The View Layer

In an MVC application, the View layer provides an interface to your application, be it for users with a browser or for another application using something like Web services. Basically the View layer is the conduit for getting data in and out of the application. It does not contain business logic, such as calculating interest for a banking application or storing items in a shopping cart for an online catalog. The View layer also does not contain any code for persisting data to or retrieving data from a data source. Rather, it is the Model layer that manages business logic and data access. The View layer simply concentrates on the interface.

Keeping the Model and View layers separate from one another allows an application's interface to change independent of the Model layer and vice versa. This separation also allows the application to have multiple interfaces (or views). For instance, an application could have a Web interface and a wireless interface. In this case, each interface is separate, but both use the same Model layer code without the Model layer being tied to either interface or either interface having to know about the other interface.

Struts and the View Layer

Struts provides a rich set of functionality and features for developing the View layer of MVC applications. There are several forms that the View layer of a Struts application can take. It can be HTML/JSP (the most common case) or it can be XML/XSLT, Velocity, Swing, or whatever your application requires. This is the power of Struts and MVC. Because HTML/JSP is the typical View technology used for Java-based Web applications, Struts provides the most functionality and features for developing your application this way. The remainder of this chapter focuses on Struts' support for creating the View layer using HTML/JSP.

Struts' HTML/JSP View layer support can be broken down into the following major components:

- JSP pages
- Form Beans
- JSP tag libraries
- Resource bundles

Each of these components is examined in detail in this chapter, but first it is helpful to understand how they fit together in the View layer.

JSP pages are at the center of the View components; they contain the HTML that is sent to browsers for users to see and they contain JSP library tags. The library tags are used to retrieve data from Form Beans and to generate HTML forms that, when submitted, will populate Form Beans. Additionally, library tags are used to retrieve content from resource bundles. Together, all of Struts' View layer components are used to generate HTML that browsers render. This is what the user sees.

On the back side, the View layer populates Form Beans with data coming from the HTML interface. The Controller layer then takes the Form Beans and manages getting their data and putting it into the Model layer. Additionally, the Controller layer takes data from the Model layer and populates Form Beans so that the data can be presented in the View layer.

The following sections explain each of these major View components in detail.

JSP Pages

JSPs are the centerpiece of the Struts View layer. They contain the static HTML and JSP library tags that generate dynamic HTML. Together the static and dynamically generated HTML gets sent to the user's browser for rendering. That is, the JSPs contain the code for the user interface with which a user interacts.

JSPs in Struts applications are like JSPs in any other Java-based Web application. However, to adhere to the MVC paradigm, the JSPs should not contain any code for performing business logic or code for directly accessing data sources. Instead, the JSPs are intended to be used solely for displaying data and capturing data. Struts provides a set of tag libraries that supports displaying data and creating HTML forms that capture data. Additionally, the tags support displaying content stored in resource bundles. Therefore, JSPs (coupled with Form Beans) provide the bulk of the Struts View layer. The JSP tag libraries glue those two together and the resource bundles provide a means of content management.

Form Beans

Form Beans provide the conduit for transferring data between the View and Controller layers of Struts applications. When HTML forms are submitted to a Struts application, Struts takes the incoming form data and uses it to populate the form's corresponding Form Bean. The Struts Controller layer then uses the Form Beans to access data that must be sent to the Model layer. On the flip side, the Controller layer populates Form Beans with Model layer data so that it can be displayed with the View layer. Essentially, Form Beans are simple data containers. They either contain data from an HTML form that is headed to the Model via the Controller or contain data from the Model headed to the View via the Controller.

Form Beans are basic Java beans with getter and setter methods for each of their properties, allowing their data to be set and retrieved easily. The **org.apache.struts .action.ActionForm** class is the base abstract class that all Form Beans must subclass. Because Form Beans are simple data containers, they are principally comprised of fields, and setter and getter methods for those fields. Business logic and data access code should not be placed in these classes. That code goes in Model layer classes. The only other methods that should be in these classes are helper methods or methods that override **ActionForm**'s base **reset( )** and **validate( )** methods.

The **ActionForm** class has a **reset( )** method and a **validate( )** method that are intended to be overridden by subclasses where necessary. The **reset( )** method is a hook that Struts calls before the Form Bean is populated from an HTML form submission. The **validate( )** method is a hook that Struts calls after the Form Bean has been populated from an HTML form submission. Both of these methods are described in detail later in this section.

Following is an example Form Bean:

```
import org.apache.struts.action.ActionForm; public class EmployeeForm extends ActionForm {  private String firstName;  private String lastName;  private String department;  public void setFirstName(String firstName) {  this.firstName = firstName;  }  public String getFirstName() {  return firstName;  }  public void setLastName(String lastName) {  this.lastName = lastName;  }  public String getLastName() {  return lastName;  }  public void setDepartment(String department) {  this.department = department;  }  public String getDepartment() {  return department;  } }
```

Form Bean properties can be of any object type, be it a built-in class like **String** or a complex application-specific class such as an **Address** object that has fields for street address, city, state, and ZIP. Struts uses reflection to populate the Form Beans and can traverse object hierarchies to any level so long as the getter and setter methods are public. For example, if your Form Bean had an **Address** object field named **address**, to access the **city** field on the **Address** object the Form Bean would need a public **getAddress( )** method that returned an **Address** object. The **Address** object would need a public **getCity( )** method that would return a **String**.

Often, it's best to have Form Bean fields be **String**s instead of other types. For example, instead of having an **Integer**-type field for storing a number, it's best to use a **String**-type field. This is because all HTML form data comes in the form of strings. If a letter rather than a number is entered in a numeric field, it's better to store the value in a **String** so that the original data can be returned to the form for correcting. If instead the data is stored in a **Long**, when Struts attempts to convert the string value to a number, it will throw a **NumberFormatException** if the value is a letter. Then, when the form is redisplayed showing the invalid data, it will show 0 instead of the originally entered value, because letters cannot be stored in numeric-type fields.

Configuring Form Beans

To use Form Beans, you have to configure them in the Struts configuration file. Following is a basic Form Bean definition:

<!-- Form Beans Configuration --> <form-beans> <form-bean name="searchForm" type="com.jamesholmes.minihr.SearchForm"/> </form-beans>
Form Bean definitions specify a logical name and the class type for a Form Bean. Once defined, Form Beans are associated with actions by action mapping definitions, as shown next:

<!-- Action Mappings Configuration --> <action-mappings> <action path="/search" type="com.jamesholmes.minihr.SearchAction" **name="searchForm"** scope="request" validate="true" input="/search.jsp"> </action> </action-mappings>

Actions specify their associated Form Bean with the **name** attribute of the **action** tag, as shown in the preceding snippet. The value specified for the **name** attribute is the logical name of a Form Bean defined with the **form-bean** tag. The **action** tag also has a **scope** attribute to specify the scope that the Form Bean will be stored in and a **validate** attribute to specify whether or not the Form Bean's **validate( )** method should be invoked after the Form Bean is populated.

The reset( ) Method

As previously stated, the abstract **ActionForm** class has a **reset( )** method that subclasses can override. The **reset( )** method is a hook that gets called before a Form Bean is populated with request data from an HTML form. This method hook was designed to account for a shortcoming in the way browsers handle check boxes. Browsers send the value of a check box only if it is checked when the HTML form is submitted. For example, consider an HTML form with a check box for whether or not a file is read-only:

<input type="checkbox" name="readonly" value="true">
When the form containing this check box is submitted, the value of "true" is sent to the server only if the check box is checked. If the check box is not checked, no value is sent.

For most cases, this behavior is fine; however, it is problematic when Form Bean **boolean** properties have a default value of "true." For example, consider the read-only file scenario again. If your application has a Form Bean with a read-only property set to true and the Form Bean is used to populate a form with default settings, the read-only property will set the read-only check box's state to checked when it is rendered. If a user decides to uncheck the check box and then submits the form, no value will be sent to the server to indicate that the check box has been unchecked (i.e., set to false). By using the **reset( )** method, this can be solved by setting all properties tied to check boxes to false before the Form Bean is populated. Following is an example implementation of a Form Bean with a **reset( )** method that accounts for unchecked check boxes:

import org.apache.struts.action.ActionForm; public class FileForm extends ActionForm { private boolean readOnly; public void setReadOnly(boolean readOnly) { this.readOnly = readOnly; } public boolean getReadOnly() { return readOnly; } public void reset() { readOnly = false; } }

The **reset( )** method in this example class ensures that the **readOnly** property is set to false before the form is populated. Having the **reset( )** method hook is equivalent to having the HTML form actually send a value for unchecked check boxes.

A side benefit of the **reset( )** method hook is that it offers a convenient place to reset data between requests when using Form Beans that are stored in session scope. When Form Beans are stored in session scope, they persist across multiple requests. This solution is most often used for wizard-style process flows. Sometimes it's necessary to reset data between requests, and the **reset( )** method provides a convenient hook for doing this.

The validate( ) Method

In addition to the **reset( )** method hook, the **ActionForm** class provides a **validate( )** method hook that can be overridden by subclasses to perform validations on incoming form data. The **validate( )** method hook gets called after a Form Bean has been populated with incoming form data. Following is the method signature for the **validate( )** method:

public ActionErrors validate(ActionMapping mapping, HttpServletRequest request)

Notice that the **validate( )** method has a return type of **ActionErrors**.

The **org.apache.struts.action.ActionErrors** class is a Struts class that is used for storing validation errors that have occurred in the **validate( )** method. If all validations in the **validate( )** method pass, a return value of null indicates to Struts that no errors occurred.

Note Form Bean data validations could be performed in action classes; however, having them in Form Beans allows them to be reused across multiple actions where more than one action uses the same Form Bean. Having the validation code in each action would be redundant.

Following is an example Form Bean with a **validate( )** method:

import javax.servlet.http.HttpServletRequest; import org.apache.struts.action.ActionError; import org.apache.struts.action.ActionErrors; import org.apache.struts.action.ActionForm; import org.apache.struts.action.ActionMapping; public class NameForm extends ActionForm { private String name; public void setName(String name) { this.name = name; } public String getName() { return name; } public ActionErrors

```
validate(ActionMapping mapping,  HttpServletRequest request)  {  if (name == null ||
name.length() < 1) {  ActionErrors errors = new
ActionErrors();  errors.add("name",  new
ActionError("error.name.required"));  return errors;  }  return null;  } }
```

This example Form Bean has one field, **name**, that is validated in the **validate( )** method. The **validate( )** method checks whether or not the **name** field is empty. If it is empty, it returns an error indicating that fact. The **ActionErrors** object is basically a collection class for storing **org.apache.struts.action.ActionError** instances. Each validation inside the **validate( )** method creates an **ActionError** instance that gets stored in the **ActionErrors** object. The **ActionError** class takes a key to an error message stored in the Struts resource bundle. Struts uses the key to look up the corresponding error message. The **ActionError** class also has constructors that take additional arguments that contain replacement values for the error message associated with the specified key.

Struts also has a built-in Validator framework that greatly simplifies performing data validations. The Validator framework allows you to declaratively configure in an XML file the validations that should be applied to Form Beans. For more information on the Validator framework, see Chapter 6.

Note The **validate( )** method will be called only if the Form Bean has been configured for validation in the Struts configuration file.

The Lifecycle of Form Beans

Form Beans have a defined lifecycle in Struts applications. To fully understand how Form Beans work, it’s necessary to understand this lifecycle. The Form Bean lifecycle is shown in Figure 4-1.

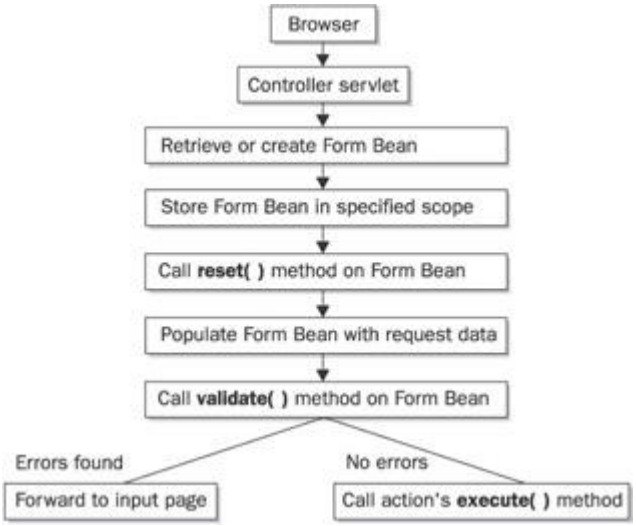

Figure 4-1: The Form Bean lifecycle

Following is an explanation of the Form Bean lifecycle. When a request is received by the Struts controller servlet, Struts maps the request to an action class that is delegated to process the request. If the action being delegated to has an associated Form Bean, Struts attempts to look up the specified Form Bean in request or session scope, based on how the action is configured in the Struts configuration file. If an instance of the Form Bean is not found in the specified scope, an instance is created and placed in the specified scope. Next, Struts calls the **reset( )** method on the Form Bean so that any processing is executed that needs to occur before the Form Bean is populated. After that, Struts populates the Form Bean with data from the incoming request. Next, the Form Bean's **validate( )** method is called. The next step in the process is based on the return value from the **validate( )** method. If the **validate( )** method records any errors and subsequently returns a non-null **ActionErrors** object, Struts forwards back to the action's input page. If, however, the return value from the **validate( )** method is null, Struts continues processing the request by calling the action's **execute( )** method.

DynaActionForms

A useful addition to the 1.1 release of Struts was the introduction of Dynamic Form Beans. Dynamic Form Beans are an extension of Form Beans that allows you to specify their properties inside the Struts configuration file instead of having to create a concrete class, with a getter and setter method for each property. The concept of Dynamic Form Beans originated because many developers found it tedious to create for every page a Form Bean that had a getter and setter method for each of the fields on the page's HTML form. Using Dynamic

Form Beans allows the properties to be specified in a Struts configuration file. To change a property, simply update the configuration file. No code has to be recompiled.

The following snippet illustrates how Dynamic Form Beans are configured in the Struts configuration file:

<!-- Form Beans Configuration --> <form-beans> <form-bean name="employeeForm" type="org.apache.struts.action.DynaActionForm"> <form -property name="firstName" type="java.lang.String"/> <form-property name="lastName" type="java.lang.String"/> <form-property name="department" type="java.lang.String"/> </form-bean> </form-beans>
Dynamic Form Beans are declared in the same way as standard Form Beans, by using the **form-bean** tag. The difference is that the type of the Form Bean specified with the **form-bean** tag's **type** attribute must be **org.apache.struts.action.DynaActionForm** or a subclass thereof. Additionally, the properties for Dynamic Form Beans are specified by nesting **form-property** tags beneath the **form-bean** tag. Each property specifies its name and class type. Additionally, an initial value for the property can be specified using the **form-property** tag's **initial** attribute, as shown next:

<form-property name="department" type="java.lang.String" initial="Engineering"/>
If an initial value is not supplied for a property, Struts sets the initial value using Java's initialization conventions. That is, numbers are set to zero and objects are set to null.

Because you declare Dynamic Form Beans in the Struts configuration file instead of creating concrete classes that extend **ActionForm**, you do not define **reset( )** or **validate( )** methods for the Dynamic Form Beans. The **reset( )** method is no longer necessary for setting default values because the **initial** attribute on the **form-property** tag achieves the same effect. The **DynaActionForm** class's implementation of the **reset( )** method resets all properties to their initial value when it is called. You can either code the functionality of the **validate( )** method inside action classes or use the Validator framework for validation. These two options eliminate the need to create a **validate( )** method on the Form Bean. If, however, you have a special case where you need to have an implementation of the **reset( )** and/or **validate( )** method for your Dynamic Form Bean, you can subclass **DynaActionForm** and create the

methods there. Simply specify your **DynaActionForm** subclass as the type of the Form Bean in the Struts configuration file to use it.

JSP Tag Libraries

Struts comes packaged with a set of its own custom JSP tag libraries that aids in the development of JSPs. The tag libraries are fundamental building blocks in Struts applications because they provide a convenient mechanism for creating HTML forms whose data will be captured in Form Beans and for displaying data stored in Form Beans. Additionally, Struts' tag libraries provide several utility tags to accomplish things such as conditional logic, iterating over collections, and so on. With the advent of the JSP Standard Tag Library (JSTL), many of the utility tags have been superceded. (Using JSTL with Struts is covered in Chapter 15.)

Following is a list of the Struts tag libraries and their purpose:

- **HTML** Used to generate HTML forms that interact with the Struts APIs.
- **Bean** Used to work with Java bean objects in JSPs, such as to access bean values.
- **Logic** Used to cleanly implement simple conditional logic in JSPs.
- **Nested** Used to allow arbitrary levels of nesting of the HTML, Bean, and Logic tags that otherwise do not work.

Later in this book, each of these libraries has an entire chapter dedicated to its use, but this section provides a brief introduction to using the tag libraries, focusing on Struts' core JSP tag library, the HTML Tag Library, as an example. This library is used to generate HTML forms that, when submitted, populate Form Beans. Additionally, the HTML Tag Library tags can create HTML forms populated with data from Form Beans. To use the HTML Tag Library in a Struts application, you need to include a snippet like the following in your application's Web Archive (**.war**) deployment descriptor, **web.xml**:

<taglib> <taglib-uri>/WEB-INF/tlds/struts-html.tld</taglib-uri> <taglib-location>/WEB-INF/tlds/struts-html.tld</taglib-location> </taglib>

This snippet sets up the HTML Tag Library. For information on setting up the other tag libraries, see their respective chapters.

Recall from the overview of the **web.xml** file in Chapter 2 that the <**taglib-uri**> tag is used to declare the URI (or alias) that will be referenced in each of your JSPs with a **taglib** directive. The <**taglib-location**> tag is used to declare the actual location of the Tag Library Descriptor (**.tld**) file in your Web Archive.

The following snippet illustrates how your JSPs declare their use of the HTML Tag Library with a JSP **taglib** directive:

<%@ taglib uri="/WEB-INF/tlds/struts-html.tld" prefix="html" %>
Notice that the **uri** attribute specified here is the same as that declared with the <**taglib-uri**> tag in the **web.xml** file. Also, notice that the **prefix** attribute is set to **html**. This attribute can be set to whatever you want; however, **html** is the accepted default for the HTML Tag Library. The **prefix** attribute declares the prefix that each tag must have when it is used in the JSP, as shown here:

<**html:form** action="/Search">
Because **html** is defined as the prefix, the **form** tag is used as shown. However, if you were to choose to use a prefix of **struts-html**, the tag would be used as follows:

<**struts-html:insert** attribute="header"/>