Chapter 4. Introduction to Struts

Forms Processing

Goals	After completing this chapter, the student will be able to • understand the MVC architecture. • set up an application using Struts. • use the Struts bean and html tags. • process user input from HTML forms through Struts.
Prerequisites	The student will need to have an understanding of JSP, JavaBeans, and custom tags.
Objectives	This chapter is presented to provide the student with an understanding of the Struts framework.

The Model-View-Controller Architecture

"Model-View-Controller" is a way to build applications that promotes complete separation between business logic and presentation. It is not specific to web applications, or Java, or J2EE (it predates all of these by many years), but it can be applied to building J2EE web applications.

The "view" is the user interface, the screens that the end user of the application actually sees and interacts with. In a J2EE web application, views are JSP files. For collecting user input, you will have a JSP that generates an HTML page that contains one or more HTML forms. For displaying output (like a report), you will have a JSP generates an HTML page that probably contains one or more HTML tables. Each of these is a view: a way for the end user to interact with the system, putting data in, and getting data out.

When the user clicks 'Submit' in an HTML form, the request (complete with all the form information) is sent to a "controller". In a J2EE web application, controllers are JavaBeans. The controller's job is to take the data entered by the user (in the HTML form that the JSP generated) and pass it to the "model", which is a separate Java class that contains actual business logic. The model does whatever it does (for instance, store the user's data in a database), then returns some result back to the controller (perhaps a new ID value from the database, or perhaps just a result code saying "OK, I'm done"). The controller takes this value and figures out what the user needs to see next, and presents the user with a new view (for instance, a new JSP file that displays a confirmation that the data they entered was successfully saved).

This all sounds like a lot of work, and it is. But there is a point to architecting applications this way: flexibility. The beauty of model-view-controller separation is that new views and controllers can be created independently of the model. The model -- again, this is pure business logic -- knows nothing of HTML forms or JSP pages. The model defines a set of business functions that only ever get called by controllers, and the controllers act as proxies between the end user (interacting with the view) and the business logic (encapsulated in the model). This means that you can add a new view and its associated controller, and your model doesn't know or care that there are now two different ways for human beings to interact with the application.

For instance, in an application with complicated data entry screens, you could add a JSP that generated a quick-edit form with default values instead of a longer, standard form. Both JSPs (the short form and the long form) could use the same controller; default values could simply be stored in the HTML `<form>` as hidden input values, and the controller would never know the difference. Or you could create a completely new interface -- a desktop application in addition to a web application. The desktop application would have views implemented in some interface-building tool, and have associated controllers for each screen. But these controllers would call the business functions in the model, just like the controllers in the web application. This is called "code reuse", specifically "business logic reuse", and it's not just a myth: it really can happen, and the Model-View-Controller architecture is one way to make it happen.

What is Struts?

Struts is a framework that promotes the use of the Model-View-Controller architecture for designing large scale applications. The framework includes a set of custom tag libaries and their associated Java classes, along with various utility classes. The most

powerful aspect of the Struts framework is its support for creating and processing web-based forms. We will see how this works later in this chapter.

Struts Tags

Common Attributes

Almost all tags provided by the Struts framework use the following attributes:

Attribute	Used for
`id`	the name of a bean for temporary use by the tag
`name`	the name of a pre-existing bean for use with the tag
`property`	the property of the bean named in the `name` attribute for use with the tag
`scope`	the scope to search for the bean named in the `name` attribute

Referencing Properties

Bean properties can be referenced in three different ways: simple, nested, or indexed. Shown here are examples for referencing the properties each way:

Reference Method	Example
simple	`<!-- uses tutorial.getAnAttribute() -->` `<bean:write name="tutorial" property="anAttribute"/>`

nested	`<!-- uses tutorial.getAnAttribute().getAnotherAttribute() -->`
	`<bean:write name="tutorial"`
	`property="anAttribute.anotherAttribute"/>`

indexed	`<!-- uses tutorial.getSomeAttributes(3) to access the -->`
	`<!-- fourth element of the someAttributes property array -->`
	`<bean:write name="tutorial" property="someAttributes[3]"/>`

flavorful mix of methods	`<!-- uses`
	`foo.getSomeAttributes(2).getSomeMoreAttributes(1) -->`
	`<bean:write name="foo"`
	`property="goo[2].someAttributes[1]"/>`

Creating Beans

Beans are created by Java code or tags.

Here is an example of bean creation with Java code:

```
// Creating a Plumber bean in the request scope
Plumber aPlumber = new Plumber();
request.setAttribute("plumber", aPlumber);
```

Beans can be created with the `<jsp:useBean></jsp:useBean>` tag:

```
<!-- If we want to do <jsp:setProperty ...></jsp:setProperty> or -->
<!-- <jsp:getProperty ... ></jsp:getProperty> -->
<!-- we first need to do a <jsp:useBean ... ></jsp:useBean> -->

<jsp:useBean id="aBean" scope="session" class="java.lang.String">
creating/using a bean in session scope of type java.lang.String
</jsp:useBean>
```

Most useful is the creation of beans with Struts tags:

```
<!-- Constant string bean -->
<bean:define id="greenBean" value="Here is a new constant string bean;
pun intended."/>

<!-- Copying an already existent bean, frijole, to a new bean, lima -->
<bean:define id="lima" name="frijole"/>

<!-- Copying an already existent bean, while specifying the class -->
<bean:define id="lima" name="frijole"
class="com.SomePackageName.Beans.LimaBean"/>

<!-- Copying a bean property to a different scope -->
<bean:define id="goo" name="foo" property="geeWhiz" scope="request"
toScope="application"/>
```

Other Bean Tags

The Struts framework provides other tags for dealing with issues concerning copying cookies, request headers, JSP implicity defined objects, request parameters, web application resources, Struts configuration objects, and including the dynamic response data from an action. These tags are not discussed here, but it is important to be aware of their existence.

```
<bean:cookie ... >
<bean:header ... >
<bean:page ... >
<bean:parameter ... >
<bean:header ... >
<bean:resource ... >
<bean:struts ... >
```

Bean Output

The `<bean:message>` and `<bean:write>` tags from the Struts framework will write bean and aplication resources properties into the current `HttpResponse` object.

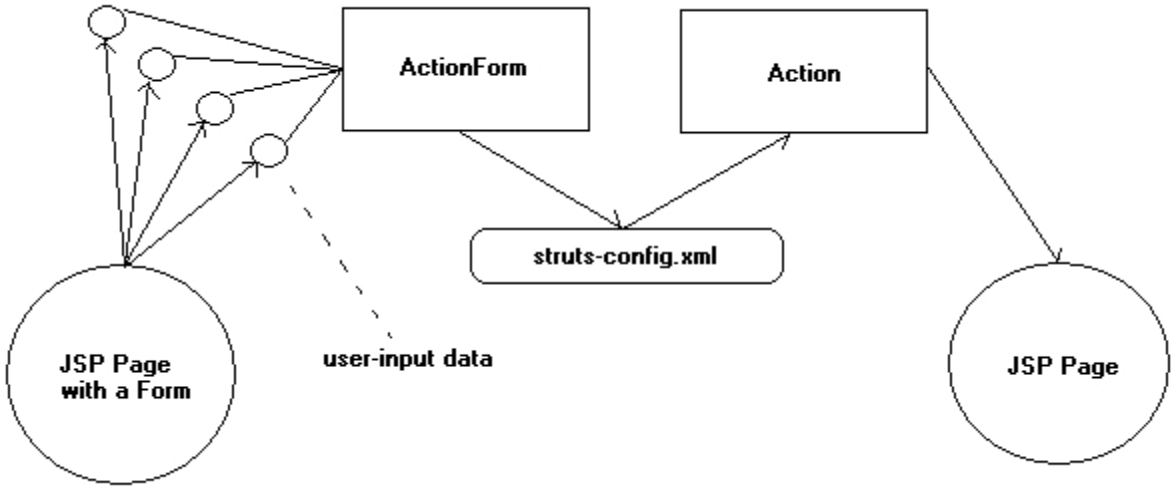
`<bean:message ... >`	This tag allows locale specific messages to be displayed by looking up the message in the application resources `.properties` file.
	
	`<!-- looks up the error.divisionByZero resource -->`
	`<!-- and writes it to the HttpResponse object -->`
	`<bean:message key="error.divisionByZero"/>`
	
	`<!-- looks up the prompt.name resource -->`
	`<!-- and writes it to the HttpResponse object; -->`
	`<!-- failing that, it writes the string -->`
	`<!-- contained in the attribute arg0-->`
	`<bean:message key="prompt.name" arg0='Enter a name:'/>`

`<bean:write ... >`	This tag writes the string equivalent of the specified bean or bean property to the current `HttpResponse` object.
	
	`<!-- writes the value of customer.getStreetAddress().toString() -->`
	`<!-- to the HttpResponse object -->`
	`<bean:write name="customer" property="streetAddress"/>`

Creating HTML Forms

Quite often information needs to be collected from a user and processed. Without the ability to collect user input, a web application would be useless. In order to get the users information, an html form is used. User input can come from several widgets, such as text fields, text boxes, check boxes, pop-up menus, and radio buttons. The data corresponding to the user input is stored in an `ActionForm` class. A configuration file called `struts-config.xml` is used to define exactly how the user input are processed.

The following diagram roughly depicts the use of Struts for using forms.

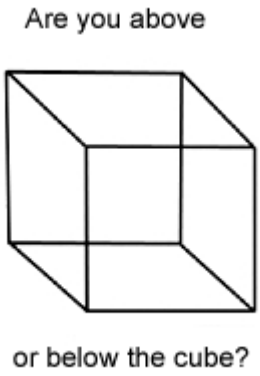


The Struts html tags are used to generate the widgets in the html that will be used in gathering the users data. There are also tags to create a form element, html body elements, links, images, and other common html elements as well as displaying errors. Below are the tags provided by html section of the Struts framework and a short description of each.

Note: Most of the tags for use with forms are able to take the `name` and `property` attributes representing the initial state of the widgets and for capturing the state that the widgets were in when a form submission occurred.

`<html:base>`	Generates a `<base>` tag. This tag should be used inside of a `<head>` tag.
`<html:button>`	Generates an `<input type="button">` tag. This tag should be used inside a `<form>` element.
`<html:cancel>`	Generates an `<input type="submit">` tag and causes the Action servlet not to invoke its `validate()` method. This tag should be used inside a `<form>` element.

`<html:checkbox>`
 	
`<html:multibox>`
	`<html:checkbox>` Generates an `<input type="checkbox">`.
	
	`<html:multibox>` Generates an `<input type="checkbox">`. "Checkedness" depends upon whether the property array specified contains a corresponding value as the one specified

	for the multibox.
`<html:errors>`	Generates html to display any errors that may have occurred during invocation of the `validate()` method.
`<html:file>`	
`<html:form>`	Generates `<form>`.
`<html:hidden>`	*There is a hidden element here which is invisible.* :-) Generates `<input type="hidden">`.
`<html:html>`	Generates `<html>`.
`<html:image>`	
`<html:img>`	
`<html:link>`	A link to an external site Generates an html link.
`<html:password>`	Generates `<input type="password">` for use in collecting information that should not be shown on-screen.
`<html:radio>`	Credit Debit Generates a radio button (`<input type="radio">`).
`<html:reset>`	Generates `<input type="reset">`.
`<html:rewrite>`	

`<html:select>`
 	
`<html:options>`
 	
`<html:option>`	

	<html:select> Generates <select>. <html:options> Generates html for an entire list of <option> tags. <html:option> Generates a single <option>.
<html:submit>	Generates <input type="submit"> to submit form data entered by the user.
<html:text>	Name Email Address Generates <input type="text">.
<html:textarea>	Generates <textarea>.

The `ActionForm` class

The purpose of the `ActionForm` class is to contain and provide validation of the user-input data. This class is subclassed for application specific customization.

Here is a template for a customized `ActionForm` class with markers denoting where special items should be located in the class with `$` symbols.

```
package $PACKAGE_NAME$;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;

import $OTHER_PACKAGES_TO_IMPORT$;

public class $THE_NAME_OF_THIS_FORM_CLASS$ extends ActionForm {

// The attributes of the class should go here
// For example, private int age;
$PRIVATE_ATTRIBUTES_FOR_THIS_CLASS$

// The constructor method for this class
public $THE_NAME_OF_THIS_FORM_CLASS$() {
```

```
}

// The method to invoke when a reset occurs
public void reset(ActionMapping mapping, HttpServletRequest request) {
}

// The method providing validation of this classes attributes
public ActionErrors validate(ActionMapping mapping, HttpServletRequest request)
{
   ActionErrors errors = new ActionErrors();

   if
($SOME_CODE_TO_VALIDATE_AN_ATTRIBUTE_OF_THIS_CLASS_RETURNING_true_ON_SUCCESS$)
{
     //
   } else {
     // Add an error to the errors to be returned that designates the
validation of the
     // current attribute failed. The information will come from the
Application Resources.
     errors.add("$THE_ATTRIBUTES_NAME$", new
ActionError("$SOME_KEY_FROM_THE_ApplicationResources.properties_FILE$"));
   }

   // Return the errors
   return errors;
}

$ACCESSOR_AND_MUTATOR_METHODS_FOR_THE_ATTRIBUTES_OF_THIS_CLASS$
// For example,
// public int getAge() { return age; }
// public void setAge(int newAge) { age = newAge; }

}
```

The `Action` class

The purpose of the `Action` class is to perform the appropriate actions on the user input gathered from the form.

Here is a template for a customized `Action` class with markers denoting where special items should be located in the class with `$` symbols.

```
package $PACKAGE_NAME$;

import java.util.Vector;
import java.io.IOException;
import java.lang.reflect.InvocationTargetException;
import java.util.Locale;
```

```
import java.util.Hashtable;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionServlet;
import org.apache.struts.util.MessageResources;
import org.apache.struts.util.PropertyUtils;

import $OTHER_PACKAGES_TO_IMPORT$;

public final class $THE_NAME_OF_THIS_ACTION_CLASS$ extends Action {

// The constructor method for this class
public $THE_NAME_OF_THIS_ACTION_CLASS$() {
}

// The method used for processing the user-input
public ActionForward perform(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)
   throws IOException, ServletException {

   Locale locale = getLocale(request);
   MessageResources messages = getResources();
   HttpSession session = request.getSession();
   $SOME_FORM_CLASS$ $SOME_FORM_CLASS_INSTANCE$ =
($SOME_FORM_CLASS$)form;

   // ActionErrors errors = new ActionErrors();

   $CODE_FOR_PROCESSING_USER_INPUT$

   if ($PROCESSING_WAS_SUCCESSFUL$) {
   } else {
     return (mapping.findForward("$FAILURE_FORWARD_PAGE$"));
   }

   if (mapping.getAttribute() != null) {

     if ("request".equals(mapping.getScope()))
       request.removeAttribute(mapping.getAttribute());
     else
```

```
        session.removeAttribute(mapping.getAttribute());

    }

    return (mapping.findForward("$SUCCESS_FORWARD_PAGE$"));

}

}
```

SimpleStruts: a simple Struts application

(If you have not already done so, you can download this and other examples used in this course. Mac OS X or other UNIX users click here instead.)

This example demonstrates the use of Struts for a simple web application using HTML forms. The application will request a name from the user and will display that name on the page containing the form.

The `NameForm` class will be used for storing the user input. The `SetNameAction` class will receive an instance of the `NameForm` class containing the user input on which to perform its action. An instance of the `Name` class will be stored in the user's session to represent the name that was last entered.

Do this:

1. Create the directory structure. The root directory is `SimpleStruts`, and it has the standard `WEB-INF` directory with `classes` inside, and `com/masslight/strutsExampleClasses` inside that. It also has a `lib` directory within `WEB-INF`, which is something we haven't seen before; we'll see in a minute what goes there.

code/Chapter4/SimpleStruts

```
SimpleStruts
  |
  +-- index.jsp (*)
  |
  +-- build.xml (*)
  |
  +-- WEB-INF
        |
        +-- web.xml (*)
        |
        +-- struts-config.xml (*)
        |
```

code/Chapter4/SimpleStruts

```
        +-- struts-bean.tld (*)
        |
        +-- struts-form.tld (*)
        |
        +-- struts-html.tld (*)
        |
        +-- struts-logic.tld (*)
        |
        +-- struts-template.tld (*)
        |
        +-- struts.tld (*)
        |
        +-- app.tld (*)
        |
        +-- classes
        |     |
        |     +-- com
        |           |
        |           +-- masslight
        |                 |
        |                 +-- strutsExampleClasses
        |                       |
        |                       +-- ApplicationResources.properties (*)
        |                       |
        |                       +-- Name.java (*)
        |                       |
        |                       +-- NameForm.java (*)
        |                       |
        |                       +-- SetNameAction.java (*)
        |
        +-- lib
              |
              +-- struts.jar (*)

(*) denotes a file
```

2. Copy the Struts tag library descriptor files into `WEB-INF`. The files `struts.tld`, `struts-bean.tld`, `struts-form.tld`, `struts-html.tld`, `struts-logic.tld`, and `struts-template.tld` are available in the `lib` directory of your Struts installation.

```
c:\j2ee\code\Chapter4\SimpleStruts\WEB-INF\> copy
c:\j2ee\struts\lib\struts*.tld .
c:\j2ee\struts\lib\struts-bean.tld
c:\j2ee\struts\lib\struts-form.tld
c:\j2ee\struts\lib\struts-html.tld
```

```
c:\j2ee\struts\lib\struts-logic.tld
c:\j2ee\struts\lib\struts-template.tld
c:\j2ee\struts\lib\struts.tld
        6 file(s) copied.
```

3. Copy the Struts parser, `struts.jar`, into `WEB-INF/lib/`. This file is available in the `lib` directory of your Struts installation
4. Create the tag descriptor library file for any custom tags you may use beyond the Struts tags. In this case, the file defines no custom tags, but it's good practice to have it in place, in case you need to add your own tags later.

code/Chapter4/SimpleStruts/WEB-INF/app.tld

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
     PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
     "http://java.sun.com/j2ee/dtds/web-jsptaglib_1_1.dtd">

<taglib>
     <tlibversion>1.0</tlibversion>
     <jspversion>1.1</jspversion>
     <shortname>utility</shortname>
     <info>
       Empty tag library template
     </info>
</taglib>
```

5. Create the `struts-config.xml` file.

   There are three main sections to a `struts-config.xml` configuration file. They are the "Form Bean Definitions" section, the "Global Forward Definitions" section, and the "Action Mapping Definitions" section.

   The `NameForm` class will be defined in the form bean definition section, where it will receive a name for use in the "Action Mapping Definitions" section. The global forward definition section defines a forward called "success".

   A global forward is defined in the "Global Forward Definitions" section. Whenever this forward is referenced, `index.jsp` will be displayed.

   The "Action Mapping Definitions" is the most important section within the configuration file. This section takes a form defined in the "Form Bean Definitions" section and maps it to an action class.

Here is what the `struts-config.xml` configuration file will look like for this application:

code/Chapter4/SimpleStruts/WEB-INF/struts-config.xml

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

<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration
1.0//EN"
          "http://jakarta.apache.org/struts/dtds/struts-config_1_0.dtd">

<struts-config>

  <!-- ========== Form Bean Definitions
=================================== -->
  <form-beans>
    <!-- name form bean -->
    <form-bean name="nameForm"
type="com.masslight.strutsExampleClasses.NameForm"/>
  </form-beans>

  <!-- ========== Global Forward Definitions
============================== -->
  <global-forwards>
    <forward name="success" path="/index.jsp"/>
  </global-forwards>

  <!-- ========== Action Mapping Definitions
============================== -->
  <action-mappings>
    <!-- Save user registration -->
    <action path="/setName"
            type="com.masslight.strutsExampleClasses.SetNameAction"
            name="nameForm"
            scope="request"
            input="/index.jsp"/>
  </action-mappings>

</struts-config>
```

6. Create the `web.xml` file.

   The `web.xml` web application configuration file will need to define the servlet `ActionServlet`, to which control will be transferred whenever an appropriate URL pattern is accessed. The servlet is defined just as any other servlet will be defined. The URL pattern is specified by a servlet mapping. For this application, the URL pattern is any requested resource that ends with a `.do` extension.

In order to use the Struts tags, the `.tld` files describing the tags will need to be included in the configuration file. The references to these tags are made just as they were for our own custom tags in the previous chapter. The Struts framework is simply a complex set of tag libraries (`struts*.tld`), with associated code (`struts.jar`).

The `web.xml` configuration file should look like this:

code/Chapter4/SimpleStruts/WEB-INF/web.xml

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

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>

  <!-- Action Servlet Configuration -->
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-
class>
    <init-param>
      <param-name>application</param-name>
      <param-
value>com.masslight.strutsExampleClasses.ApplicationResources</param-
value>
    </init-param>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <init-param>
      <param-name>debug</param-name>
      <param-value>2</param-value>
    </init-param>
    <init-param>
      <param-name>detail</param-name>
      <param-value>2</param-value>
    </init-param>
    <init-param>
      <param-name>validate</param-name>
      <param-value>true</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
  </servlet>

  <!-- Action Servlet Mapping -->
  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
```

code/Chapter4/SimpleStruts/WEB-INF/web.xml

```
  </servlet-mapping>

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

  <!-- Application Tag Library Descriptor -->
  <taglib>
    <taglib-uri>/WEB-INF/app.tld</taglib-uri>
    <taglib-location>/WEB-INF/app.tld</taglib-location>
  </taglib>

  <!-- Struts Tag Library Descriptors -->
  <taglib>
    <taglib-uri>/WEB-INF/struts-bean.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
  </taglib>

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

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

</web-app>
```

7. The `ApplicationResources.properties` file provides resources that will be used by any subclassed Struts classes (for example, `SetNameAction`). This resources file provides a place to define prompts, labels that will display on buttons, and other information that may change. By placing this information in the `ApplicationResources.properties` file, recompiling any servlets used in the application can be avoided, as well as encouraging separation of logic and presentation.

   The `ApplicationResources.properties` file looks like this:

code/Chapter4/SimpleStruts/WEB-INF/classes/com/masslight/strutsExampleClasses/ApplicationResources.properties

```
button.save=Change name
button.reset=Reset
error.name.required=To change the name, a name must be entered
prompt.name=Enter name:
welcome.title=A Simple Application
```

8. Instances of the `Name` class are placed in the user sessions. Only one will exist in any particular user session. It provides methods for accessing and mutating a name.

   Place the following inside the `Name.java` file:

code/Chapter4/SimpleStruts/WEB-INF/classes/com/masslight/strutsExampleClasses/Name.java

```
package com.masslight.strutsExampleClasses;

import java.io.Serializable;

public final class Name implements Serializable {

  private String name = null;

  public String getName() {
    return (this.name);
  }

  public void setName(String name) {
    this.name = name;
  }

  public String toString() {
    StringBuffer sb = new StringBuffer("Name[name=");
    sb.append(name);
    sb.append("]");
    return (sb.toString());
  }

}
```

9. The `NameForm` class stores and validates the user input. It provides methods for accessing and mutating the data in the form. Notice how the validate method uses information from the `ApplicationResources.properties` file.

   Place the following in the `NameForm.java` file:

code/Chapter4/SimpleStruts/WEB-INF/classes/com/masslight/strutsExampleClasses/NameForm.java

```
package com.masslight.strutsExampleClasses;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;

public final class NameForm extends ActionForm  {

  private String action = "Set";
  private String name = null;

  public String getAction() {
    return (this.action);
  }

  public void setAction(String action) {
    this.action = action;
  }

  public String getName() {
    return (this.name);
  }

  public void setName(String name) {
    this.name = name;
  }

  public void reset(ActionMapping mapping, HttpServletRequest request) {
    this.action = "Set";
    this.name = null;
  }

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

}
```

10. The `SetNameAction` class has a `perform` method, which receives an instance of the `NameForm` class containing the user input.

    Although the `perform` method here operates directly on the form data, in a real-world application it should instead delegate this duty to a business logic object. (We will see how this works in chapter 7, once we learn about EJBs.) The `perform` method takes the name from the form and creates a new instance of the `Name` class, which is placed inside the session. If a bean by that name is already present in the session, it is replaced by the new instance.

    Notice how the class uses this code:

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

    to display the next JSP. A new forward could have been defined in the `struts-config.xml` called "failed" or "DEFCON1". If any of the code in the `perform` method had failed, the user could be forwarded to the JSP page defined in the forward definition.

    Place the following code in the `SetNameAction.java` file:

code/Chapter4/SimpleStruts/WEB-INF/classes/com/masslight/strutsExampleClasses/SetNameAction.java

```
package com.masslight.strutsExampleClasses;

import java.io.IOException;
import java.lang.reflect.InvocationTargetException;
import java.util.Locale;
import java.util.Hashtable;
import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.struts.action.*;
import org.apache.struts.util.*;

public final class SetNameAction extends Action {

  public ActionForward perform(ActionMapping mapping,
                               ActionForm form,
                               HttpServletRequest request,
                               HttpServletResponse response)
    throws IOException, ServletException {

    Locale locale = getLocale(request);
    MessageResources messages = getResources();
```

code/Chapter4/SimpleStruts/WEB-INF/classes/com/masslight/strutsExampleClasses/SetNameAction.java

```
    HttpSession session = request.getSession();
    NameForm nameform = (NameForm)form;
    String action = request.getParameter("action");

    if (action == null)
      action = "Set";

    if (servlet.getDebug() >= 1)
      servlet.log("SetNameAction:  Processing " + action + " action");

    Name name = (Name)session.getAttribute("name");

    ActionErrors errors = new ActionErrors();

    String value = null;

    value = nameform.getName();

    if ("Set".equals(action)) {
      name = null;
      name = new Name();
      name.setName(nameform.getName());
    }

    session.setAttribute("name", name);

    if (mapping.getAttribute() != null) {

      if ("request".equals(mapping.getScope()))
        request.removeAttribute(mapping.getAttribute());
      else
        session.removeAttribute(mapping.getAttribute());

    }

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

  }

}
```

12. The view of the application is done with the JSP `index.jsp`. It represents the user interface and allows the user to interact with the application.

    Information from the `ApplicationResources.properties` file is used for the page title, prompts, and the `Submit` button label. This is done with the

`<bean:message>` tag. The attribute `"key"` is used to specify the application resource to lookup in the `ApplicationResources.properties` file.

Any errors that were found by the `validate` method of the `NameForm` class are displayed by using the `<html:errors/>` tag.

The `action` attribute of the `<form>` tag is `"setName"`. When the user clicks the `Submit` button, the action defined for `setName` in `struts-config.xml` is invoked.

A text field is created and associated with the property "name" in an instance of the `NameForm` class by using the attribute property in the `<html:text>` tag. Two optional attributes are used to define the length and the maximum characters to accept.

A new tag is introduced from the Struts logic tag library. `<logic:present>` checks to see if a bean whose name is specified with the name attribute exists. If a user has previously entered a name and submitted it without any errors, then a bean by that name will exist. Whenever the bean is found, the information in the enclosed by the `<logic:present>` tags are included in the html that is generated. Don't worry about this for now; the next chapter will discuss handling conditional logic in Struts.

The `<bean:write>` tag is used to display the value of the property in the bean specified by the "property" and "name" attributes.

Place the following inside the `index.jsp` file:

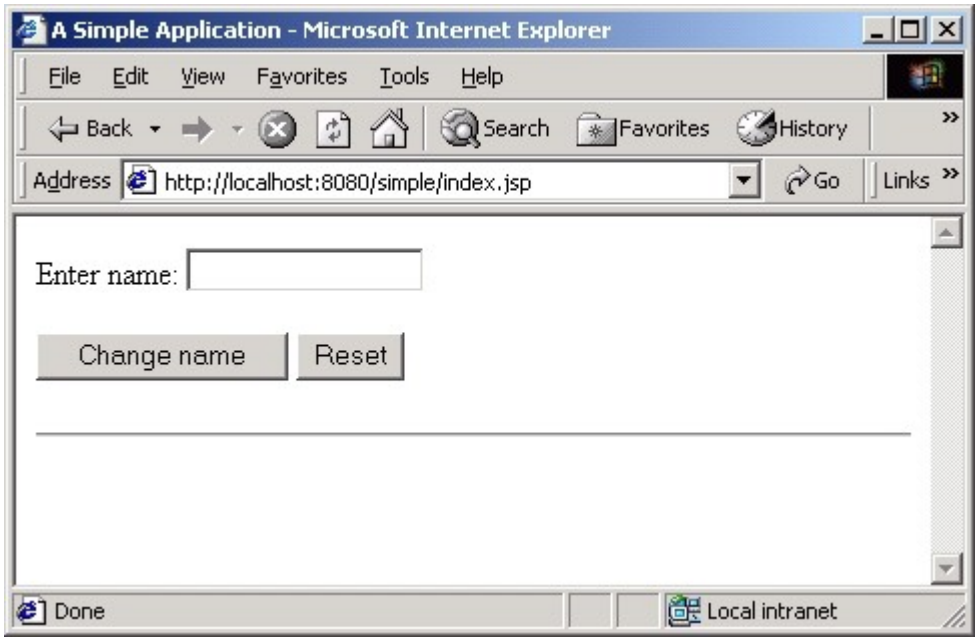
code/Chapter4/SimpleStruts/index.jsp

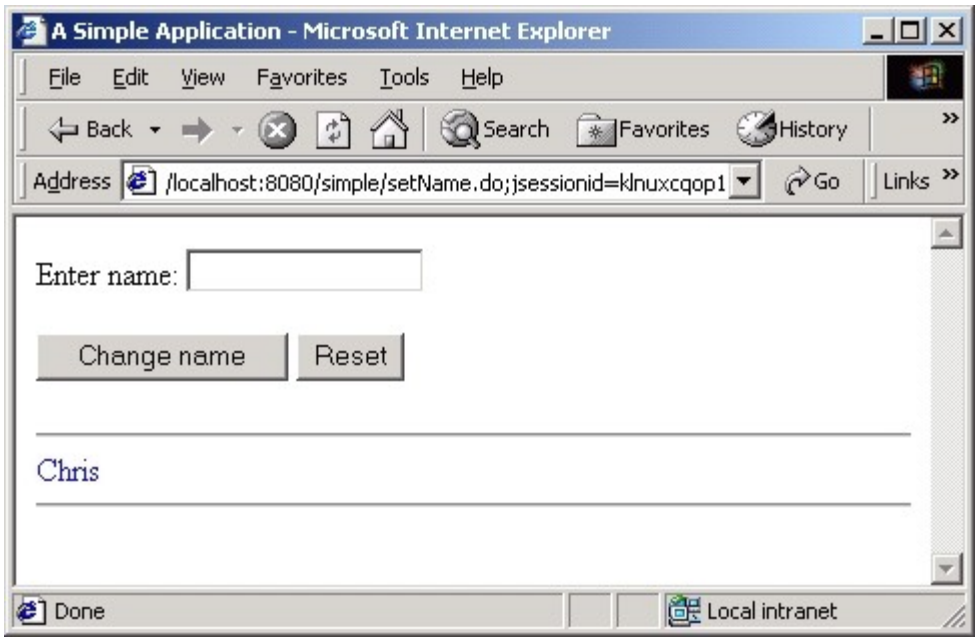
```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/app.tld" prefix="app" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>

<html:html>
  <head>
    <title>
      <bean:message key="welcome.title"/>
    </title>
    <html:base/>
  </head>
  <body>
    <html:errors/>
    <html:form action="/setName">
      <html:hidden property="action"/>
```

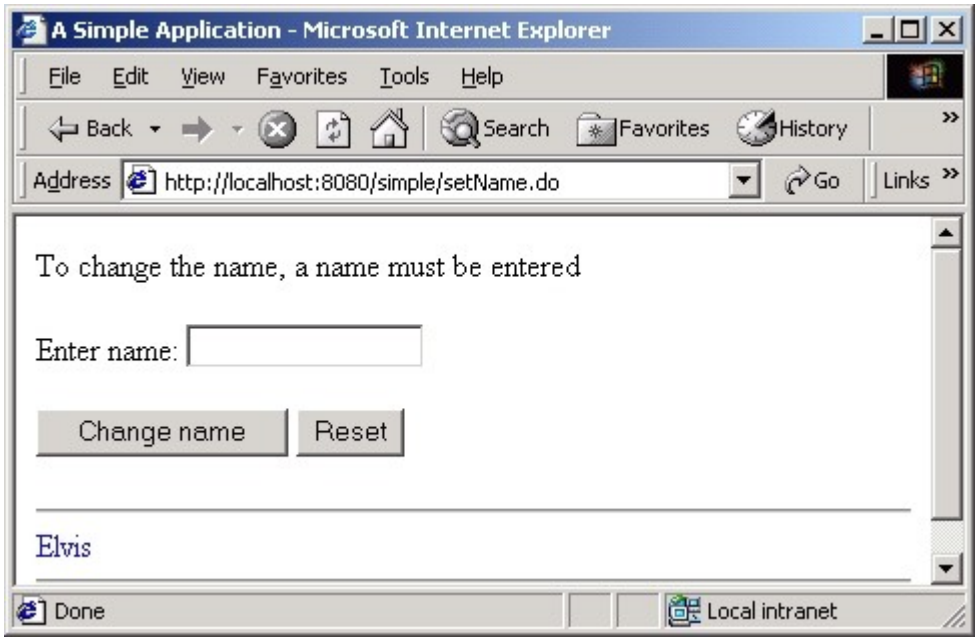
code/Chapter4/SimpleStruts/index.jsp

```
      <bean:message key="prompt.name"/>
      <html:text property="name" size="16" maxlength="16"/>
      <br></br>
      <html:submit>
        <bean:message key="button.save"/>
      </html:submit>
      <html:reset>
        <bean:message key="button.reset"/>
      </html:reset>
    </html:form>
    <hr></hr>
    <logic:present name="name">
      <font color="blue">
        <bean:write name="name" property="name"/>
      </font>
      <hr></hr>
    </logic:present>
  </body>
</html:html>
```

13. `ant all` to compile, jar, and deploy.
14. Go to `http://localhost:8080/SimpleStruts/index.jsp` to test your application







Exercises

Exercise 1. Extending our form

Add the following to the form and appropriate methods and variables to the Name, SetNameAction, and NameForm classes to process the information correctly:

- a set of radio buttons for selection of contentness (happy or sad)

- a set of checkboxes to allow the user to select favorite types of music (rock, gospel, folk, country, and classical)
- a selection to choose an age range (18-24, 25-40, 41-67, 67-300)
- a text-area for entering side notes about the person (5 rows and 40 columns)

Exercise 2. Think

What happens when the browser is closed and the site is visited once again? How can this be avoided?
