Pro Apache Struts with Ajax

John Carnell
with Rob Harrop,
Edited by Kunal Mittal

Apress®

Pro Apache Struts with Ajax

Copyright © 2006 by John Carnell, Rob Harrop, Kunal Mittal

ISBN-13 (pbk): 978-1-59059-738-5

ISBN-10 (pbk): 1-59059-738-9

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewer: John Fallows
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Managers: Beth Christmas, Elizabeth Seymour
Copy Edit Manager: Nicole Flores
Copy Editors: Ami Knox, Bill McManus
Assistant Production Director: Kari Brooks-Copony
Production Editor: Lori Bring
Compositor: Diana Van Winkle, Van Winkle Design
Proofreader: April Eddy
Indexer: Michael Brinkman
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

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, e-mail `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 Source Code/Download section.

CHAPTER 2

Struts Fundamentals

Building a web-based application can be one of the most challenging tasks for a development team. Web-based applications often encompass functionality and data pulled from multiple IT systems. Most of the time, these systems are built on a variety of heterogeneous software and hardware platforms. Hence, the question that the team always faces is, how do we build web applications that are extensible and maintainable, even as they get more complex?

Most development teams attack the complexity by breaking the application into small, manageable parts that can communicate with one another via well-defined interfaces. Generally, this is done by breaking the application logic into three basic tiers: the presentation tier, business logic tier, and data access tier. By layering the code into these three tiers, the developers isolate any changes made in one tier from the other application tiers. However, simply grouping the application logic into three categories is not enough for medium to large projects. When coordinating a web-based project of any significant size, the application architect for the project must ensure that all the developers write their individual pieces to a standard framework that their code will "plug" into. If they do not, the code base for the application will be in absolute chaos, because multiple developers will implement their own pieces using their own development style and design.

The solution is to use a generalized development framework that has specific plug-in points for each of the major pieces of the application. However, building an application development framework from the ground up entails a significant amount of work. It also commits the development team to build and support the framework. Framework support forces the development team to exhaust resources that could otherwise be used for building applications.

The next three chapters of this book introduce you to a readily available alternative for building your own web application development framework, the *Apache Struts* development framework. These chapters do not cover every minute detail associated with the Struts development framework; instead, they are a guide on how to use Struts to build the JavaEdge application, introduced in Chapter 1.

This chapter is going to focus on installing the Struts framework, configuring it, and building the first screen in the JavaEdge application. We cover the following topics in this chapter:

- A brief history of the Struts development framework.
- A Struts-based application walkthrough.
- Setting up your first Struts project, including the physical layout of the project, and an explanation of all the important Struts configuration files.

- Configuring a Struts application. Some of the specific configuration issues that will be dealt with here include
    - Configuring the Struts `ActionServlet`
    - Configuring Struts actions in the struts-config.xml file
- Best practices for Struts configuration

In addition to our brief Struts configuration tutorial, we are going to discuss how Struts can be used to build a flexible and dynamic user interface. We will touch briefly on some, but not all, of the custom JSP tag libraries available to the Struts developer. Some of the things you can do with tag libraries that will be covered in this chapter include

- Manipulating JavaBeans by using the Struts Bean tag library
- Making JSP pages dynamic by leveraging the conditional and iterating power of the Struts Logic tag library

Let's begin our discussion with some of the common problems faced while building an application.

The JavaEdge Application Architecture

The JavaEdge application, which we are going to show you how to develop, is a very simple weblog (that is, a blog) that allows the end users to post their stories and comment on the other stories. We have already discussed the requirements of the JavaEdge application in Chapter 1 in the section called "The JavaEdge Application." The application is going to be written completely in Java. In addition, all the technologies used to build this application will be based on technology made available by the Apache Software Foundation.

In this section, we'll focus on some of the architectural requirements needed to make this application extensible and maintainable. This application is built by multiple developers. To enforce consistency and promote code reuse, we will use an application development framework that provides plug-in points for the developers to add their individual screens and elements.

The framework used should alleviate the need for the JavaEdge developers to implement the infrastructure code normally associated with building an application. Specifically, the development framework should provide

- *A set of standard interfaces for plugging the business logic into the application*: A developer should be able to add and modify new pieces of functionality using the framework while keeping the overall application intact (that is, a small change in the business logic should not require major updates in any part of the application).
- *A consistent mechanism for performing tasks*: This includes tasks such as end-user data validation, screen navigation, and invocation of the business logic. None of these tasks should be hard coded into the application source code. Instead, they should be implemented in a declarative fashion that allows easy reconfiguration of the application.
- *A set of utility classes or custom JSP tag libraries that simplify the process in which the developer builds new applications and screens*: Commonly repeated development tasks, such as manipulating the data in a JavaBean, should be the responsibility of the framework and not the individual developer.

The chosen development framework must provide the scaffolding in which the application is to be built. Without this scaffolding, antipatterns such as Tier Leakage and Hardwired will manifest themselves. We will demonstrate how Struts can be used to refactor these antipatterns in this chapter. Now, let's start the discussion on the architectural design of the JavaEdge application.

The Design

We will use a Model-View-Controller (MVC) pattern as the basis for the JavaEdge application architecture. The three core components of the MVC pattern, also known as a Model-2 JSP pattern by Sun Microsystems, are shown in Figure 2-1.

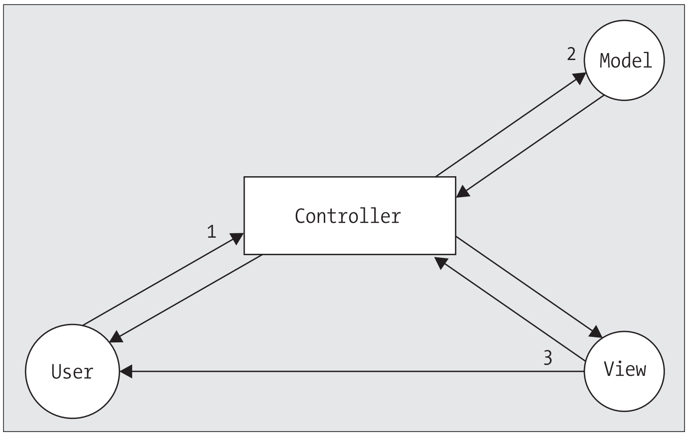

Figure 2-1. *A Model-View-Controller pattern*

The numbers shown in the diagram represent the flow in which a user's request is processed. When a user makes a request to an MVC-based application, it is always intercepted by the controller (step 1). The controller acts as a traffic cop, examining the user's request and then invoking the business logic necessary to carry out the requested action.

The business logic for a user request is encapsulated in the model (step 2). The model executes the business logic and returns the execution control back to the controller. Any data to be displayed to the end user will be returned by the model via a standard interface.

The controller will then look up, via some metadata repository, how the data returned from the model is to be displayed to the end user. The code responsible for formatting the data, to be displayed to the end user, is called the *view* (step 3). Views contain only the presentation logic and no business logic. When the view completes formatting the output data returned from the model, it will return execution control to the controller. The controller, in turn, will return control to the end user who made the call.

The MVC pattern is a powerful model for building applications. The code for each screen in the application consists of a model and a view. Neither of these components has explicit knowledge of the other's existence. These two pieces are decoupled via the controller, which acts as intermediary between these two components. At runtime, the controller assembles the required business logic and the view associated with a particular user request. This clean decoupling of the business and presentation logic allows the development team to build a pluggable architecture. As a result, new functionality and methods to format end-user data can easily be written while minimizing the chance of any changes disrupting the rest of the application.

New functionality can be introduced into the application by writing a model and view and then registering these items to the controller of the application. Let's assume that you have a web application whose view components are JSP pages generating HTML. If you want to rewrite this application for a mobile device, or in something like Swing instead of standard web-based HTML for users' requests, you would only need to modify the view of the application. The changes you make to the view implementation will not have an impact on the other pieces of the application. At least in theory!

In a Java-based web application, the technology used to implement an MVC framework might look as shown in Figure 2-2.

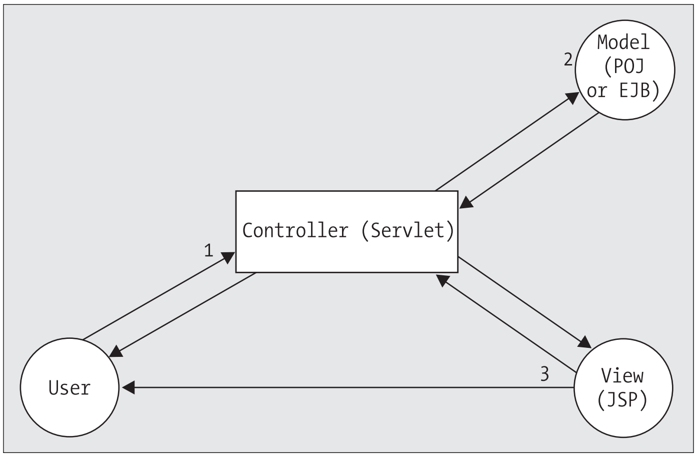

Figure 2-2. *The Java technologies used in an MVC*

An MVC-based framework offers a very flexible mechanism for building web-based applications. However, building a robust MVC framework infrastructure requires a significant amount of time and energy from your development team. It would be better if you could leverage an already existing implementation of an MVC framework. Fortunately, the Struts development framework is a full-blown implementation of the MVC pattern.

In the next section, we are going to walk through the major components of the Struts architecture. While Struts has a wide variety of functionalities available in it, it is still in its most basic form, which is an implementation of an MVC pattern.

Using Struts to Implement the MVC Pattern

The Struts development framework (and many of the other open source tools used in this book) is developed and managed by the Apache Software Foundation (ASF). The ASF has its roots in the Apache Group. The Apache Group was a loose confederation of open source developers who, in 1995, came together and wrote the Apache Web Server. (The Apache Web Server is the most popular web server in use and runs over half of the web applications throughout the world.) Realizing that the group needed a more formalized and legal standing for protecting their open source intellectual property rights, the Apache Group reorganized as a nonprofit organization—the Apache Software Foundation—in 1999.

The Struts development framework was initially designed by Craig R. McClanahan. Craig, a prolific open source developer, is also one of the lead developers for another well-known Apache project, the Tomcat servlet container. He wrote the Struts framework to provide a solid underpinning for quickly developing JSP-based web applications. He donated the initial release of the Struts framework to the ASF, in May 2002.

All of the examples in this book are based on Struts release 1.2, which is the latest stable release. It is available for download from `http://struts.apache.org/`.

With this brief history of Struts complete, let's walk through how a Struts-based application works.

Walking Through Struts

Earlier in this chapter, we discussed the basics of the MVC pattern, on which the Struts development framework is based. Now, let's explore the workflow that occurs when an end user makes a request to a Struts-based application. Figure 2-3 illustrates this workflow.

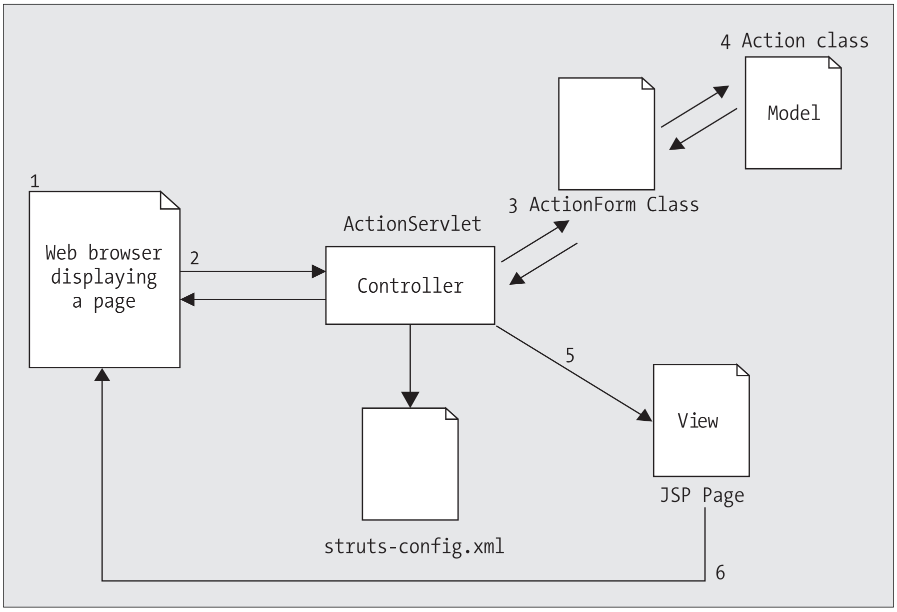

Figure 2-3. *The Struts implementation of an MVC pattern*

Imagine an end user looking at a web page (step 1). This web page, be it a static HTML page or a JavaServer Page, contains a variety of actions that the user may ask the application to undertake. These actions may include clicking a hyperlink or an image that takes them to another page, or perhaps submitting an online form that is to be processed by the application. All actions that are to be processed by the Struts framework will have a unique URL mapping (that is, /execute/*) or file extension (that is, *.do). This URL mapping or file extension is used by the servlet container to map all the requests over to the Struts `ActionServlet`.

The Struts `ActionServlet` acts as the controller for the Struts MVC implementation. The `ActionServlet` will take the incoming user request (step 2) and map it to an action mapping defined in the struts-config.xml file. The struts-config.xml file contains all of the configuration information needed by the Struts framework to process an end user's request. An `<action>` is an XML tag defined in the struts-config.xml file that tells the `ActionServlet` the following information:

- The `Action` class that is going to carry out the end user's request. An `Action` class is a Struts class that is extended by the application developer. Its primary responsibility is to contain all of the logic necessary to process an end user's request.
- An `ActionForm` class that will validate any form data that is submitted by the end user. It is extended by the developer. It is important to note that not every action in a Struts application requires an `ActionForm` class. An `ActionForm` class is necessary only when the data posted by an end user needs to be validated. An `ActionForm` class is also used by the `Action` class to retrieve the form data submitted by the end user. An `ActionForm` class will have `get()` and `set()` methods to retrieve each of the pieces of the form data. This will be discussed in greater detail in Chapter 3.
- Where the users are to be forwarded to after their request has been processed by the `Action` class. There can be multiple outcomes from an end user's request. Thus, an action mapping can contain multiple forward paths. A forward path, which is denoted by the `<forward>` tag, is used by the Struts `ActionServlet` to direct the user to another JSP page or to another action mapping in the struts-config.xml file.

Once the controller has collected all of the preceding information from the `<action>` element for the request, it will process the end user's request. If the `<action>` element indicates that the end user is posting the form data that needs to be validated, the `ActionServlet` will direct the request to the defined `ActionForm` class (step 3).

An `ActionForm` class contains a method called `validate()`. (The configuration code examples given later in this chapter may help you to understand this discussion better.) The `validate()` method is overridden by the application developer and holds all of the validation logic that will be applied against the data submitted by the end user. If the validation logic is successfully applied, the user's request will be forwarded by the `ActionServlet` to the `Action` class for processing. If the user's data is not valid, an error collection called `ActionErrors` is populated by the developer and returned to the page where the data was submitted.

If the data has been successfully validated by the `ActionForm` class, or the `<action-mapping>` does not define an `ActionForm` class, the `ActionServlet` will forward the user's data to the `Action` class defined by the action mapping (step 4). The `Action` class has three public methods and several protected ones. For the purpose of this discussion, we will consider only the `execute()` method of the `Action` class. This method, which is overridden by the application developer, contains the entire business logic necessary for carrying out the end-user request.

Once the `Action` has completed processing the request, it will indicate to the `ActionServlet` where the user is to be forwarded. It does this by providing a `key` value that is used by the `ActionServlet` to look up from the action mapping. The actual code used to carry out a forward will be shown in the section called "Configuring the homePageSetup Action Element" later in this chapter. Most of the time, users will be forwarded to a JSP page that will display the results of their request (step 5). The JSP page will render the data returned from the model as an HTML page that is displayed to the end user (step 6).

In summary, a typical web screen, based on the Struts development framework, will consist of the following:

- An action that represents the code that will be executed when the user's request is being processed. Each action in the web page will map to exactly one `<action>` element defined in the struts-config.xml file. An action that is invoked by an end user will be embedded in an HTML or a JSP page as a hyperlink or as an `action` attribute inside a `<form>` tag.
- An `<action>` element that will define which `ActionForm` class, if any, will be used to validate the form data submitted by the end user. It will also define which `Action` class will be used to process the end user's request.
- An `Action` class can use one or more forwards. A forward is used to tell the `ActionServlet` which JSP page should be used to render a response to the end user's request. A forward is defined as a `<forward>` element inside of the `<action>` element. Multiple forwards can be defined within a single `<ActionMapping>` element.

Now that we have completed a conceptual overview of how a single web page in a Struts application is processed, let's look at how a single page from the JavaEdge blog is written and plugged into the Struts framework.

Getting Started: The JavaEdge Source Tree

Before diving into the basics of Struts configuration, we need to enumerate the different pieces of the JavaEdge application's source tree. The JavaEdge blog is laid out in the directory structure shown in Figure 2-4.

The root directory for the project is called waf. There are several key directories underneath it, as listed here:

- *src*: Contains the entire JavaEdge source code of the application. This directory has several subdirectories, including
  - *java*: All Java source files for the application.
  - *ojb*: All ObjectRelationalBridge configuration files. These files are discussed in greater detail in Chapter 5.
  - *web*: The entire source code of the application that is going to be put in the WEB-INF directory. Files in this directory include any image file used in the application along with any JSP files.
  - *sql*: All of the MySQL-compliant SQL scripts for creating and prepopulating the waf database used by the JavaEdge application.
- *build*: Contains the Ant build scripts used to compile, test, and deploy the application.
- *lib*: Contains the jar files for the various open source projects used to build the JavaEdge application.

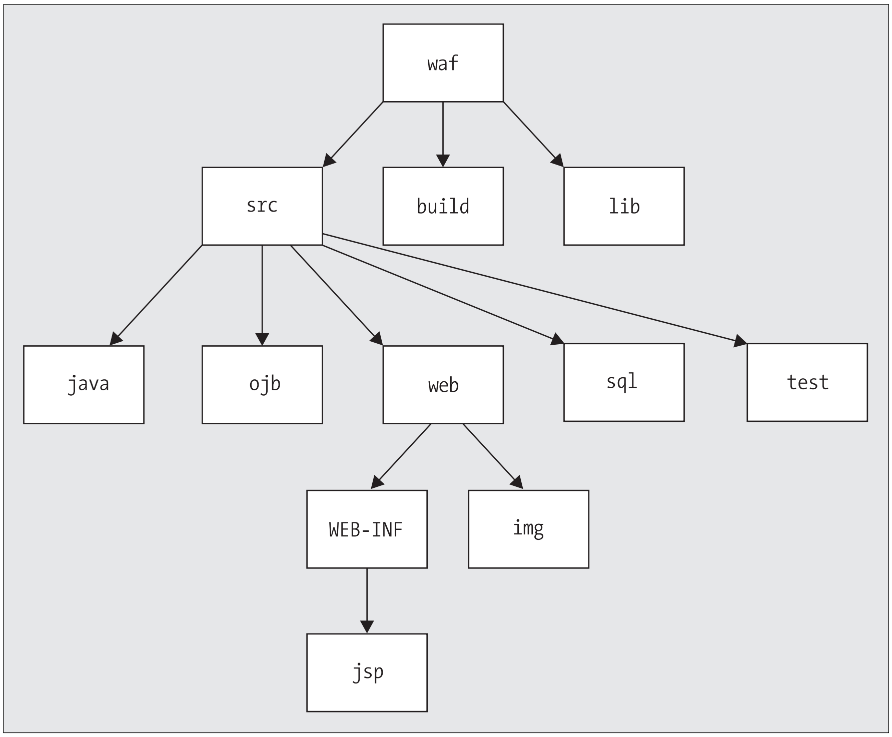

Figure 2-4. *The JavaEdge directory structure*

The JavaEdge application is built, tested, and deployed with the following software:

Tomcat 5.5.16: Tomcat is an implementation of Sun Microsystems' Servlet and JSP specifications. It is considered by Sun Microsystems to be the reference implementation for its specifications. The JavaEdge application is built and deployed around Tomcat. In Chapter 4, the open source application server bundle, JBoss 3/Tomcat 5.5.16, is used to run the application. Tomcat is available for download at `http://tomcat.apache.org/`. JBoss is an open source J2EE application server produced by JBoss. It can be downloaded at `http://jboss.org`.

MySQL: MySQL was chosen because it is one of the most popular open source databases available today. It is highly scalable and extremely easy to install and configure. MySQL 5.0 is available for download at `http://www.mysql.com`.

Ant: Version 1.6.5 of the Apache Software Foundation's Ant build utility can be downloaded at `http://ant.apache.org/`.

Lucene: Lucene is a Java-based open source search engine. Version 1.9.1 can be downloaded at `http://lucene.apache.org`.

Velocity: Version 1.4 of this alternative templating framework to JSP is available at `http://jakarta.apache.org/velocity/`.

ObjectRelationalBridge (OJB) 1.0.4: OJB is an open source Object Relational mapping tool available from the Apache DB Project. It can be downloaded from `http://db.apache.org/ojb`.

■**Note** All of the source code used in this book can be downloaded from the Apress web site (`http://www.apress.com`). We will not be discussing how to configure any of the development tools listed previously in this chapter. For information on how to configure these tools to run the code examples in this book, please refer to the readme.txt file packaged with the source code.

We will start the JavaEdge Struts configuration by demonstrating how to configure the application to recognize the Struts `ActionServlet`.

Configuring the ActionServlet

Any application that is going to use Struts must be configured to recognize and use the Struts `ActionServlet`. Configuring the `ActionServlet` requires that you manipulate two separate configuration files:

web.xml: Your first task is to configure the Struts `ActionServlet` as you would any other servlet by adding the appropriate entries to the web.xml file.

struts-config.xml: Your second task is to configure the internals of the `ActionServlet`. Since version 1.1 of the Struts framework, the recommended mechanism for this configuration is to use the struts-config.xml file. You can still configure the `ActionServlet` using the `init-param` tag in web.xml, but this feature will be removed at a later date and is now officially deprecated.

An example of the `<servlet>` tag that is used to configure the `ActionServlet` for the JavaEdge application is shown here:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com//dtd/web-app_2_3.dtd">

<web-app>

  <!--Setting up the MemberFilter-->
  <filter>
    <filter-name>MemberFilter</filter-name>
    <filter-class>com.apress.javaedge.common.MemberFilter</filter-class>
  </filter>

  <filter-mapping>
    <filter-name>MemberFilter</filter-name>
    <url-pattern>/execute/*</url-pattern>
  </filter-mapping>
```

```
<!-- Standard Action Servlet Configuration (with debugging) -->
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
  <init-param>
         <param-name>config</param-name>
         <param-value>/WEB-INF/struts-config.xml </param-value>
   </init-param>
   <init-param>
         <param-name>validating</param-name>
         <param-value>true </param-value>
   </init-param>
  <load-on-startup>2</load-on-startup>
</servlet>

<!-- Standard Action Servlet Mapping -->
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>/execute/*</url-pattern>
</servlet-mapping>

<!-- The Usual Welcome File List -->
<welcome-file-list>
  <welcome-file>default.jsp</welcome-file>
</welcome-file-list>

<taglib>
  <taglib-uri>/taglibs/struts-bean</taglib-uri>
  <taglib-location>/WEB-INF/taglibs/struts-bean.tld</taglib-location>
</taglib>

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

<taglib>
  <taglib-uri>/taglibs/struts-logic</taglib-uri>
  <taglib-location>/WEB-INF/taglibs/struts-logic.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>/taglibs/struts-template</taglib-uri>
  <taglib-location>/WEB-INF/taglibs/struts-template.tld</taglib-location>
</taglib>
```

```
  <taglib>
    <taglib-uri>http://jakarta.apache.org/taglibs/veltag-1.0</taglib-uri>
    <taglib-location>/WEB-INF/taglibs/veltag.tld</taglib-location>
  </taglib>

  <!-- Tiles Tage Library Descriptors -->
  <taglib>
     <taglib-uri>/taglibs/struts-tiles</taglib-uri>
     <taglib-location>/WEB-INF/taglibs/struts-tiles.tld</taglib-location>
  </taglib>
</web-app>
```

Anyone who is familiar with Java servlet configuration will realize that there is nothing particularly sophisticated going on here. The `<filter>` and `<filter-mapping>` tags define a filter that checks if the user has logged in to the application. If the user has not yet logged in, they will automatically be logged in as an anonymous user. This filter is called every time the Struts `ActionServlet` is invoked. The `<servlet>` tag defines all the information needed to use the Struts `ActionServlet` in the JavaEdge application. The `<servlet-name>` tag provides a name for the servlet. The `<servlet-class>` tag indicates the fully qualified Java class name of the Struts `ActionServlet`.

From the preceding example, you will notice that not all configuration settings have been moved into the struts-config.xml file. Mainly, the configuration parameters that are still specified using the `<init-param>` tag are those that are required to either find or read the struts-config.xml file. Specifically, you are left with the parameters in Table 2-1.

Table 2-1. *The ActionServlet's web.xml Configuration Parameters*

Parameter Name	Parameter Value
`config`	This parameter provides the `ActionServlet` with the location of the struts-config.xml file. By default the `ActionServlet` looks for struts-config.xml at /WEB-INF/struts-config.xml. If you place your struts-config.xml at this location, then you can omit this parameter, although we recommend that you always specify the location. That way if the default value for this parameter changes in a later release of Struts, then your application won't be broken.
`validating`	You should always leave this parameter set to `true`. Setting this parameter to `true` causes the struts-config.xml file to be read by a validating XML parser. This *will* at some point in your development career save you from tearing your hair out trying to debug your application only to find there is a rogue angle bracket in your config file.

The other important part of configuring the `ActionServlet` is setting up the mapping so the container passes the correct requests to the Struts framework for processing. This is done by defining a `<servlet-mapping>` tag in the web.xml file. The mapping can be done in one of two ways:

- URL prefix mapping
- Extension mapping

In URL prefix mapping, the servlet container examines the URL coming in and maps it to a servlet. The `<servlet-mapping>` for the JavaEdge application is shown here:

```
<web-app>
...
  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>/execute/*</url-pattern>
  </servlet-mapping>
</web-app>
```

This servlet mapping indicates to the servlet container that any request coming into the JavaEdge application, which has a URL pattern of `/execute/*`, should be directed to the `ActionServlet` (defined by the `<servlet-name>` shown previously) running under the JavaEdge application. For example, if you want to bring up the home page for the JavaEdge application, you would point your browser to `http://localhost:8080/JavaEdge/execute/homePageSetup`, where `JavaEdge` is the application name, `execute` is the URL prefix, and `homePageSetup` is the Struts action.

■**Note** It is important to note that all URLs shown in our code examples are case sensitive and must be entered exactly as they appear.

The servlet container, upon getting this request, would go through the following steps:

1. Determine the name of the application. The user's request indicates that they are making a request for the JavaEdge application. The servlet container will then look in the web.xml file associated with the JavaEdge application.
2. The servlet container will find the servlet that it should invoke. For this, it looks for a `<servlet-mapping>` tag that matches a URL pattern called execute. In the JavaEdge web.xml file, this `<servlet-mapping>` tag maps to the `ActionServlet` (that is, the Struts `ActionServlet`).
3. The user's request is then forwarded to the `ActionServlet` running under the JavaEdge application. The `homePageSetup` in the preceding URL is the action the user would like the Struts framework to carry out. Remember, an action in Struts maps to an `<action>` element in the struts-config.xml file. (Note that we will be going through how to set up an `<action>` element in the section "Configuring the homePageSetup Action Element.") This `<action>` element defines the Java classes and JSP pages that will process the user's request.

The second way to map the user's request to the `ActionServlet` is to use extension mapping. In this method, the servlet container will take all URLs that map to a specified extension and send them to the `ActionServlet` for processing. In the example that follows, all of the URLs that end with an `*.st` extension will map to the Struts `ActionServlet`:

```
<web-app>
  <servlet-mapping>
```

```
    <servlet-name>action</servlet-name>
    <url-pattern>*.st</url-pattern>
  </servlet-mapping>
</web-app>
```

If you use extension mapping to map the user's requests to the `ActionServlet`, the URL to get to the JavaEdge home page would be `http://localhost:8080/ JavaEdge/homePageSetup.st`, where `JavaEdge` is the application name, `homePageSetup` is the Struts action, and `.st` is the extension.

For the JavaEdge application being built in the next four chapters, we will be using the URL prefix method (this is the best practice for setting up and prepopulating the screens).

Once the `ActionServlet` is configured within the container, all that is left to do is configure the actual parameters for the Struts environment. The most important piece of configuration needed is specifying the controller. Since version 1.1, the actual processing of requests has been refactored from the `ActionServlet` and placed in a controller object. This pattern, called the *Application Controller pattern*, provides a simple mechanism to decouple the processing of the Struts request from the actual physical request mechanism, in this case the `ActionServlet`. To configure the controller, you simply add this entry to the struts-config.xml file:

```
<controller
    processorClass="org.apache.struts.action.RequestProcessor">
```

Although this entry in the configuration file is entirely optional, `RequestProcessor` is the default controller, and adding it means that any changes to the Struts framework in the future, such as a change in the default controller, will not affect your application. The controller element has a wide variety of parameters for configuring the Struts request controller, the most widely used being those in Table 2-2.

Table 2-2. *Configuration Parameters for the Struts Request Controller*

Parameter Name	Parameter Value
`className`	Using this parameter, you can define a separate configuration bean to handle the configuration of the Struts controller. By default this parameter is set to `org.apache.struts.config.ControllerConfig`.
`contentType`	Using this parameter, you can configure the default content type to use for each response from the Struts controller. The default for this is text/html and the default can be overridden by each action or JSP within your application as needed.
`locale`	Set this parameter to `true` (which is the default) to store a `Locale` object in the user's session if there isn't one already present.
`maxFileSize`	If you are taking advantage of the Struts file-upload capabilities, then you can configure the maximum file size allowed for upload. You specify an integer value to represent the maximum number of bytes you wish to allow. Alternatively you can suffix the number with K, M, or G to represent kilobytes, megabytes, or gigabytes, respectively. The default for this is 250 megabytes.
`multipartClass`	By default, the `org.apache.struts.upload.CommonsMultipartRequestHandler` class is used to handle multipart uploads. If you have your own class to handle this behavior or you want to override the behavior of the default class, then you can use this parameter to do so.

The final part of this configuration is to configure a resource bundle to enable you to externalize the application's resources such as error messages, label text, and URLs. The Struts framework provides support for resource bundles in almost all areas, and its support is central to delivering a successfully internationalized application. To configure the resource bundle, you simply specify the name of the properties file that stores your externalized resources:

```
<message-resources
    parameter="ApplicationResources"
    null="false" />
```

The `parameter` attribute is the name of the properties file without the file extension that contains the application resources. For example, if your resource bundle is named ApplicationResources.properties, then the value of the parameter attribute is `ApplicationResources`.

Additional configuration parameters for both the `<controller>` and `<message-resource>` tags can be found at `http://struts.apache.org/1.x/userGuide/configuration.html`.

As the servlet configuration is complete for the JavaEdge application, let's focus on setting up and implementing your first Struts action, the `homePageSetup` action. This action sends the user to the JavaEdge home page. However, before the user actually sees the page, the action will retrieve the latest postings from the JavaEdge database. These postings will then be made available to the JSP page, called homePage.jsp.

■**Note** If you look at homePage.jsp, you will notice that it is very small and that it does not seem to contain any content. homePage.jsp describes the physical layout of the page in terms of individual screen components. The actual content for the JavaEdge home page is contained in homePageContent.jsp. Chapter 6 will go into greater detail on how to "componentize" your application's screens.

This page displays the latest ten stories in a summarized format and allows the user to log in to JavaEdge and view their personal account information. In addition, the JavaEdge reader is given a link to see the full story and any comments made by the other JavaEdge readers.

To set up the `homePageSetup` action, the following steps must be undertaken:

1. A Struts `<action>` element must be added in the struts-config.xml file.
2. An `Action` class must be written to process the user's request.
3. A JSP page, in this case homePage.jsp, must be written to render the end user's request.

It is important to note that the Struts framework follows all of Sun Microsystems' guidelines for building and deploying web-based applications. The installation instructions, shown here, can be used to configure and deploy Struts-based applications in any J2EE-compliant application server or servlet container.

Configuring the homePageSetup Action Element

Setting up your first struts-config.xml file is a straightforward process. This file can be located in the WEB-INF directory of the JavaEdge project, downloaded from the Apress web site (http://www.apress.com). The location of the struts-config.xml file is also specified in the config attribute, in the web.xml entry of the ActionServlet.

The struts-config.xml file has a root element, called <struts-config>:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE struts-config
  PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
         "http://struts.apache.org/dtds/struts-config_1_1.dtd">
<struts-config>
...
</struts-config>
```

All actions for the JavaEdge application are contained in a tag called <action-mappings>. Each action has its own <action> tag. To set up homeSetupAction, you would add the following information to the struts-config.xml file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE struts-config
  PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
         "http://struts.apache.org/dtds/struts-config_1_1.dtd">
<struts-config>
  <action-mappings>
    <action
      path="/homePageSetup"
      type="com.apress.javaedge.struts.homepage.HomePageSetupAction"
      unknown="true">
      <forward name="homepage.success" path="/WEB-INF/jsp/homePage.jsp"/>
    </action>
  </action-mappings>
</struts-config>
```

An action has a number of different attributes that can be set. In this chapter, we will only be concerned with the path, type, and unknown attributes of the <action> element. The other <action> element attributes are discussed in Chapter 3. Let's now discuss the previously mentioned attributes briefly.

- path: Holds the action name. When an end-user request is made to the ActionServlet, it will search all of the actions defined in the struts-config.xml file and try to make a match, based on the value of the path attribute.

  If the ActionServlet finds a match, it will use the information in the rest of the <action> element to determine how to fulfill the user's request. In the preceding example, if users point their web browser to http://localhost:8080/JavaEdge/homePageSetup, the ActionServlet will locate the action by finding the <action> element's path attribute that matches /homePageSetup. It is important to note that all path names are case sensitive.

■Note Note that all values in the path attribute for an action must start with a forward slash (/) to map to the attribute. If you fail to put this in your path attribute, Struts will not find your action.

- type: Holds the fully qualified name of the Action class. If the user invokes the URL shown in the preceding bullet, the ActionServlet will instantiate an Action subclass of type com.apress.javaedge.struts.homepage.HomePageSetupAction. This class will contain all of the logic to look up the latest ten stories that are going to be displayed to the end user.
- unknown: Can be used by only one <action> element in the entire struts-config.xml file. When set to true, this tag tells the ActionServlet to use this <action> element as the default behavior whenever it cannot find a path attribute that matches the end user's requested action. This prevents the user from entering a wrong URL and, as a result, getting an error screen. Since the JavaEdge home page is the starting point for the entire application, we set the homePageSetup action as the default action for all unmatched requests. Only one <action> tag can have its unknown attribute set to true. The first one encountered, in the struts-config.xml file, will be used and all others will be ignored. If the unknown attribute is not specified in the <action> tag, the Struts ActionServlet will take it as false. The false value simply means that Struts will not treat the action as the default action.

An <action> tag can contain one or more <forward> tags. A <forward> tag is used to indicate where the users are to be directed after their request has been processed. It consists of two attributes, name and path. The name attribute is the name of the forward. Its value is the user-defined value that can be arbitrarily determined. The path attribute holds a relative URL, to which the user is directed by the ActionServlet after the action is completed. The value of the name attribute of the <forward> tag is a completely arbitrary name. However, this attribute is going to be used heavily by the Action class defined in the <action> tag. Later in this chapter, when we demonstrate the HomePageSetupAction class, you will find out how an Action class uses the <forward> tags for handling the screen navigation. When multiple <forward> tags exist in a single action, the Action class carrying out the processing can indicate to the ActionServlet that the user can be sent to multiple locations.

Exception handling has been greatly improved since the Struts 1.1 release. Struts now allows developers to register unchecked exceptions raised in the Struts action with the Struts ActionServlet. This concept, known as *exception handlers*, relieves developers of the need to clutter up their Action code with what is essentially the same application exception logic. Refer to Chapter 4 for more details on handling exceptions in Struts.

Sometimes, you might have to reuse the same <forward> tag across multiple <action> tags. For example, in the JavaEdge application, if an exception is raised in the business tier, it is caught and rewrapped as an ApplicationException.

In Struts version 1.0x of the JavaEdge application, when an ApplicationException is caught in an Action class, the JavaEdge application will forward the end user to a properly formatted error page. Rather than repeating the same <forward> tag in each Struts action defined in the application, you can define it to be global. This is done by adding a <global-forwards> tag at the beginning of the struts-config.xml file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE struts-config
  PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
          "http://struts.apache.org/dtds/struts-config_1_1.dtd">
<struts-config>
```

```
  <global-forwards type="org.apache.struts.action.ActionForward">
    <forward name="system.error" path="/WEB-INF/jsp/systemError.jsp"/>
  </global-forwards>
  <action-mappings>
        ...
  </action-mappings>
</struts-config>
```

The `<global-forwards>` tag has one attribute, called `type`, which defines the `ActionForward` class that forwards the user to another location. Struts is an extremely pluggable framework, and it is possible for a development team to override the base functionality of the Struts `ActionForward` class with its own implementation. If your development team is not going to override the base `ActionForward` functionality, the `type` attribute should always be set to `org.apache.struts.action.ActionForward`. After the `<global-forwards>` tag is added to the struts-config.xml file, any `Action` class in the JavaEdge application can redirect a user to `systemError.jsp` by indicating to the `ActionServlet` that the user's destination is the `system.error` forward.

Now let's discuss the corresponding `Action` class of the `homePageSetup`, that is, `HomePageSetupAction.java`.

Building HomePageSetupAction.java

The `HomePageSetupAction` class, which is located in the src/java/com/apress/javaedge/struts/homepage/HomePageSetupAction.java file, is used to retrieve the top postings made by JavaEdge users. The code for this `Action` class is shown here:

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

import com.apress.javaedge.common.ApplicationException;
import com.apress.javaedge.story.StoryManagerBD;
import com.apress.javaedge.story.IStoryManager;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.Collection;

/**
 *  Retrieves the top ten posting on JavaEdge.
 */
public class HomePageSetupAction extends Action {

    /** The execute() method comes from the base Struts Action class. You
     * override this method and put the logic to carry out the user's
     * request in the overridden method.
     */
    public ActionForward execute(ActionMapping mapping,
                                 ActionForm     form,
                                 HttpServletRequest request,
                                 HttpServletResponse response) {


        IStoryManager storyManagerBD = StoryManagerBD.getStoryManagerBD();
        Collection topStories = storyManagerBD.findTopStory();
        request.setAttribute("topStories", topStories);

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

Before we begin with the discussion on the `HomePageSetupAction` class, let's have a look at the Command design pattern.

The Power of the Command Pattern

The `Action` class is an extremely powerful development metaphor, because it is implemented using the Command design pattern.

DESIGN PATTERNS IN STRUTS

This chapter introduces the Command pattern. According to the Gang of Four's (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) definition, a Command pattern

> *Encapsulates a request as an object, thereby letting you parameterize clients with different requests . . .*
>
> —*Design Patterns, Elements of Reusable Object-Oriented Software* (Addison-Wesley, ISBN: 0-20163-361-2), p. 233

A Command pattern lets the developer encapsulate a set of behaviors in an object and provides a standard interface for executing that behavior. Other objects can also invoke the behavior, but they have no exposure to how the behavior is implemented. This pattern is implemented with a concrete class and either an abstract class or an interface.

The JavaEdge application uses different J2EE design patterns such as the Business Delegate and Value Object patterns. Chapters 4 and 5 will explore these patterns in greater detail.

The parent class or interface contains a single method definition (usually named `perform()` or `execute()`) that carries out some kind of action. The actual behavior for the requested action is implemented in a child class (which, in this example, is `HomePageSetupAction`), extending the `Command` class. The Struts `Action` class is the parent class in the Command pattern implementation. Figure 2-5 illustrates the relationship between the `Action` and `HomePageSetupAction` classes.

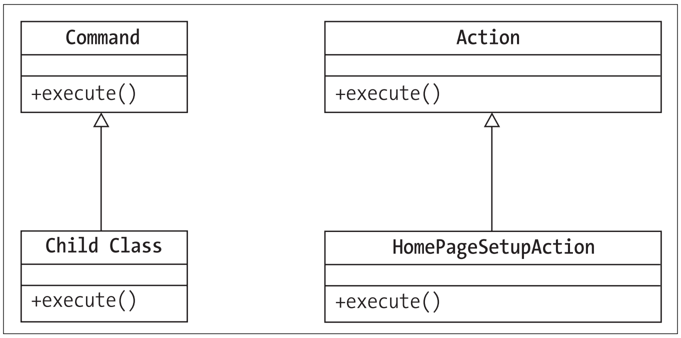

Figure 2-5. *A simple object model of the Action and HomePageSetupAction classes*

The use of the Command design pattern is one of reasons why Struts is so flexible. The `ActionServlet` does not care how a user request is to be executed. It only knows that it has a class that descends from `Action` and will have an `execute()` method. When the end user makes a request, the `ActionServlet` just executes the `execute()` method in the class that has been defined in struts-config.xml. If the development team wants to change the way in which an end-user request is processed, it can do it in two ways: either rewrite the code for the already implemented `Action` class or write a new `Action` class and modify the struts-config.xml file to point to the new `Action` class. The `ActionServlet` never knows that this change has occurred. Later in this section, we will discuss how Struts' flexible architecture can be used to solve the Hardwired antipattern. For the sake of this discussion on the Command pattern, let's go back to the `HomePageSetupAction` class.

The first step in writing the `HomeSetupAction` class is to extend the Struts `Action` class:

```
public class HomePageSetupAction extends Action
```

Next, the `execute()` method for the class needs to be overridden. (In the `Action` class source code, several `execute()` methods can be overridden, some of which are deprecated as of version 1.1. Other methods allow you to make requests to Struts from a non-HTTP-based call. For the purpose of this book, we will be dealing with only HTTP-based `execute()` methods.)

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

The `execute()` method signature takes four parameters:

- `ActionMapping`: Used to find an `ActionForward` from the struts-config.xml file and return it to the `ActionServlet`. This `ActionForward` class contains all the information needed by the `ActionServlet` to forward end users to the next page in the application.

- `ActionForm`: A helper class that is used to hold any form data submitted by the end user. The `ActionForm` class is not being used in the `HomePageSetupAction` class shown earlier. This class will be discussed in greater detail in Chapters 3 and 4.

- HttpServletRequest: A standard HttpServletRequest object passed around within the servlet.
- HttpServletResponse: A standard HttpServletResponse object passed around within the servlet.

Now let's look at the actual implementation of the execute() method:

```
IStoryManager storyManagerBD = StoryManagerBD.getStoryManagerBD();
Collection topStories = storyManagerBD.findTopStory();
```

The first step, carried out by the execute() method, is to use the StoryManagerBD class to retrieve a business delegate of type IStoryManager. The storyManagerBD variable is then used to retrieve a Collection, called topStories, of the top stories currently submitted to the JavaEdge application. The topStories collection holds up to ten instances of type StoryVO. A StoryVO is based on the J2EE design pattern called the *Value Object pattern*. A value object is used to wrap data retrieved from a data source in a Java-based implementation-neutral interface. Each StoryVO in the topStories collection represents a single row of data retrieved from the JavaEdge database's story table.

The Business Delegate pattern is a J2EE design pattern used to abstract away how a piece of business logic is actually being invoked and carried out. In the preceding example, the HomePageSetupAction class does not know how the StoryManagerBD class is actually retrieving the collection of stories. The StoryManagerBD could be using an EJB, Web service, or a Plain Old Java Object to carry out the requested action.

■ **Note** The term *J2EE patterns* is a bit of a misnomer. The Business Delegate pattern and Value Object pattern—also known as the Data Transfer Object pattern—were used in other languages before Java. However, they were called J2EE patterns when the patterns were explained in the book *Core J2EE Design Patterns: Best Practices and Design Strategies* (Alur et al., Prentice Hall, ISBN: 0-13064-884-1).

After the storyManagerBD.findTopStory() method is executed, the topStories object will be placed as an attribute of the request object:

```
request.setAttribute("topStories", topStories);
```

When the ActionServlet forwards this to the homePage.jsp page (as defined in the struts-config.xml file), the homePage.jsp will be able to walk through each item in the topStories Collection and display the data in it to the end user.

Once the story data has been retrieved, an ActionForward will be generated by calling the findForward() method in the mapping object passed into the execute() method:

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

We have finished showing you how to configure the struts-config.xml file and build an Action class to prepopulate the JavaEdge's home screen with story data. Before we look at the JSP file, homePage.jsp, let's discuss how to refactor the Hardwired antipattern.

Refactoring the Hardwired Antipattern

The declarative architecture of the Struts development framework provides a powerful tool for avoiding or refactoring a Hardwired antipattern. (Refer to Chapter 1 for the discussion of Hardwired and other antipatterns.)

All activities executed by the user in a Struts-based application should be captured within an `<action>` tag defined in the struts-config.xml file. Using an `<action>` tag gives the developer flexibility in the way in which the screen navigation and application of business rules are carried.

The advantage of using the `<action>` tag is that it forces the development team members to take a declarative approach to writing their applications. It decouples the various pieces of code associated with building out a screen from one another. For instance, when JSP developers build an application without the Struts framework, they oftentimes will have a JSP page directly invoke a piece of business logic to process a user's request.

This essentially "hardwires" the JSP page to that piece of business logic. If you want to change the behavior of the application, you need to either rewrite the class containing the business logic or have the JSP call a completely different method or class containing the new business logic. The problem is that modifying the relationship between the calling code (the JSP) and the called code (the Java class containing the business logic) is easy to do when dealing with one or two applications. However, maintaining this type of relationship in an enterprise environment where the same caller/called relationship might occur in 10 to 20 applications can be extremely difficult.

What the `<action>` tag allows is for the development team to extract the caller/called relationship out of the code into a metadata file (struts-config.xml). The development team describes caller/called relationships in a declarative fashion, rather than programmatically. If the development team wants to change the behavior of a screen in an application, it can modify the `<action>` tag to describe a new Struts `Action` class to carry out users' requests. The development team still has to write code to implement the new functionality, but there are now fewer touchpoints in the existing application that it has to modify.

This all ties back to the following:

■ **Note** If you touch the code, you break the code. The less code you have to modify to implement new functionality, the less chance there is that existing functionality will be broken and cause a ripple of destructive behavior through your applications.

According to our experience, while building a Struts application, `<action>` elements defined within the application fall into three general categories:

- *Setup actions*: Used to perform any activities that take place before the user sees a screen. In the JavaEdge home page example, you use the `/HomePageSetup` action to retrieve the top stories from the JavaEdge database and place them as an attribute in the `HttpServletRequest` object.
- *Form actions*: Actions that will process the data collected from the end user.
- *Tear-down actions*: Can be invoked after a user's request has been processed. Usually, this type of action carries out any cleanup needed after the user's request has been processed.

These three types of actions are purely conceptual. There is no way in the Struts `<action>` tag to indicate that the action being defined is a setup, form, or tear-down action. However, this classification is very useful for your own Struts applications. A setup action allows you to easily enforce "precondition" logic before sending a user to a form. This logic ensures that, before the user even sees the page, certain conditions are met. Setup actions are particularly useful when you have to prepopulate a page with data. In Chapters 3 and 4, when we discuss how to collect the user data in Struts, you will find several examples of a setup action used to prepopulate a form. In addition, putting a setup action before a page gives you more flexibility in maneuvering the user. This setup action can examine the current application state of end users, and based on this state navigate them to any number of other Struts actions or JSP pages.

A form action is invoked when the user submits the data entered in an HTML form. It might insert a record into a database or just perform some simple data formatting on the data entered by the user.

A tear-down action is used to enforce "postcondition" logic. This logic ensures that after the user's request has been processed, the data needed by the application is still in a valid state. Tear-down actions might also be used to release any resources previously acquired by the end user.

As you become more comfortable with Struts, you will prefer chaining together the different actions. You will use the setup action to enforce preconditions that must exist when the user makes the initial request. The setup action usually retrieves some data from a database and puts it in one of the different JSP page contexts (that is, page, request, session, or application context). It then forwards the user to a JSP page that will display the retrieved data. If the JSP page contains a form, the user will be forwarded to a form action that will process the user's request. The form action will then forward the user to a tear-down action that will enforce any postcondition rules. If all postcondition rules are met, the tear-down action will forward the user to the next JSP page the user is going to visit.

It's important to note that by using the strategies previously defined, you can change an application's behavior by reconfiguring the struts-config.xml file. This is a better approach than to go constantly into the application source code and modify the existing business logic.

With this discussion on the Hardwired antipattern wrapped up, let's have a look at homePage.jsp and the Struts tag libraries that are used to render the HTML page that users will see after the request has been processed.

Constructing the Presentation Tier

Now we are going to look at how many of the Struts custom JSP tag libraries can be used to simplify the development of the presentation tier. With careful design and use of these tag libraries, you can literally write JSP pages without ever writing a single Java scriptlet. The Struts development framework has four sets of custom tag libraries:

- Bean
- Logic
- HTML
- Tiles

We will not be discussing the Struts HTML or the Struts Tiles tag libraries in this chapter. Instead, we will discuss these tags in Chapters 4 and 6, respectively.

Before we begin our discussion of the individual tag libraries, the web.xml file for the JavaEdge application has to be modified to include the following Tag Library Definitions (TLDs):

```
<web-app>
  ...
 <taglib>
    <taglib-uri>/taglibs/struts-bean</taglib-uri>
    <taglib-location>/WEB-INF/taglibs/struts-bean.tld</taglib-location>
  </taglib>

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

  <taglib>
    <taglib-uri>/taglibs/struts-logic</taglib-uri>
    <taglib-location>/WEB-INF/taglibs/struts-logic.tld</taglib-location>
  </taglib>

  <taglib>
    <taglib-uri>http://jakarta.apache.org/taglibs/veltag-1.0</taglib-uri>
    <taglib-location>/WEB-INF/taglibs/veltag.tld</taglib-location>
  </taglib>

  <taglib>
     <taglib-uri>/taglibs/struts-tiles</taglib-uri>
     <taglib-location>/WEB-INF/taglibs/struts-tiles.tld</taglib-location>
  </taglib>
</web-app>
```

With these TLDs added to the web.xml file, we next look at the first page in the JavaEdge application that an end user will encounter: the JavaEdge home page.

The JavaEdge Home Page

All of the pages in the JavaEdge application are broken into three core components: a header, footer, and page body. All of the pages share the same header and footer. Let's look at the header and footer JSP files (header.jsp and footer.jsp) for all JavaEdge JSP pages and the source code for the homePageContent.jsp. We are not going to go into the actual page in great deal in this section. Instead, we will explore the different Struts tags and demonstrate their use throughout the rest of this book as we construct the JavaEdge application.

■**Note** There are two sets of JSP files in the JavaEdge application. When using the URLs listed in the book for the JavaEdge application, you are going to be using JSP files using the `<template>` tags found in Struts 1.0x. These files were kept in here for backward compatibility with the first edition of this book. The `<template>` tags are going to be deprecated in future of releases of Struts.

In Chapter 6, we look at a second set of JSP files based on the Tiles framework; these replaced the `<template>` tags. The JSP files in the first set use the exact same set of JSP code, with the only difference being how the screens are componentized. So anything you see in this chapter and the following regarding the JSP tag libraries is applicable to the code seen in Chapter 6.

header.jsp

Following is the code for the header JSP file:

```
<%@ 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-template" prefix="template" %>

<font size="7"><bean:message key="javaedge.header.title"/></font></p>

<div align="center">
  <center>
  <html:form action="login">
  <table border="0" cellpadding="0"
             cellspacing="0" style="border-collapse: collapse"
             bordercolor="#111111" width="100%" id="AutoNumber1"
             bgcolor="#FF66FF">
    <tr>
      <logic:notEqual scope="session"
             name="memberVO" property="memberId" value="1">
         <td width="16%" bgcolor="#99CCFF" align="center">
            <bean:message key="javaedge.header.logout"/>
         </td>
      </logic:notEqual>
      <logic:notEqual scope="session"
           name="memberVO" property="memberId" value="1">
         <td width="17%" bgcolor="#99CCFF" align="center">
            <bean:message key="javaedge.header.myaccount"/>
         </td>
      </logic:notEqual>

      <td width="17%" bgcolor="#99CCFF" align="center">
         <bean:message key="javaedge.header.postastory"/>
      </td>

      <td width="17%" bgcolor="#99CCFF" align="center">
          <bean:message key="javaedge.header.viewallstories"/>
       </td>

      <td width="17%" bgcolor="#99CCFF" align="center">
```

```
          <bean:message key="javaedge.header.search"/>
       </td>

      <logic:equal scope="session" name="memberVO" property="memberId" value="1">
        <td width="17%" bgcolor="#99CCFF" align="center">
          <bean:message key="javaedge.header.signup"/>
        </td>
      </logic:equal>
    </tr>
    <tr>
       <logic:equal scope="session" name="memberVO" property="memberId" value="1">
        <td width="16%" bgcolor="#99CCFF" align="left" colspan="4">
           <bean:message key="javaedge.header.userid"/>
                    <input type="text" name="userId"/>
           <bean:message key="javaedge.header.password"/>
                    <input type="password" name="password"/>
           <html:submit property="submitButton" value="Submit"/>
           <html:errors property="invalid.login"/>
        </td>
      </logic:equal>
    </tr>

  </table>
  </html:form>
  </center>
</div>
```

footer.jsp

Next is the code for the footer JSP file:

```
<%@ 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-template" prefix="template" %>

<table border="0" cellpadding="0"
           cellspacing="0" style="border-collapse: collapse"
           bordercolor="#111111" width="100%"
           id="AutoNumber1" bgcolor="#FF66FF">
    <tr bgcolor="#99CCFF">
       <td>
          &nbsp;
       </td>
    </tr>
</table>
```

homePageContent.jsp

Here is the code for homePageContent.jsp:

```
<%@ 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-template" prefix="template" %>

<BR/><BR/>
<H1>Today's Top Stories</H1>
<TABLE>
      <logic:iterate id="story" name="topStories"
             scope="request" type="com.apress.javaedge.story.StoryVO">
        <TR bgcolor="#99CCFF">
             <TD>

               <bean:write name="story" scope="page" property="storyTitle"/><BR/>
               <FONT size="1">
                  Posted By: <bean:write name="story"
                                                  property="storyAuthor.firstName"/>
                  <bean:write name="story" property="storyAuthor.lastName"/>
                  on <bean:write name="story" property="submissionDate"/>
               </FONT>
             </TD>
           </TR>
           <TR>
             <TD>
               <bean:write name="story" property="storyIntro"/>
             </TD>
           </TR>
           <TR>
             <TD align="right">
                <a href='/JavaEdge/execute/storyDetailSetup?storyId=③
                  <bean:write name="story" property="storyId"/>'>
                               Full Story</a><BR/><BR/>
             </TD>
           </TR>
       </logic:iterate>
   </TABLE>
```

Now let's break these different pages apart and see how the Struts JSP tag libraries were used to build the pages. Let's start with the Struts bean tags.

Bean Tags

Well-designed JSP pages use JavaBeans to separate the presentation logic in the application from the data that is going to be displayed on the screen. A JavaBean is a regular class that can contain the data and logic. In the JavaEdge home page example, the HomePageSetupAction class retrieves a set of StoryVO objects into a collection and puts them into the session. The StoryVO class is a JavaBean that encapsulates all of the data for a single story posted in the JavaEdge database. Each data element, stored within a StoryVO object, has a getXXX() and setXXX() method for each property. The code for the StoryVO class is shown here:

```
package com.apress.javaedge.story;

    import com.apress.javaedge.common.ValueObject;
    import com.apress.javaedge.member.MemberVO;

    import java.util.Vector;

    /**
     * Holds story data retrieved from the JavaEdge database.
     */
    public class StoryVO extends ValueObject {

        private Long storyId;
        private String storyTitle;
        private String storyIntro;
        private byte[] storyBody;
        private java.sql.Date submissionDate;
        private Long memberId;
        private MemberVO storyAuthor;
        public Vector comments = new Vector(); // of type StoryCommentVO


        public Long getStoryId() {
            return storyId;
        }

        public void setStoryId(Long storyId) {
            this.storyId = storyId;
        }

        public String getStoryTitle() {
            return storyTitle;
        }

        public void setStoryTitle(String storyTitle) {
            this.storyTitle = storyTitle;
        }
    public String getStoryIntro() {
        return storyIntro;
    }

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

    public String getStoryBody() {
        return new String(storyBody);
    }

    public void setStoryBody(String storyBody) {
        this.storyBody = storyBody.getBytes();
    }


    public java.sql.Date getSubmissionDate() {
        return submissionDate;
    }

    public void setSubmissionDate(java.sql.Date submissionDate) {
        this.submissionDate = submissionDate;
    }


    public Vector getComments() {
        return comments;
    }

    public void setComments(Vector comments) {
        this.comments = comments;
    }

    public MemberVO getStoryAuthor() {
        return storyAuthor;
    }

    public void setStoryAuthor(MemberVO storyAuthor) {
        this.storyAuthor = storyAuthor;
    }
} // end StoryVO
```

The JSP specification defines a number of JSP tags that give the developer the ability to manipulate the contents of a JavaBean.

The Struts Bean tag library offers a significant amount of functionality beyond that offered by the standard JSP tag libraries. The functionality provided by the Bean tag library can be broken into two broad categories of functionality:

- Generating output from an existing JavaBean residing in the page, request, or session scope.
- Creating new JavaBeans. These new JavaBeans can hold the data specified by the developer or retrieved from web artifacts, such as a cookie or a value stored in an HTTP header.

We are going to begin with the most common use of the Struts bean tag, the retrieval and display of data from a JavaBean.

Two bean tags are available for generating output in the Struts Bean tag library:

- `<bean:write>`
- `<bean:message>`

The `<bean:write>` tag retrieves a value from a JavaBean and writes it to the web page being generated. Examples of this tag can be found throughout the homePageContent.jsp file. For example, the following code will retrieve the value of the `property` (`storyTitle`) from a bean, called `story`, stored in the `page` context:

```
<bean:write name="story" scope="page" property="storyTitle"/><BR/>
```

To achieve the same result via a Java scriptlet would require the following code:

```
<%
  StoryVO story = (StoryVO) pageContext.getAttribute("story");

  if (story != null){
    out.write(story.getStoryTitle());
  }
  else{
     //Throw an exception unless the <bean:write> ignore attribute is
     // set to true.
  }
%>
```

The `<bean:write>` tag supports the concept of the nested property values. For instance, the `StoryVO` class has a property called `storyAuthor`. This property holds an instance of a `MemberVO` object. The `MemberVO` class contains the data about the user who posted the original story. The homePageContent.jsp page retrieves the values from a `MemberVO` object by using a nested notation in the `<bean:write>` tag. For instance, to retrieve the first name of the user who posted one of the stories to be displayed, the following syntax is used:

```
<bean:write name="story" property="storyAuthor.firstName"/>
```

In the preceding example, the `<bean:write>` tag is retrieving the `storyAuthor` by calling `story.getStoryAuthor()` and then the `firstName` property by calling `storyAuthor.getFirstName()`.

The <bean:write> tag has the attributes listed in Table 2-3.

Table 2-3. *Attributes for the <bean:write> Tag*

Attribute Name	Attribute Description
filter	Determines whether or not characters that are sensitive in HTML should be replaced with their & counterparts. For example, if the data retrieved from a call to StoryVO.getTitle() contains an & symbol, setting the filter attribute to true would cause the <bean:write> tag to write the character as &amp. The default value for this attribute is true.
ignore	When set to true, this attribute tells the <bean:write> not to throw a runtime exception, if the bean name cannot be located in the scope specified. The <bean:write> tag will simply generate an empty string to be displayed in the page. (If scope is not specified, the same rules apply here, as specified previously.) If this attribute is not set or is set to false, a runtime exception will be thrown by the <bean:write> tag, if the requested bean cannot be found.
name	The name of the JavaBean to be retrieved.
property	The name of the property to be retrieved from the JavaBean. The <bean:write> tag uses the reflection to call the appropriate get() method of the JavaBean from which you are retrieving the data. Therefore, your JavaBean has to follow the standard JavaBean naming conventions (that is, a get prefix followed by the first letter of the method name capitalized).
scope	The scope in which to look for the JavaBean. Valid values include page, request, and session. If this attribute is not set, the <bean:write> tag will start searching for the bean at the page level and continue until it finds the bean.

The second type of tag for generating output is the Struts <bean:message> tag. The <bean:message> tag is used to separate the static content from the JSP page in which it resides. All the contents are stored in a properties file, independent of the application. The properties file consists of a name-value pair, where each piece of the text that is to be externalized is associated with a key. The <bean:message> tag will use this key to look up a particular piece of text from the properties file.

To tell the name of the properties file to the ActionServlet, you need to make sure that the application parameter is set in the web.xml file. The properties file, usually called ApplicationResources.properties, is placed in the classes directory underneath the WEB-INF directory of the deployed applications. In the JavaEdge source tree, the ApplicationResources.properties file is located in working directory/waf/src/web/WEB-INF/classes (where working directory is the one in which you are editing and compiling the application source).

For the purpose of the JavaEdge application, an <init-param> tag must be configured as shown here:

```
<servlet>
  ...
  <init-param>
    <param-name>application</param-name>
    <param-value>ApplicationResources</param-value>
  </init-param>
</Servlet>
```

The static content for the JavaEdge application has not been completely externalized using the `<bean:message>` functionality. Only the header.jsp file has been externalized. The following `<bean:message>` example, taken directly from header.jsp, will return the complete URL for the JavaEdge login page:

```
<bean:message key="javaedge.header.logout"/>
```

When this tag call is processed, it will retrieve the value for the `javaedge.header.logout` key from the ApplicationResources.properties file. All of the name-value pairs from the ApplicationResources.properties file used in the header.jsp file are shown here:

```
javaedge.header.title=The Java Edge
javaedge.header.logout=<a href="/JavaEdge/execute/LogoutSetup">Logout</a>
javaedge.header.myaccount=<a href="/JavaEdge/execute/MyAccountSetup">My Account</a>
javaedge.header.postastory=<a href="/JavaEdge/execute/postStorySetup">
      Post a Story</a>
javaedge.header.viewallstories=<a href="/JavaEdge/execute/ViewAllSetup">
      View All Stories</a>
javaedge.header.signup=<a href="/JavaEdge/execute/signUpSetup">Sign Up</a>
javaedge.header.search=<a href="/JavaEdge/execute/SearchSetup">Search</a>
```

If the `<bean:message>` tag cannot find this key in the ApplicationResources.properties file, the `<bean:message>` tag will throw a runtime exception.

The `<bean:message>` tag has the attributes listed in Table 2-4.

Table 2-4. *Attributes for the <bean:message> Tag*

Attribute Name	Attribute Description
`arg0`	Parameter value that can be passed into the text string retrieved from the properties file. For instance, if a property had the value `hello.world=Hi {0}!`, using `<bean:message key="hello.world" arg="John"/>` would return the following text to the output stream: Hi John!. The `<bean:message>` tag can support at most five parameters being passed to a message.
`arg1`	Second parameter value that can be passed to the text string retrieved from the properties file.
`arg2`	Third parameter value that can be passed to the text string retrieved from the properties file.
`arg3`	Fourth parameter value that can be passed to the text string retrieved from the properties file.
`arg4`	Fifth parameter value that can be passed to the text string retrieved from the properties file.
`bundle`	The name of the application scope bean in which the `MessageResources` object containing the application messages is stored.
`key`	Key in the properties file for which the `<bean:message>` tag is going to look.
`locale`	The name of the session scope bean in which the `Locale` object is stored.

Next we'll have an interesting discussion on the Tight-Skins antipattern before moving on to bean creation.

The Tight-Skins Antipattern

Recollecting our discussion in Chapter 1, the Tight-Skins antipattern occurs when the development team does not have a presentation tier whose look and feel can be easily customized. The Tight-Skins antipattern is formed when the development team embeds the static content in the JSP pages. Any changes to the static content result in having to hunt through all of the pages in the application and making the required changes.

As you saw earlier, the `<bean:message>` tag can be used to centralize all the static content in an application to a single file called ApplicationResources.properties. However, the real strength of this tag is it makes it very easy to write internationalized applications that can support multiple languages. The JavaEdge header toolbar is written to support only English. However, if you want the JavaEdge's header toolbar to support French, you need to follow these steps:

1. Create a new file called ApplicationResources_fr.properties.

   The _fr extension to the ApplicationResources.properties file is not just a naming convention followed here. This extension is part of the ISO-3166 standard. For a complete list of all of the country codes supported by this standard, please visit `http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt`.

2. Copy all of the name-value pairs from the JavaEdge's application into the new ApplicationResources_fr.properties file. Translate all of the static contents in these name-value pairs to French. Also, if the JavaEdge application is going to support only French, you may rename the file from ApplicationResources_fr.properties to ApplicationResources.properties and replace the existing ApplicationResources.properties file. However, if you want to support English and French at the same time, you to need to tell Struts which `java.util.Locale` is to be used for the user. A `Locale` object is part of the standard Java SDK and is used to hold the information about a region. For more details on the `Locale` object, please refer to the Sun JDK documentation (available at `http://java.sun.com`).

3. To support both English and French concurrently, you could ask the users the language in which they want to see the site when they are registering for a JavaEdge account. Their language preference could be stored in the JavaEdge database. If a user chooses French as their language preference, then anytime that user logs in to the JavaEdge application, the following code could be executed in any `Action` class to switch the language preference from English over to French:

```
HttpSession session = request.getSession();
session.setAttribute(org.apache.struts.action.Action.LOCALE_KEY,
                   new java.util.Locale(LOCALE.FRENCH, LOCALE.FRENCH) );
```

Struts stores a `Locale` object in the session as the attribute key `org.apache.struts.action.Action.LOCALE_KEY`. Including a new `Locale` object (which is instantiated with the values for French) will cause Struts to reference the ApplicationResources_fr.properties file for the time for which the user's session is valid (or at least until a new `Locale` object containing another region's information is placed in the user's session).

Accessing Indexed or Mapped Data

The Struts JSP tag libraries allow you to directly access a Java object stored inside of an `Array`, `List`, or `Map` object. For example, say you modified the `MemberVO` class to contain an array of all of the addresses associated with the JavaEdge user. This modification would `get()` and `set()` a `String` array containing all of the address information:

```
public String[] getAddresses(){
    return addresses;
}

public void setAddresses(String[] addresses){
    this.addresses=addresses;
}
```

As you will see later in the chapter, you can walk through the returned array by using the `<logic:iterate>` tag to retrieve each individual address stored in the array or `Collection`. However, if you wanted to directly access an address via an array index, you can use the following syntax:

```
<bean:write name="memberVO" scope="request" property="addresses[1] "/>
```

Behind the scenes, the preceding code would be the equivalent of the following JSP code:

```
<%
     MemberVO memberVO = (MemberVO) request.getAttribute("memberVO");

     String[] addresses = memberVO.getAddresses();
     out.write(addresses[1]);
%>
```

Now let’s make the address code a little bit more sophisticated. Let’s create a value object, called `AddressVO`, to hold the entire address record. The code for `AddressVO` is shown here:

```
package com.apress.javaedge.member;

public class AddressVO {
  public static final String HOME_ADDRESS="HOME";
  public static final String BUSINESS_ADDRESS="BUS";
  public static final String TEMPORARY_ADDRESS="TEMP";

  private String addressId;
  private String addressType;
  private String street1;
  private String street2;
  private String street3;
  private String city;
  private String state;
  private String zip;
  private String country;
```

```
    public String getCountry() {
        return country;
    }

    public void setCountry(String country) {
        this.country = country;
    }

    public String getZip() {
        return zip;
    }

    public void setZip(String zip) {
        this.zip = zip;
    }

    public String getAddressId() {
        return addressId;
    }

    public void setAddressId(String addressId) {
        this.addressId = addressId;
    }

    public String getAddressType() {
        return addressType;
    }

    public void setAddressType(String addressType) {
        this.addressType = addressType;
    }

    public String getStreet1() {
        return street1;
    }

    public void setStreet1(String street1) {
        this.street1 = street1;
    }

    public String getStreet2() {
        return street2;
    }

    public void setStreet2(String street2) {
        this.street2 = street2;
    }
    public String getStreet3() {
        return street3;
    }

    public void setStreet3(String street3) {
        this.street3 = street3;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }
}
```

Let's rewrite the `getAddress()` and `setAddress()` methods to return a specific `AddressVO`. The `getAddress()` and `setAddress()` methods would "wrapper" a `HashMap` object and allow the user to return a specific address by a type: BUSINESS, HOME, or TEMPORARY. The `getAddress()` and `setAddress()` methods on the `MemberVO` for retrieving the addresses would look something like this:

```
public void setAddress (String addressType, Object address){
  addresses.put(addressType, address);
}

public Object getAddress(String addressType)
  Object holder = addresses.get(addressType);

  if (holder==null) return "";

  return holder;
}
```

If you want to use a Struts custom tag library to access directly a property on the business address for a JavaEdge member, the syntax would look something like this:

```
<bean:write name="memberVO" scope="request"
            property="address(BUSINESS).street1"/>
```

The preceding code translates into the following JSP code:

```
<%
     MemberVO memberVO = (MemberVO) request.getAttribute("memberVO");
     out.write(memberVO.getAddress("BUSINESS").getStreet1());
%>
```

One thing to be concerned about is ensuring the value returned from the `HashMap` is actually a valid object. If the object being requested is not found, the `getAddress()` method must decide how to handle the returned `NULL` value.

The concept of accessing mapped properties is an extremely powerful one. We will explore it in greater detail in Chapter 3, where we will examine how to use the `HashMap` and the `ActionForm` classes to build dynamic `ActionForms`.

Bean Creation

Struts offers a number of helper tags (bean creation tags) for creating the JavaBeans to be used within a JSP page. With these tags, a number of tasks can be carried out within the JSP page, without the need to write Java scriptlet code. These tasks include

- Retrieving a value from a cookie and creating a new JavaBean to hold the cookie's contents
- Retrieving an HTTP parameter and storing its value in a new JavaBean
- Retrieving a configuration element of Struts (such as a forward, mapping, or form bean) and storing its information in a JavaBean
- Retrieving an object from the JSP page context (that is, the application, request, response, or session objects)
- Defining a new JavaBean from scratch and placing a value in it
- Copying the contents of a single property from an existing JavaBean into a new JavaBean

Table 2-5 gives a brief summary of the different bean creation tags available.

Table 2-5. *The Different Struts Bean Creation Tags*

Bean Name	Bean Description
`<bean:cookie>`	Creates a new JavaBean to hold the contents of the specified cookie. To retrieve a cookie named `shoppingCart` into a bean, you use the following syntax: `<bean:cookie id="cart" name="shoppingCart" value="None"/>`. This call creates a new JavaBean called `cart`, which will hold the value stored in the cookie `shoppingCart`. The `value` attribute tells the bean to store the string `None`, if the cookie cannot be found. Essentially, the `value` attribute allows you to define a default value for a cookie if no cookie with the correct name can be found. If the `value` attribute is not specified and the cookie cannot be found, a runtime exception will be raised.
`<bean:define>`	Creates a new JavaBean and populates it with a string value defined by the developer. The following `<bean:define>` tag creates a JavaBean called `hello` that will hold the ever ubiquitous phrase, `"Hello World"`: `<bean:define id="hello" value="Hello World" scope="session"/>`. This bean will be placed in a session of the application.

Bean Name	Bean Description
`<bean:header>`	Creates a new JavaBean and populates it with an item retrieved from the HTTP header. In the following example, the `referer` property is being pulled out of the HTTP header and placed in a bean called `httpReferer`: `<bean:header id="httpReferer" name="referer"/>`. However, since no `value` attribute is being defined, a runtime exception will be thrown if the `referer` value cannot be found in the HTTP header.
`<bean:include>`	Creates a JavaBean to hold the content returned from a call to another URL. The following example will take the content retrieved from a call to the /test.jsp page and place it in a JavaBean called `testInclude`: `<bean:include id="testInclude" name="/test.jsp"/>`.
`<bean:page>`	Creates a new JavaBean to hold an object retrieved from the JSP page context. The following example will retrieve the session object from the `HttpServletRequest` object and place it as a JavaBean called `hSession`: `<bean:page id="hSession" property="session"/>`.
`<bean:parameter>`	Creates a new JavaBean to hold the contents of a parameter retrieved from the `HttpServletRequest` object. To retrieve a request parameter, called `sendEmail`, from the `HttpServletRequest`, you use the following code: `<bean:parameter id="sendEmailFlag" name="sendEmail" value="None"/>`. Like the `<bean:cookie>` tag, if the `value` attribute is not specified and the requested parameter is not located, a runtime exception will be raised.
`<bean:resource>`	Retrieves the data from a file located in a web application resource file. This data can be retrieved as a string or an `InputStream` object by the tag. The following code will create a new JavaBean, called `webXmlBean`, which will hold the contents of the web.xml file as a string: `<bean:resource id="webXmlBean" name="/web.xml"/>`.
`<bean:struts>`	Creates a new JavaBean to hold the contents of a Struts configuration object. The following `<bean:struts>` tag will retrieve the `homePageSetup` action and place the corresponding object into a JavaBean called `homePageSetupMap`: `<bean:struts id="homePageSetupMap" forward="/homePageSetup"/>`.

We have not used any of the bean creation tags in the JavaEdge application. There is simply no need to use them for any of the pages in this application. Also, in our opinion, most of the bean creation tags can be included in an `Action` class using Java code. According to our experience, the overuse of the bean creation tags can clutter up the presentation code and make it difficult to follow.

Logic Tags

The Logic tag library gives the developer the ability to add a conditional and interactive control to the JSP page without having to write Java scriptlets. These tags can be broken into three basic categories:

- Tags for controlling iteration.
- Tags for determining whether a property in an existing JavaBean is equal to, not equal to, greater than, or less than another value. In addition, there are logic tags that can determine whether or not a JavaBean is present within a particular JSP page context (that is, page, request, session, or application scope).
- Tags for moving (that is, redirecting or forwarding) a user to another page in the application.

Iteration Tags

The Logic tag library has a single tag, called `<logic:iterate>`, which can be used to cycle through a `Collection` object in the JSP page context. Recollect that in the `HomePageSetupAction` class, a collection of `StoryVO` objects is placed into the `request`. This collection holds the latest ten stories posted to the JavaEdge site. In the homePageContent.jsp page, you cycle through each of the `StoryVO` objects in the `request` by using the `<logic:iterate>` tag:

```
<logic:iterate id="story" name="topStories" scope="request"
               type="com.apress.javaedge.valueobject.StoryVO">
  <TR bgcolor="#99CCFF">
    <TD>
      <bean:write name="story" scope="page" property="storyTitle"/><BR/>
      ...
</logic:iterate>
```

In the preceding code snippet, the `<logic:iterate>` tag looks up the `topStories` collection in the `request` object of the JSP page. The `name` attribute defines the name of the collection. The `scope` attribute defines the scope in which the `<logic:iterate>` tag is going to search for the JavaBean. The `type` attribute defines the Java class that is going to be pulled out of the collection, in this case, `StoryVO`. The `id` attribute holds the name of the JavaBean, which holds a reference to the `StoryVO` pulled out of the collection. When referencing an individual bean in the `<logic:iterate>` tag, you use the `<bean:write>` tag. The `name` attribute of the `<bean:write>` tag must match the `id` attribute defined in the `<logic:iterate>`.

```
<bean:write name="story" scope="page" property="storyTitle"/>
```

Keep in mind the following points while using the `<logic:iterate>` tag:

- Multiple types of collections can be supported by the `<logic:iterate>` tag. These types include
    - Java `Collection` objects
    - Java `Map` objects
    - Arrays of objects or primitives
    - Java `Enumeration` objects
    - Java `Iterator` objects
- If your collection can contain `NULL` values, the `<logic:iterate>` tag will still go through the actions defined in the loop. It is the developer's responsibility to check if a `NULL` value is present by using the `<logic:present>` or `<logic:notPresent>` tags. (These tags will be covered in the next section, "Conditional Tags.")

Conditional Tags

The Struts development framework also provides a number of tags to perform basic conditional logic. Using these tags, a JSP developer can perform a number of conditional checks on the common servlet container properties. These conditional tags can check for the presence of the value of a piece of data stored as one of the following types:

- Cookie
- HTTP header
- `HttpServletRequest` parameter
- JavaBean
- Property on a JavaBean

The Struts conditional tags `<logic:equal>` and `<logic:notEqual>` can be used to test the equality or nonequality of a value sitting in a cookie, header variable, or JavaBean. For instance, the JavaEdge application always has a `memberVO` placed in the session of the user using the application.

If the user has not logged in, their session will hold a `memberVO` object whose `memberId` property is equal to `"1"`. If they are logged in, the `memberVO` will hold the data retrieved from the member table. In the header.jsp file, the `<logic:equal>` and `<logic:notEqual>` tags are used to determine whether or not a login or logout link should be displayed to the end user:

```
<logic:notEqual scope="session" name="memberVO"
                          property="memberId" value="1">
         <td width="16%" bgcolor="#99CCFF" align="center">
            <bean:message key="javaedge.header.logout"/>
         </td>
</logic:notEqual>
<logic:notEqual scope="session" name="memberVO"
property="memberId" value="1">
         <td width="17%" bgcolor="#99CCFF" align="center">
            <bean:message key="javaedge.header.myaccount"/>
         </td>
</logic:notEqual>
```

Alternatively, you could modify how security is handled in the JavaEdge application and only place a `memberVO` in the user's session when they have actually logged in. Then you could use the `<logic:present>` and `<logic:notPresent>` tags to determine if the user has logged in and then display the corresponding login/logout links:

```
<logic:notPresent scope="session" name="memberVO" >
  <td width="16%" bgcolor="#99CCFF" align="center">
    <bean:message key="javaedge.header.login"/>
  </td>
</logic:notPresent>
```

```
<logic:present scope="session" name="memberVO">
  <td width="16%" bgcolor="#99CCFF" align="center">
    <bean:message key="javaedge.header.logout"/>
  </td>
</logic:present>
```

In this JSP code, a column containing a link to the login URL will be rendered only if the JavaEdge user has not yet logged in to the application. The `<logic:notPresent>` checks the user's session to see if there is a valid `memberVO` object present in the session. The `<logic:present>` tag in the preceding code checks if there is a `memberVO` object in the user's session. If there is one, a column will be rendered containing a link to the logout page.

The `<logic:present>` and `<logic:notPresent>` tags are extremely useful, but in terms of applying the conditional logic are extremely blunt instruments. Fortunately, Struts provides you with a number of other conditional logic tags.

Conditional Logic and Cookies

Suppose that the user authentication scheme was again changed and the JavaEdge application set a flag indicating that the user was authenticated by placing a value of `true` or `false` in a cookie called `userloggedin`. You could rewrite the preceding code snippet as follows to use the `<logic:equals>` and `<logic:notEquals>` tags:

```
<logic:notEquals cookie="userloggedin" value="true">
  <td width="16%" bgcolor="#99CCFF" align="center">
    <bean:message key="javaedge.header.login"/>
  </td>
</logic:notEquals>

<logic:equals cookie="userloggedin" value="true">
  <td width="16%" bgcolor="#99CCFF" align="center">
    <bean:message key="javaedge.header.logout"/>
  </td>
</logic:equals>
```

You can use the `<logic:equals>` and `<logic:notEquals>` tags to even check a property in a JavaBean. For instance, you could rewrite the authentication piece of the JavaEdge application to set an attribute (called `authenticated`) in the `memberVO` object to a hold a string value of `true` or `false`. You could then check the property in the `memberVO` JavaBean using the following code:

```
<logic:notEquals name="memberVO" property="authenticated" scope="session"
                 value="true">
  <td width="16%" bgcolor="#99CCFF" align="center">
    <bean:message key="javaedge.header.login"/>
  </td>
</logic:notEquals>

<logic:equals name="memberVO" property="authenticated" scope="session"
              value="true">
  <td width="16%" bgcolor="#99CCFF" align="center">
```

```
    <bean:message key="javaedge.header.logout"/>
  </td>
</logic:equals>
```

When applying the conditional logic tags against a property on a JavaBean, keep two things in mind:

- *The scope that you are looking for the JavaBean in*: If you do not define a `scope` attribute, all of the contexts in the JSP will be searched. If you define this attribute and the value you are looking for is not there, a runtime exception will be thrown by the Java tag.
- *Chaining the property values of a JavaBean using dot (.) notation*: You can find examples of dot notation in the "Bean Output" section in this chapter.

Some other conditional logic tags are available:

- `<logic:greaterThan>`: Checks if the value retrieved from a JavaBean property, `HttpServletRequest` parameter, or HTTP header is greater than the value stored in the `value` attribute of the `<logic:greaterThan>` tag.
- `<logic:lessThan>`: Checks if the value retrieved from a JavaBean property, `HttpServletRequest` parameter, or HTTP header value is less than the value stored in the `value` attribute of the `<logic:lessThan>` tag.
- `<logic:greaterEqual>`: Checks if the value retrieved from a JavaBean property, `HttpServletRequest` parameter, or HTTP header value is greater than or equal to the value stored in the `value` attribute of the `<logic:greaterEqual>` tag.
- `<logic:lessEqual>`: Checks if the value retrieved from a JavaBean property, `HttpServletRequest` parameter, or HTTP header value is less than or equal to the value stored in the `value` attribute of the `<logic:lessEqual>` tag.

The logic tags just shown will try to convert the value they are retrieving to a float or double and perform a numeric comparison. If the retrieved value cannot be converted to a float or double, these tags will perform the comparisons based on the string values of the items being retrieved.

Movement Tags

These logic tags in the Struts tag library offer the developer the ability to redirect the user to a new URL. The two movement logic tags are

- `<logic:forward>`: Forwards the user to a specified global `<forward>` tag defined in the struts-config.xml file
- `<logic:redirect>`: Performs a redirect to a URL specified by the developer

Let's see how these two tags can be used. To bring up the JavaEdge application, users need to point the browser to `http://localhost:8080/JavaEdge/homePageSetup`. This forces users to know they have to go to the `/homePageSetup` action. An easier solution would be to allow them to go to `http://localhost:8080/JavaEdge`.

In a non–Struts-based application, this could be accomplished by setting up a `<welcome-file-list>` tag in the application's web.xml file. This tag allows you to define the default JSP or HTML file, which is presented when users come to the application and do not define a specific page. However, this is the problem for the Struts application. The `<welcome-file-list>` allows you to specify only filenames and not URLs or Struts actions.

However, using the movement logic tags provides you with the ability to work around this shortcoming. First, we will walk you through a solution using a `<logic:forward>` tag. You still need to set up the `<welcome-file-list>` tag in the web.xml file of JavaEdge. You are going to set up a file, called default.jsp, for the default file to be executed:

```
<web-app>
  ...
  <welcome-file-list>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

Next, you add a new `<forward>` tag, called `default.action`, to the `<global-forwards>` tag in the struts-config.xml file for the JavaEdge application:

```
<struts-config>
  <global-forwards type="org.apache.struts.action.ActionForward">
    <forward name="system.error" path="/WEB-INF/jsp/systemError.jsp"/>
    <forward name="default.action" path="/execute/homePageSetup"/>
  </global-forwards>
   ...
</struts-config>
```

The last step is to write the default.jsp file. This file contains the following two lines of code:

```
<%@ taglib uri="/taglibs/struts-logic.tld" prefix="logic" %>
<logic:forward name="default.action"/>
```

You can perform the same functionality with the `<logic:redirect>` tag. If you implement default.jsp using a `<logic:redirect>` tag, you still need to set up the default.jsp in the web.xml file. However, you do not need to add another `<forward>` tag to the `<global-forwards>` tag located in struts-config.xml. Instead, you just need to write the default.jsp in the following manner:

```
<%@ taglib uri="/taglibs/struts-logic.tld" prefix="logic" %>
<logic:redirect page="/execute/homePageSetup"/>
```

This code will generate a URL relative to the JavaEdge application (`http:// localhost:8080/Javaedge/execute/HomePageSetup`). You are not restricted, while using the `<logic:redirect>`, to redirect to a relative URL. You can also use a fully qualified URL and even redirect the user to another application. For instance, you could rewrite the default.jsp as follows:

```
<%@ taglib uri="/taglibs/struts-logic.tld" prefix="logic" %>
<logic:redirect
 href="http://localhost:8080/JavaEdge/execute/homePageSetup"/>
```

Using `<logic:redirect>` and `<logic:forward>` is the equivalent of calling the `sendRedirect()` method on the `HttpServletResponse` class in the Java Servlet API. The difference between the two tags is that the `<logic:forward>` tag will let you forward only to a `<global-forward>` defined in the struts-config.xml file. The `<logic:redirect>` tag will let you redirect to any URL.

The `<logic:redirect>` tag has a significant amount of functionality. However, you have had just a brief introduction to what the `<logic:redirect>` tag can do. A full listing of all the attributes and functionalities of this tag can be found at `http://struts.apache.org/1.x/struts-el/tlddoc/logic/redirect.html`.

Summary

In this chapter, we explored the basic elements of a Struts application and how to begin using Struts to build the applications. To build a Struts application, you need to know the following:

- The basic components of a Struts application:
    - `ActionServlet`: Represents the controller in the Struts MVC implementation. It takes all user requests and tries to map them to an `<action>` entry in the struts-config.xml file.
    - `action`: Defines a single task that can be carried by the end user. Also, it defines the class that will process the user's request and the JSP page that will render the HTML the user sees.
    - `Action` class: Contains the entire logic to process a specific user request.
    - `ActionForm`: Is associated with an `<action>` tag in the struts-config.xml file. It wraps all the form data submitted by the end user and also can perform validation on the data entered by the user.
    - *JSP pages*: Used to render the HTML pages that the users will see as a result of their request to the `ActionServlet`.
- The configuration files necessary to build a Struts application:
    - *web.xml*: This file contains the entire `ActionServlet` configuration, the mapping of user requests to the `ActionServlet`, and all the Struts Tag Library Definitions.
    - *struts-config.xml*: This file contains all the configuration information for a Struts-based application.
    - *ApplicationResources.properties*: This file is a central location for static content for a Struts application. It allows the developer to easily change the text or internationalize an application.

- The different Struts tag libraries for building the presentation piece of the application, including the following:
    - *Bean*: Provides the developer with JSP tags for generating output from a JavaBean and creating a JavaBean from common JSP web artifacts.
    - *Logic*: Can be used to apply the conditional logic in the JSP page through `Collections` stored in the user's JSP page context and redirect the user to another page.
    - *HTML*: These tags are not discussed in this chapter. However, they offer a significant amount of functionality and are discussed in greater detail in Chapters 3 and 4.

Also, we identified some different areas where Struts can be used to refactor the web antipatterns that might form during the design and implementation of web-based applications. Refactoring of the following antipatterns was discussed:

- *Hardwired*: We looked at how to chain together Struts actions to perform the precondition, form processing, and postcondition logic. This segregation of the business logic into the multiple applications provides a finer control over the application of the business logic and makes it easier to redirect the user to different Struts actions and JSP pages.
- *Tight-Skins*: While examining this antipattern, we looked at how to use the bean and logic tags to implement role-based presentation logic.

This chapter lays the foundation for the material covered in Chapters 3 and 4. In the next chapter, we are going to cover how to implement web-based forms using the Struts form tags. We will also look at the Struts HTML tag library and how it simplifies form development. Finally, the next chapter will focus on how to use the Struts `ActionForm` class to provide a common mechanism for validating the user data and reporting validation errors back to the user.