of rendering a drop-down box:

- Use an <html:select> tag and build a static list of options by hard coding a static list of <html:option> tags in the code.

- Use an <html:select> tag and dynamically build the list by reading the data from a Java collection object using the <html:options> tag.

The <html:select> tag renders a <select> tag in HTML. The <html:option> tag renders a single option for the placement in the drop-down list. If you want to display a drop-down list containing a list of name prefixes, you would write the following code in your JSP file:

```
<html:select property="someBeanProperty">
  <html:option value="NS">Please select a prefix</html:option>
  <html:option value="Mr.">Mr.</ html:option>
```

```
  <html:option value="Ms.">Ms.</ html:option>
  <html:option value="Mrs.">Mrs.</ html:option>
  <html:option value="Dr.">Dr.</ html:option>
</html:select>
```

This code snippet would generate the following HTML:

```
<select name="someBeanProperty">
  <option value="NS">Please select a prefix</option>
  <option value="Mr.">Mr.</option>
  <option value="Ms.">Ms.</option>
  <option value="Mrs.">Mrs.</option>
  <option value="Dr.">Dr.</option>
</select>
```

The `<html:select>` tag has one important attribute, the `property` attribute. It is the name of the property of the `ActionForm` bean that will store the item selected from the drop-down list. The `<html:option>` tag must always be contained within an `<html:select>` tag. The `value` attribute in the `<html:option>` tag specifies the value that will be sent in the users' request for the selected item from the drop-down list when they hit the submit button.

The `<html:select>` and `<html:option>` tag work well while generating a drop-down list that does not change. However, if you want to create a drop-down list based on data that is dynamic, such as data pulled from a database, you need to the use the `<html:options>` tag. The `<html:options>` tag allows you to generate an `<option>` list from a Java `Collection` object.

Let's assume that in a `SetupAction` class, you created a `Vector` object and populated it with the prefix codes. You then put that code in the `request` object as shown here:

```
Vector prefixes = new Vector();
prefixes.add("NS");
prefixes.add("Mr.");
prefixes.add("Ms.");
prefixes.add("Mrs.");
prefixes.add("Dr.");
request.setAttribute("prefixes", prefixes);
```

You could then render this collection into a drop-down list using the following code:

```
<html:select property="someBeanProperty">
  <html:options name="prefixes">
</html:select>
```

Checkboxes

Setting up a checkbox to appear on an HTML form is easy to do. It just requires the use of a checkbox flag. To create a checkbox on a form, you can use the following syntax:

```
<html:checkbox property="someBeanProperty" value="true"/>
```

The `property` attribute for the checkbox matches the name of the `property` in the `ActionForm` that the checkbox is going to get and set data from. The `value` attribute is the value that will be sent in the HTTP request if the user checks the checkbox. If no value is specified, then the default value will always be `on`.

One important thing to remember is that when a checkbox is not checked, no value will be passed in the HTTP request. This also means that the value that was already set in the `ActionForm` property associated with the checkbox will not change. You have to check the `request` to see if the checkbox is present in the request. If it is not, you have to set the `ActionForm` property to a `false` or `off` value:

```
if (request.getAttribute("someBeanProperty") == null) {
  this.setSomeBeanProperty(false);
}
```

This is important because if the submitted data has a validation error and the `ActionServlet` returns the user to the screen where they entered data, any checkboxes that had been moved from a checked to an unchecked state will still show up on the screen as checked.

So in the `validate()` method of your `ActionForm` bean you must check the `request` object for the checkbox parameter. If the checkbox parameter has not been submitted as part of the request, you must set the corresponding property in the `ActionForm` to be `false`. This has to be done before you start doing any validation of the form data, or else you will end up with your form data inconsistently handling the checkbox information passed to it. This also means that if you want to prepopulate a form with checkboxes set in an off status, the `reset()` method of the `ActionForm` being used to populate the page must set the properties in the `ActionForm` (that map to checkboxes) to a `false` value.

Radio Buttons

To render a radio button in a form, you use the `<html:radio>` tag. This tag has two core attributes: `property` and `value`. These two attributes are similar in behavior to the `<html:checkbox>` tag. The `property` attribute defines the name of the property in the `ActionForm` that the radio button maps to. The `value` attribute is the value that will be sent, if the radio button is selected when the user submits the form.

NOTE *If you do not preset a radio button and no radio button is selected by the user, the property on the* `ActionForm` *representing the radio button will not have a value set on it.*

Use one of the `ActionForm` *prepopulation techniques described earlier if you require the radio button to have some default value associated with it.*

To group a set of radio button controls together so that only one of a group of radio buttons can be set, you set each radio button's `property` attribute to point to the same `ActionForm` property.

If you wanted to use a radio button instead of the drop-down list to show a selection of prefixes to the user, you could write the following code:

```
<LI>Mr. <html:radio property="someBeanProperty" value="Mr."/>
<LI>Ms. <html:radio property="someBeanProperty" value="Ms."/>
<LI>Mrs. <html:radio property="someBeanProperty" value="Mrs."/>
<LI>Dr. <html:radio property="someBeanProperty" value="Dr."/>
```

The HTML generated by this code would look as shown here:

```
<LI>Mr. <input type="radio" name="someBeanProperty" value="Mr.">
<LI>Ms. <input type="radio" name="someBeanProperty" value="Ms.">
<LI>Mrs. <input type="radio" name="someBeanProperty" value="Mrs.">
<LI>Dr. <input type="radio" name="someBeanProperty" value="Dr.">
```

Building More Dynamic ActionForms

The concept of wrapping data within an `ActionForm` is a powerful one because it allows the application developer to retrieve and manipulate data submitted by the end user without having to litter their code with the gory details of accessing an `HttpRequest` object. However, as most developers will quickly discover, for large projects that collect significant amounts of data, building `ActionForm` classes can be extremely tedious.

Writing individual `get()/set()` methods for each attribute being submitted by the end user is a time-consuming and thus error-prone process. Fortunately, Struts 1.1 now provides two mechanisms to simplify the process of building `ActionForm` classes:

- Dynamic `ActionForm`
- `Map`-backed `ActionForm`

Dynamic ActionForms allow the developer to declare a Struts form bean and its corresponding attributes in the application's struts-config.xml file. We are not going to go into any greater detail on Dynamic ActionForms in this chapter. Instead, we will cover them in greater detail in Chapter 7.

Map-backed ActionForm are an exciting new addition to the Struts framework. A Map-backed ActionForm allows the developer to wrap a Map object (that is, HashMap, TreeMap, etc.) and expose it on the ActionForm class.

Shown next is the PostStoryForm rewritten as a Map-backed ActionForm:

```
package com.apress.javaedge.struts.poststory;

import com.apress.javaedge.common.VulgarityFilter;
import org.apache.struts.action.*;
import org.apache.struts.util.MessageResources;

import javax.servlet.http.HttpServletRequest;
import java.util.HashMap;
import java.util.Map;

public class PostStoryMapForm extends ActionForm{
    private HashMap attributeMap = new HashMap();


    private Map getMap(){
       return attributeMap;
    }

    public void setAttribute(String attributeKey, Object attributeValue){
      getMap().put(attributeKey, attributeValue);
    }

public Object getAttribute(String attributeKey){

         Object holder = getMap().get(attributeKey);

         if (holder==null) return "";

         return holder;
    }

   private void checkForEmpty(String fieldName, String fieldKey,
                                String value, ActionErrors errors){
      //Same implementation as the PostStoryForm.
    }
```

```
    private void checkForVulgarities(String fieldName, String fieldKey,
                                     String value, ActionErrors errors){
      //Same implementation as the PostStoryForm.
    }

    private void checkForLength(String fieldName, String fieldKey,
                                String value, int maxLength,
                                ActionErrors errors){
      //Same implementation as the PostStoryForm.
    }

public ActionErrors validate(ActionMapping mapping,
                                HttpServletRequest request) {
        ActionErrors errors = new ActionErrors();

        checkForEmpty("Story Title", "error.storytitle.empty",
                      (String)getAttribute("storyTitle"),errors);
        checkForEmpty("Story Intro", "error.storyintro.empty",
                      (String)getAttribute("storyIntro"), errors);
        checkForEmpty("Story Body",  "error.storybody.empty",
                      (String)getAttribute("storyBody"), errors);

        checkForVulgarities("Story Title", "error.storytitle.vulgarity",
                      (String)getAttribute("storyTitle"), errors);
        checkForVulgarities("Story Intro", "error.storyintro.vulgarity",
                      (String)getAttribute("storyIntro"), errors);
        checkForVulgarities("Story Body",  "error.storybody.vulgarity",
                      (String)getAttribute("storyBody"), errors);

        checkForLength("Story Title", "error.storytitle.length",
                      (String)getAttribute("storyTitle"), 100, errors);
        checkForLength("Story Intro", "error.storyintro.length",
                      (String)getAttribute("storyIntro"), 2048, errors);
        checkForLength("Story Body", "error.storybody.length",
                      (String)getAttribute("storyBody"),  10000, errors);

        return errors;
    }

         public void reset(ActionMapping mapping,
                        HttpServletRequest request) {
       ActionServlet servlet =  this.getServlet();
       MessageResources messageResources = servlet.getInternal();
```

```
        setAttribute("storyTitle",
    messageResources.getMessage➥
            ("javaedge.poststory.title.instructions"));
        setAttribute("storyIntro", messageResources.getMessage➥
                        ("javaedge.poststory.intro.instructions"));
        setAttribute("storyBody" , messageResources.getMessage➥
                        ("javaedge.poststory.body.instructions"));
      }
}
```

The first thing you should notice about the `PostStoryMapForm` class is that there is no `get()` and `set()` for individual attributes. All attributes for the form bean are stored in a `HashMap` called `attributeMap`.

```
private HashMap attributeMap = new HashMap();
```

All access to the `attributeMap` variable is controlled by a pair of methods called `getAttribute()` and `setAttribute()`.

```
public void setAttribute(String attributeKey, Object attributeValue){
  getMap().put(attributeKey, attributeValue);
}

public Object getAttribute(String attributeKey){

  Object holder = getMap().get(attributeKey);

  if (holder==null) return "";

  return holder;
}
```

These methods do nothing more than provide an entry point for Struts to perform a retrieval and insertion of objects into the `attributeMap` variable.

NOTE *The method names* `getAttribute()` *and* `setAttribute()` *are arbitrary names. You can call your "wrapper" method to an internal* `HashMap` *anything you want as long as the method names follow standard JavaBean naming conventions and they have the same method signatures shown in the preceding code.*

Keep in mind that entry methods like `getAttribute()` and `setAttribute()` do not have to perform straight calls into the `Map` object. You can place code in the

entry methods to ensure that form attributes being retrieved out of and set in the internal `Map` object are formatted in a particular manner. In the `PostStoryMapForm` class, the `getAttribute()` call always returns an empty `String` object if the attribute from the `attributeMap` is `null`.

```
Object holder = getMap().get(attributeKey);

if (holder==null) return "";

return holder;
```

When data is accessed from the `PostStoryMapForm` class, the `getAttribute()` and `setAttribute()` methods are used. You can see this in the `PostStoryMapForm`'s `validate()` method.

```
public ActionErrors validate(ActionMapping mapping,
                                    HttpServletRequest request) {
  ActionErrors errors = new ActionErrors();

  checkForEmpty("Story Title", "error.storytitle.empty",
                      (String)getAttribute("storyTitle"),errors);
  checkForEmpty("Story Intro", "error.storyintro.empty",
                      (String)getAttribute("storyIntro"), errors);
  checkForEmpty("Story Body",  "error.storybody.empty",
                      (String)getAttribute("storyBody"), errors);

    . . .
  return errors;
}
```

At this point you might be wondering how the individual attributes in the `PostStoryMapForm` class are accessed in a JSP page. Before Struts 1.1, there was no way a Struts custom tag could directly access an element contained within an `Array` or `Map` object. Sure, you could always use a combination of `<logic:iterate>` and `<logic:equals>` tags to find a value, but you could never tell the Struts tag to directly access element X contained within a particular attribute on a form bean.

With the release of Struts 1.1, you can now do this. Code examples often speak volumes, so let's look at a rewritten version of the Post a Story page that uses the `PostStoryMapForm` class to retrieve and set form data. This new page, called postStoryMapContent.jsp, is shown here:

```
<%@ page language="java" %>
<%@ taglib uri="/taglibs/struts-bean" prefix="bean" %>
<%@ taglib uri="/taglibs/struts-html" prefix="html" %>
<%@ taglib uri="/taglibs/struts-logic" prefix="logic" %>
<%@ taglib uri="/taglibs/struts-tiles" prefix="tiles" %>
```

```
<BR/><BR/>
<H1><bean:message key="javaedge.poststory.text.header"/></H1>


<html:errors/>
<html:form action="postStory">

<TABLE>
   <TR>
      <TD>
        <bean:message key="javaedge.poststory.text.intro"/>

        <logic:present scope="session" name="memberVO">
          <B><bean:write name="memberVO" scope="session"
               property="firstName"/> <bean:write name="memberVO"
               scope="session" property="lastName"/></B><BR/>
        </logic:present>
        <logic:notPresent scope="session" name="memberVO">
          <B>Anonymous</B>
        </logic:notPresent>

      </TD>
      <TD>
        <BR/><BR/> 
      </TD>
   </TR>
   <TR>
     <TD>
       <font size="2"><b>
<bean:message
     key="javaedge.poststory.form.titlelabel"/>: 
</b></font><br/>
       <html:text property="attribute(storyTitle)"/>
     </TD>
   </TR>
   <TR>
     <TD>
        <font size="2"><b>
           <bean:message
                key="javaedge.poststory.form.introlabel"/>: 
           </b></font><br/>
       <html:textarea property="attribute(storyIntro)" cols="80" rows="5"/>
     </TD>
   </TR>
```

```
    <TR>
       <TD>
         <font size="2"><b>
            <bean:message
                key="javaedge.poststory.form.bodylabel"/>: 
            </b></font><br/>
         <html:textarea  property="attribute(storyBody)" cols="80" rows="10"/>
       </TD>
    </TR>
    <TR>
      <TD align="center">
        <html:submit property="submitButton" value="Submit"/>  
        <html:cancel value="Cancel"/>
      </TD>
    </TR>
</TABLE>
</html:form>
```

To access a mapped value in an `ActionForm`, you just need to use a slightly different nomenclature for the `property` attribute on the `<html:x>`, `<bean:x>`, or `<logic:x>` tags. Instead of just supplying the name of the property, you will also supply the key that you are looking to retrieve from the `Map` object. Figure 3-6 highlights this.

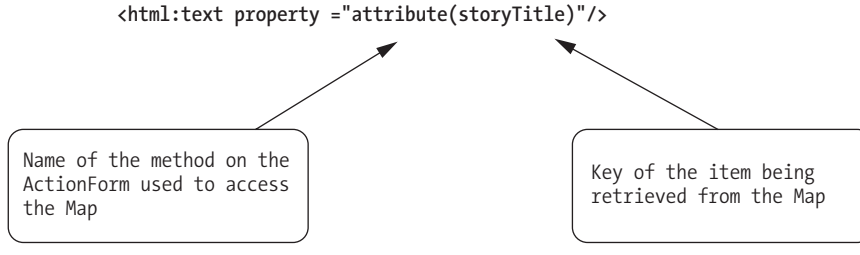

Figure 3-6. Anatomy of a `Map`*-backed* `ActionForm`

To actually see the `Map`-backed `ActionForm` work, you need to make the following modifications to the JavaEdge application:

- Add a new `<form-bean>` tag to the JavaEdge application's struts-config.xml file for the `PostStoryMapForm`.

- Modify the `/postStorySetup` and `/postStory` actions in the struts-config.xml file to use the newly created `<form-bean>`.

- Modify the PostStory.java file to cast to a `PostStoryMapForm` class instead of the `PostStoryForm` class. You also need to remove any references to `PostStoryForm`'s `get()/set()` method class and replace them with calls to `PostStoryMapForm`'s `getAttribute()` and `setAttribute()` methods.

- Modify the postStory.jsp to use the postStoryMapContent.jsp file instead of the postStoryContent.jsp file.

ActionForm Best Practices

`ActionForm` classes provide a very clean mechanism for abstracting the implementation details of getting and setting data in the `HttpServletRequest` object passed into the Struts `ActionServlet`. There are some best practices associated with using `ActionForm` classes. These best practices have evolved, as development teams using Struts have had to maintain and extend applications in a production environment.

Two of these best practices are documented here:

- Make all of the public attributes on your `ActionForm` of type `String`.

- Cleanly separate all of your `ActionForm` classes from your application's business logic. Do not pass `ActionForm` classes directly into your application's business logic.

Strings and the ActionForm

While a Struts `ActionForm` can expose any data type using a `get()/set()` method on the interface, it's a good idea to only use `Strings` as the data types being passed in and out of the `ActionForm`. The reason for this is that when a user submits form data to a Struts application, the Struts `ActionServlet` is going to pull all of the data out of the HTTP request and match that data to a `get()/set()` attribute on an `ActionForm`.

The problem arises when the user submits a piece of data in a form field that does not match the data type of its corresponding attribute on the `ActionForm`. This problem is encountered most often when dealing with numeric (that is, `Integer`, `Float`, etc.), `Date/Timestamps`, and `Boolean` data types. For example, suppose you expose an attribute on `ActionForm` with a `get()/set()` pair of methods that accept and return an `Integer` class. The user enters a value of `"a"` in the form field that is expecting an `Integer` object.

When this value is submitted, the Struts `ActionServlet` will try to set the value of `"a"` on an attribute that has been defined to be of type `Integer`. When this happens, an exception will be thrown and the user will usually end up with a big white screen full of informative Java error messages. You cannot catch this

problem in the `validate()` method on the `ActionForm`, because the data is copied out of the HTTP request before the `validate()` method is invoked.

How do you deal with this kind of problem? You could try to point JavaScript code to the HTML where the form data is being captured. The problem is that you can easily end up creating a Validation Confusion antipattern because you have effectively split your validation logic for the form into two different locations, and that is what you are trying to avoid. In addition, you lose a great deal of control when using JavaScript validation. End users can easily disable the JavaScript validation by configuring their web browser to not execute JavaScript code.

The best way to deal with this type of problem is keep all of the properties on your `ActionForm` class as type `String`. By doing this, you can then capture all of the data entered by the user without running the risk of a type mismatch. Then in the `ActionForm`'s `validate()` method you can perform type checking on the data contained within these strings and cleanly throw validation errors using Strut's `ActionError` objects.

ActionForms and Business Logic

The `ActionForm` class is used to hold data submitted by the end user. It is passed into the `execute()` method on an `Action` class, where its data can then be used by the action to carry out the business logic associated with the request. Ideally, business logic for the application should not be embedded inside of the Action class code itself.

Instead, the business logic for the application should be contained in a POJO or EJB that is completely independent of the `Action` class. Unfortunately, for many developers using Struts for the first time, the temptation to pass the `ActionForm` class from the `Action` class to the corresponding business logic is a very strong one. After all the `ActionForm` class is already holding the data submitted by the end user, so why not just pass it directly to the POJO or EJB containing the business logic?

The problem with this approach is that you are introducing the Tier Leakage antipattern into your application. You are letting an implementation detail from your presentation tier, the `ActionForm` class, be passed to your business tier. This creates a dependency on a Struts-specific class, with an unintended consequence if you want to reuse that piece of business logic outside of a Struts-based application: You have to refactor the code to not have this dependency or instantiate a Struts `ActionForm` class and populate it with data, even if you are not building a web-based application.

To avoid this problem, you should copy the data contained within your `ActionForm` class to a framework-independent class called a value object. The concept of a value object is covered in greater detail in Chapter 5. For now, think of a value object as being nothing more than a class that holds data.

There are two ways you can copy data from an ActionForm. The first mechanism is to brute force the copy and set the attributes on a value object by calling the individual get() methods on the ActionForm class. For example, in the PostStory.execute() method, you need to copy data from the ActionForm to the StoryVO object.

```
public ActionForward execute(ActionMapping mapping,
                                ActionForm     form,
                                HttpServletRequest request,
                                HttpServletResponse response){



   PostStoryForm postStoryForm = (PostStoryForm) form;

   HttpSession session = request.getSession();

   MemberVO  memberVO      = (MemberVO) session.getAttribute("memberVO");
   storyVO.setStoryIntro(postStoryForm.getStoryIntro());
   storyVO.setStoryTitle(postStoryForm.getStoryTitle());
   storyVO.setStoryBody(postStoryForm.getStoryBody());

   StoryVO storyVO = new StoryVO();

   storyVO.setStoryAuthor(memberVO);
   storyVO.setSubmissionDate(new java.sql.Date(System.currentTimeMillis()));
   storyVO.setComments(new Vector());

   //Rest of the code
   . . . . .
}
```

As you can guess, on an ActionForm with a lot of data, you can end up cluttering your Action class with a lot of code that does nothing more than copy data from the ActionForm to the value object.

The Struts framework does provide some utility classes that enable you to more quickly copy data from the ActionForm to the value object. These two classes are part of the Apache Commons BeanUtils project (http://jakarta.apache.org/commons). These classes are distributed with Struts in the commons-beanutils.jar file. The classes are

- org.apache.commons.beanutils.BeanUtils
- org.apache.commons.beanutils.PropertyUtils

Both of these classes simplify many of the most common tasks associated with manipulating a JavaBean. In the code example, we are going to show you how to use the `copyProperties()` method on the `BeanUtils` class to copy data from `postStoryForm` to `storyVO`.

```
public ActionForward execute(ActionMapping mapping,
                                  ActionForm     form,
                                  HttpServletRequest request,
                                  HttpServletResponse response){



   PostStoryForm postStoryForm = (PostStoryForm) form;

   HttpSession session = request.getSession();

   MemberVO  memberVO       = (MemberVO) session.getAttribute("memberVO");
   StoryVO storyVO = new StoryVO();


   try{
     BeanUtils.copyProperties(storyVO, postStoryForm);
   }
   catch(IllegalAccessException e) {
     throw new ApplicationException("IllegalAccessException " +
       " in PostStory.execute",e);
   }
   catch(InvocationTargetException e){
     throw new ApplicationException(
      "InvocationTargetException in PostStory.execute",
      e);
   }

   storyVO.setStoryAuthor(memberVO);
   storyVO.setSubmissionDate(new java.sql.Date(System.currentTimeMillis()));
   storyVO.setComments(new Vector());

   //Rest of the code
   . . . . .
}
```

When the `copyProperties()` method is invoked, it will use Java reflection to call each of the `get()` methods on the `postStoryForm` object.

```
try{
  BeanUtils.copyProperties(storyVO, postStoryForm);
}
catch(IllegalAccessException e) {
  throw new ApplicationException("IllegalAccessException in
                                  PostStory.execute",e);
}
catch(InvocationTargetException e){
  throw new ApplicationException("InvocationTargetException in
                                  PostStory.execute",e);
}
```

When `BeanUtils.copyProperties()` is called, it will invoke each of the `get()` methods on the object passed in as the first parameter. As `copyProperties()` calls each `get()` method, it will try to call a corresponding `set()` method that has the same name on the object passed as the second parameter.

The `BeanUtils.copyProperties()` can be a significant time saver when you are trying to copy data from an `ActionForm` to a value object. However, it still litters up the `Action` classes `execute()` method with exception-handling code.

NOTE *The* `BeanUtils.copyProperties()` *method will try to do type conversions between properties being copied from one JavaBean to another.*

This means that if an application is trying to use the `copyProperties()` *method to copy a property defined as type* `Integer` *to a property of type* `String` *on another bean, the* `copyProperties()` *method will attempt to do a type conversion. If the* `copyProperties()` *method cannot do the type conversion, it will throw an exception of type* `java.lang.reflect.InvocationTargetException`.

If you know that there are going to be no type conversions when copying the contents of one JavaBean to another JavaBean, you can use the `PropertyUtils.copyProperties()` *method. This* `copyProperties()` *method on* `PropertyUtils` *does the same function as the corresponding method on* `BeanUtils`*, but does not attempt to do type conversion of properties.*

As you will see later on in Chapter 4, one of the design goals of a Struts application should be to minimize the amount of code present in the `Action` class.

What if you were to encapsulate all of the logic for copying data to and from a value object inside of the `ActionForm` class itself? This way, the `Action` class would just need to invoke a method on the `ActionForm` to get a value object that would be passed to `Action`'s business logic. The concept of building a value object factory method into `ActionForm` is not new and was first documented in the book *Struts in Action* (Ted Husted et al., Manning Press, ISBN: 1-930-11050-2).

Let's add a new method to the `PostStoryForm` class called `buildStoryVO`.

```
public StoryVO buildStoryVO(HttpServletRequest request)
  throws ApplicationException{
  HttpSession session = request.getSession();

  MemberVO       memberVO       = (MemberVO) session.getAttribute("memberVO");
  StoryVO storyVO = new StoryVO();

  /*Example of how to use the BeanUtils class to populate a valueobject.*/
  try{
    BeanUtils.copyProperties(storyVO, this);
  }
  catch(IllegalAccessException e) {
    throw new ApplicationException("IllegalAccessException in
                                    PostStoryForm.execute",e);
  }
  catch(InvocationTargetException e){
    throw new ApplicationException("InvocationTargetException in
                                    PostStoryForm.execute",e);
  }

  storyVO.setStoryAuthor(memberVO);
  storyVO.setSubmissionDate(new java.sql.Date(System.currentTimeMillis()));
  storyVO.setComments(new Vector());

  return storyVO;
}
```

The `buildStoryVO()` method on the `PostStoryForm` class cleanly encapsulates all of the details associated with building out a `StoryVO` based on the data contained within the `PostStoryForm`. When this method is used in the `execute()` method on the `PostStory` class, you end up with a much cleaner method:

```
public ActionForward execute(ActionMapping mapping,
                                ActionForm     form,
                                HttpServletRequest request,
                                HttpServletResponse response)
            throws ApplicationException {

  if (this.isCancelled(request)){
    return (mapping.findForward("poststory.success"));
  }

  PostStoryForm postStoryForm = (PostStoryForm) form;
  StoryVO storyVO = postStoryForm.buildStoryVO(request);
```

```
    StoryManagerBD storyManager = new StoryManagerBD();
    storyManager.addStory(storyVO);
    return (mapping.findForward("poststory.success"));
}
```

You may have noticed that the `execute()` method just shown throws an `ApplicationException`. However, nowhere in the code is a `try{}/catch{}` block to handle the exception. So, how does the JavaEdge application deal with the `ApplicationException` if the exception is raised?

The JavaEdge application uses the new Struts 1.1 exception handler functionality to process the `ApplicationException`. Struts exception handlers allow a development team to declare how an `Exception` is to be thrown without cluttering up the `Action` class. This new exception-handling functionality is described in greater detail in the next chapter.

Summary

This chapter focuses on how to use Struts to collect and process the data submitted in an HTML form. The following four pieces must be present to use the Struts-based form processing:

- `ActionForm` class
- `Action` class
- JSP page that uses the Struts HTML tag library to generate the HTML `<input>` fields used to collect the user information
- `name` attribute in an `<action>` tag in the struts-config.xml file

The `ActionForm` class acts as a wrapper class for the form data submitted by the user. The `ActionServlet` will use the `ActionForm` class, defined for an action, to pull the submitted form data out of the user's HTTP request. Each piece of data sent from a form will correspond to a `get()` or `set()` method that will be used to retrieve and populate the `ActionForm`. The `ActionForm` class has two methods that can be overridden by the developer, `reset()` and `validate()`. The `reset()` method is used to clear the properties in `ActionForm` to ensure that they are always in a predetermined state. To prepopulate a form with data, you can use the `reset()` method.

The `validate()` method in the `ActionForm` class will contain any validation rules that need to be applied against the submitted data. This method should contain only the lightweight validation rules that check the constraints on the data. If any validation errors are found, an `ActionError` class will be added to the `ActionErrors` class, which is passed back by the `validate()` method. If the

ActionServlet finds that ActionErrors contains errors, it will redirect the users back to the JSP page where they submitted the data. The validation errors will then be displayed using the <html:errors> tag.

The Action class is used to process the user's request. Its execute() method holds all the business logic used to process the user's request. We did not discuss the Action class in detail in this chapter. We are going to explore the business logic handling in the next chapter.

To map the data submitted in the HTML form to the ActionForm class, you need to use a JSP page that uses the Struts HTML tag library. The Struts HTML tag library contains a number of tags used for rendering HTML input tags. Some of the tags that were covered in this chapter include

- Tags used for building the base HTML form:
  - <html:form>: Used to render a <form> tag
  - <html:submit>: Renders a submit button
  - <html:cancel>: Renders a cancel button
  - <html:errors>: Renders any validation errors that have been raised during processing.
- Tags used for entering information:
  - <html:text>: Renders a text field
  - <html:textarea>: Renders a textarea field
  - <html:select>: Renders a drop-down box
  - <html:option>: Renders one selection in a drop-down box
  - <html:checkbox>: Renders a checkbox control
  - <html:radio>: Renders a radio button control

These three pieces (ActionForm class, Action class, and JSP page) together are all associated with an <action> tag in the struts-config.xml file. Each ActionForm used in the application must be declared as a <form-bean> tag. Once it is declared, the ActionForm can be associated with an <action> tag. The name attribute of the <action> tag tells the ActionServlet class processing a user request that there is an ActionForm class that will be used to collect the data. If the <action> tag contains a validate attribute set to true, the ActionServlet will invoke the validate() method in the ActionForm class.

Finally, we showed you two best practices associated with building Struts `ActionForm` classes:

- Making all of the public attributes on your `ActionForm` string variables
- Cleanly separating all of your application's `ActionForm` classes from your application's business logic

In the next chapter, you are going to see how to build the business logic in Struts. The Struts development framework, with the help of the `Action` class, provides a clean mechanism for encapsulating the business logic. However, if you are not careful, you can limit the long-term reusability of your application code. The next chapter demonstrates how the use of several core design patterns, when coupled with the Struts, provides optimal code reusability.