

# 3

## *Building a simple application*

---

### ***This chapter covers***

- Creating a simple application
- Extending an application
- Changing an application
- Using JavaServer Pages in an application

*A human being should be able to change a diaper, plan an invasion, butcher a hog, conn a ship, design a building, write a sonnet, balance accounts, build a wall, set a bone, comfort the dying, take orders, give orders, cooperate, act alone, solve equations, analyze a new problem, pitch manure, program a computer, cook a tasty meal, fight efficiently, die gallantly. Specialization is for insects.*

—Excerpt from the notebooks of Lazarus Long,  
from Robert Heinlein's *Time Enough for Love*

### 3.1 Strut by Strut

---

Today, teams write many web applications. Using a layered architecture [POSA] for your application, as described in chapters 1 and 2, can make it easier for team members to specialize and work on different parts of an application. But it is still useful for everyone to understand the entire process from beginning to end. Before getting into the details of how Struts widgets are ratcheted, let's put together a simple but useful program from square one. In this chapter, we say hello to Struts by touring, dissecting, and then constructing an application for logging users in and out.

While not quite as trivial as the exercise in chapter 1, we will still keep it simple for now. A practical application is presented in part 4.

In this chapter, we walk through a classic logon application from a user's perspective. The exercise in chapter 1 compared the passwords entered into a registration form. Depending on whether the entries matched, control branched to one page or another. This application lets you use the accounts created by the chapter 1 exercise to actually log in. The control flow and page content change depending on your status.

After introducing the application, we break it down and zoom in on each component. If you have Struts installed on your development machine, you are welcome to follow along if you like. But if you are leaning back in your armchair, sipping cappuccino, that works too.

Then, with the groundwork laid, we step through constructing the application. Each piece is presented as you might write it yourself, in the order you might write it (less much of the tedious *re*writing). If you are working at your terminal, you could enter the source as we go. If not, every detail is presented so you can follow along from the book alone.

### 3.1.1 Why a logon application?

Our sample program allows a user to log in to the application. Once the user has logged in, the pages change to reflect that the user is now authorized. Typically this is the first step to a larger application that enables authorized users to do something interesting. But for our purposes, just logging in a user is enough to show you how a Struts application actually works.

As shown in table 3.1, we chose doing just a logon application since the process is well understood, simple, self-contained, and needed by most applications.

**Table 3.1** Why we chose a logon application

Reason	Explanation
Well understood	Most of us have logged in to our share of applications, so the process is well understood.
Simple and self-contained	A sample application that accepts a user logon can be simple to write and can also be self-contained. It does not require a complicated model.
Needed by many applications	Most of us will eventually need to write an application that uses some type of logon workflow, so this is code we can use.

## 3.2 Touring a logon application

To begin our tour, we first discuss the scope of the logon application and how you can follow along. We then look at the screens used by the application and note how they change after you've logged in. After we conclude our nickel tour, we go back and take a peek under the hood.

### 3.2.1 Start here

The purpose of our logon application is to give you a look at the nuts and bolts of a Struts application. To help us stay on track, this application contains only the components needed to demonstrate the framework. It contains no real business logic, unit tests, or fancy dialog boxes. Such things are important elements of a shipping application, but we need to walk before we can run.

The logon application is also intentionally stark. It contains no HTML chrome designed to please the eye—just the raw functionality we need to accept a logon. Of course, *your* Struts applications can be as pretty as *you* please.

If you are interested in running the application on your own machine as we go, look for the logon application on the book site's download page [Husted]. This is

packaged as a WAR ready to autodeploy. (See chapter 1 if you don't know how to autodeploy a web application.)

Having the application open is not required but can be interesting at some points. Everything you need to follow along is printed in the chapter.

First, let's tour the screens from the user's viewpoint. Then, we can go back and walk through the actual code.

### 3.2.2 Screens we'll see

As shown in table 3.2, our logon application has two screens: welcome and logon.

**Table 3.2** Logon application screens

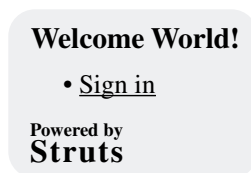
Screen	Purpose
Welcome	Greets visitor and offers links into the application
Logon	Allows input of username and password

If you are following along and have the application deployed on your local machine, you can reach the welcome page with your browser by opening:

```
http://localhost:8080/logon/
```

### 3.2.3 The welcome screen

The first time you visit the welcome screen, there will be only one link, which reads, "Sign in" (see figure 3.1). If you click this link, the logon screen will appear.

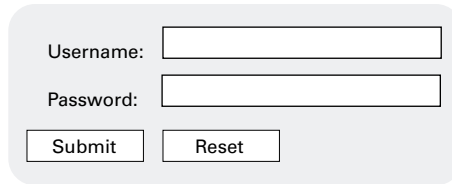


**Figure 3.1**  
The welcome screen of our logon application

### 3.2.4 The logon screen

The logon screen submits the username and password, as you can see in figure 3.2. To see the logon form in action, try clicking Submit without entering anything. If you do, the logon screen returns but with a message, like the one shown in figure 3.3.

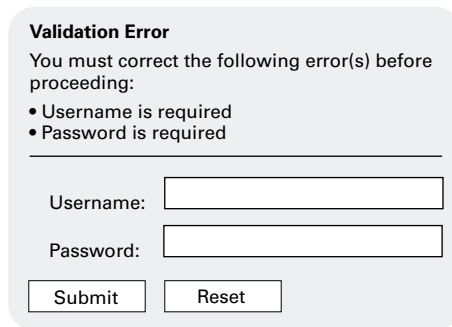
If you enter a username but forget the password and click Submit, the message changes to the one shown in figure 3.4.



Username:

Password:

**Figure 3.2**  
The logon screen



**Validation Error**  
You must correct the following error(s) before proceeding:

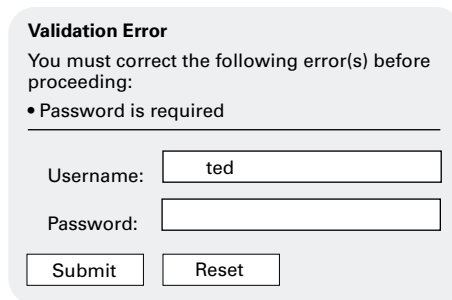
- Username is required
- Password is required

---

Username:

Password:

**Figure 3.3**  
The logon screen tells you the password and username are missing.



**Validation Error**  
You must correct the following error(s) before proceeding:

- Password is required

---

Username:

Password:

**Figure 3.4**  
The logon screen reminds you that you must enter the password.

Here are the important things to note about this workflow, from a user's viewpoint:

- It tells the user everything that is missing all at once.
- When the user submits one thing, it reminds the user only about the other thing.
- It redisplayes on the screen what the user has entered so far, without asking the user to press the Back key.
- If the user manages to enter both a username and a password, the form is accepted and the next screen is displayed.

The logon application validates the logons against a Properties file, like the one used with the registration application from chapter 1. If you download the logon

application from the book's web site [Husted], you can log on using the names of any of the book's authors, as shown in table 3.3.

**Table 3.3** Default logons

Username (or userid)	Password
Ted	Husted
Cedric	Dumoulin
George	Franciscus
David	Winterfeldt
Craig	McClanahan

---

**NOTE** The passwords are case sensitive, so be sure to use an initial capital letter.

---

### 3.2.5 The welcome screen, again

After a successful login, the welcome screen displays again—but with two differences, as you can see in figure 3.5.



**Figure 3.5**  
The welcome screen after  
the user has logged on

First, the screen has been tailored for the user. Instead of just saying “Welcome World!” it now greets the user by name.

In addition, you'll notice that another link has been added. Besides signing in (again), we can now sign out.

### 3.2.6 The welcome screen, good-bye

To close the loop, if we click the sign-out link, we are returned to the original welcome screen—the same screen shown in figure 3.1.

### 3.2.7 Feature roundup

Although simple, our application demonstrates several important techniques:

- Writing links
- Writing forms
- Validating input
- Displaying error messages
- Repopulating forms
- Displaying alternative content

While not as obvious, it also demonstrates:

- Referencing images from dynamic pages
- Rewriting hyperlinks

In the next section, we look at the source code for the application to show how the core features are implemented.

## 3.3 Dissecting the logon application

---

Now that we've said hello to our Struts logon application, let's wander back and take a closer look. We now present the code for each page, along with its related components, and explore what each piece does. After we introduce all the widgets, we show how you can assemble the application from scratch.

### 3.3.1 The browser source for the welcome screen

As you will recall, our application opens with the welcome screen. Let's have a peek at the browser source for the welcome page—just to see what's there (see listing 3.1). The part in bold is what prints on the screen.

**Listing 3.1** The browser source for our welcome page

```
<HTML>
<HEAD>
<TITLE>Welcome World!!</TITLE>
<base href="http://localhost:8080/logon/pages/Welcome.jsp">
</HEAD>
<BODY>
<H3>Welcome World!</H3>
<UL>
<LI><a href="/logon/logon.do">Sign in</a></LI>
</UL>
```

```
<IMG src='struts-power.gif' alt='Powered by Struts'>
</BODY>
</HTML>
```

If you are new to web applications, an important thing to note is that there is nothing here but standard HTML. In fact, there can *never* be anything in a web page but the usual markup that browsers understand. All web applications are constrained to the limitations of HTML and cannot do anything that you can't do with HTML. Struts makes it easier to get Velocity templates, JSP, and other systems to write the HTML we want, but everything has to done with markup the browsers understand.

### **The *jsessionid* key**

There may be one thing in the browser source that you might not recognize as standard HTML. The first time you visit this page, the sign-in link may actually look like this:

```
<LI><a href="/logon.do;jsessionid=aa6XkGuY8qc">Sign in</a></LI>
```

Most web applications need to keep track of the people using the application. HTTP has some rudimentary support for maintaining a user logon, but the approach is limited and not secure. The Java servlet framework does provide support for a robust user session but needs a mechanism to maintain the session across HTTP.

The *jsessionid* is a key maintained by the container to track the user session via HTTP. Including the session key in a hyperlink is called *URL rewriting*. The Servlet Specification [Sun, JST] encourages the use of cookies to maintain the session. When that is not possible, URL rewriting is used instead. The first time a browser makes a request to the container, the container does not know whether the browser will accept a cookie. The container can offer the browser a cookie, but can't tell if it was accepted until the next time a request is made. (HTTP has no "handshaking.") In the meantime, the response for the current request must be written. So, the first page written for a browser will *always* need to use URL rewriting. If on subsequent requests the container finds that its cookie was accepted, it can skip rewriting the URLs.

### **3.3.2 The JSP source for the welcome screen**

Now let's peek at the JSP source that generated the page shown in figure 3.1. The JSP tags appear in bold in listing 3.2.

**Listing 3.2** The JSP source for our welcome page (/pages/Welcome.jsp)

```

<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/struts-logic" prefix="logic" %>
<HTML>
<HEAD>
<TITLE>Welcome World!!</TITLE>
<html:base/>
</HEAD>
<BODY>
<logic:present name="user">
<H3>Welcome <bean:write name="user" property="username"/>!</H3>
</logic:present>
<logic:notPresent scope="session" name="user">
<H3>Welcome World!</H3>
</logic:notPresent>
<html:errors/>
<UL>
<LI><html:link forward="logon">Sign in</html:link></LI>
<logic:present name="user">
<LI><html:link forward="logoff">Sign out</html:link></LI>
</logic:present>
</UL>
<IMG src='struts-power.gif' alt='Powered by Struts'>
</BODY>
</HTML>

```

Now let's take a look at what the lines in bold do:

```

<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/struts-logic" prefix="logic" %>

```

These are the JSP equivalents to import statements and make the tag extensions available to the rest of the page. The code

```

<html:base/>

```

generates a standard HTML base tag, so that references to such things as images can be relative to the location of the original JSP page. You may have noticed that the logon application sometimes refers to .do pages. These aren't actual files on the server but references to Java classes, or Actions, written by the application developer. These Actions then forward to a JSP that creates the response.

JSPs often include references to HTML resources such as images and style sheets. The most convenient way to refer to these resources is through paths that are relative to the JSP template. But when the Action forwards control, it does so

without alerting the browser. If the browser is given any relative references, it will resolve them according to the Action URI, not the location of the JSP template.

Depending on when you access the welcome page, its “location” is shown by the browser:

- `http://localhost:8080/logon/`
- `http://localhost:8080/logon/LogonSubmit.do`
- `http://localhost:8080/logon/Logoff.do`

This is a common problem for dynamic applications. The HTML specification [W3C, HTML] provides the `base` tag as a solution. Struts provides a companion `html-base` tag that inserts the location of the JSP. If you look at the HTML source for the logon page for each of its apparent locations, you will see that in every case the `base` tag renders as:

```
<base href="http://localhost:8080/logon/pages/Welcome.jsp">
```

This lets the browser find the “Powered by Struts” image, which is also stored in the pages folder.

Now let’s take a look at this code:

```
<logic:present name="user">
  <H3>Welcome <bean:write name="user" property="username"/>!</H3>
</logic:present>
```

You’ll remember that the welcome page customizes itself depending on whether the user is logged in. This segment looks to see if we have stored a “user” bean in the client’s session. If such a bean is present, then the user is welcomed by name.

The following code shows why maintaining the user’s session is so important (see section 3.3.1). Happily, the Struts tags and servlet container cooperate to maintain the session automatically (regardless of whether the browser is set to use cookies). To the developer, it feels as if the session has been built into HTTP—which is what frameworks are all about. Frameworks extend the underlying environment so developers can focus on higher-level tasks:

```
<logic:notPresent scope="session" name="user">
  <H3>Welcome!</H3>
</logic:notPresent>
```

Conversely, if the user bean is not present, then we use a generic welcome. All of the Struts logic tags use “this” and “notThis” forms. Else tags are not provided. While this means that some tests need to be repeated, it simplifies the overall syntax and implementation of tags. Of course, other tag extensions can also be used;

you are not constrained to what is offered in the Struts distribution. Several contributor tag extensions are listed on the Struts resource page [ASE, Struts], even one with an if/then/else syntax, if you prefer to use that instead.

As mentioned in section 3.3.1, Struts automatically rewrites hyperlinks to maintain the user session. It also lets you give links a logical name and then store the actual links in a configuration file. This is like referring to a database record with a key. The name and address in the record can change as needed. Other tables will find the updated version using the key. In this case:

```
<LI><html:link forward="logon">Sign in</html:link></LI>
```

we are using `logon` as the key for a record that stores the hyperlink to use for logging on. If we need to change that link later, we can change it once in the configuration file. The pages will start using the new link when they are next rendered.

This code combines the `<logic:present>` and the `<html:link>` tags to display the logoff link only when the user is already logged in:

```
<logic:present name="user">
<LI><html:link forward="logoff">Sign out</html:link></LI>
</logic:present>
```

### 3.3.3 The configuration source for the welcome screen

Struts uses a configuration file to define several things about your application, including the logical names for hyperlinks. This is an XML document that Struts reads at startup and uses to create a database of objects. Various Struts components refer to this database to provide the framework's services. The default name of the configuration file is `struts-config.xml`.

Since the configuration file is used by several different components, presenting the configuration all at once would be getting ahead of ourselves. For now, we'll provide the relevant portions as we go. Later, when we build the application from scratch, we'll present the configuration file in its entirety.

In the initial welcome screen, we refer to a `logon` forward. This is defined in the Struts configuration as such:

```
<forward
  name="logon"
  path="/Logon.do"/>
```

Here, `logon` is a key that is used to look up the actual path for the hyperlink. A Struts action is referenced here, but the path could just as easily refer to a JSP page, Velocity template, HTML page, or any resource with a URI [W3C, URI].

---

**DEFINITION** A *uniform resource identifier (URI)* is a short string that identifies a resource on the Internet or other computer network. A resource could be a document, image, downloadable file, or electronic mailbox, among other things. A URI may correspond to a path on a server's file system but is often an alias. Many URIs in a Struts application are aliases for Java classes or Actions.

---

Since the path is defined in the configuration file, you can change your mind at any time without touching the JSP source. If you update and reload the configuration, the change will be reflected when pages next render.

### 3.3.4 The browser source for the logon screen

If we follow the sign-in link on the welcome page, it brings us to the logon screen, shown earlier in figure 3.2. Listing 3.3 shows the browser source code for this screen. Again, the part in bold is what prints on the screen.

**Listing 3.3** The browser source for our logon screen

```
<HTML>
<HEAD>
<TITLE>Sign in, Please!</TITLE>
</HEAD>
<BODY>
<form name="logonForm" method="POST" action="/logon/LogonSubmit.do">
<TABLE border="0" width="100%">
<TR>
<TH align="right">Username:</TH>
<TD align="left"><input type="text" name="username" value=""></TD>
</TR>
<TR>
<TH align="right">Password:</TH>
<TD align="left"><input type="password" name="password" value=""></TD>
</TR>
<TR>
<TD align="right"><input type="submit" name="submit" value="Submit"></TD>
<TD align="left"><input type="reset" name="reset" value="Reset"></TD>
</TR>
</TABLE>
</form>
<script language="JavaScript" type="text/javascript">
  <!--
    document.forms["logonForm"].elements["username"].focus()
  // -->
</script>
</BODY>
</HTML>
```

Listing 3.4 shows the corresponding JSP source.

**Listing 3.4 The JSP source for our logon screen (/pages/logon.jsp)**

```
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<HTML>
<HEAD>
<TITLE>Sign in, Please!</TITLE>
</HEAD>
<BODY>
<html:errors/>
<html:form action="/LogonSubmit" focus="username">
<TABLE border="0" width="100%">
<TR>
<TH align="right">Username:</TH>
<TD align="left"><html:text property="username"/></TD>
</TR>
<TR>
<TH align="right">Password:</TH>
<TD align="left"><html:password property="password"/></TD>
</TR>
<TR>
<TD align="right"><html:submit/></TD>
<TD align="left"><html:reset/></TD>
</TR>
</TABLE>
</html:form>
</BODY>
</HTML>
```

Let's step through each block here as we did with the welcome page. First, as before, this code makes the Struts `html` tag extension available to the rest of the page:

```
<%@ taglib uri="/tags/struts-html" prefix="html" %>
```

Like the Struts actions, the taglib URI is a logical reference. The location of the tag library descriptor (TLD) is given in `web.xml`.

You'll remember that if we tried to submit the form without entering a logon, an error message displayed. The following tag renders the error messages. When there are no messages, the tag outputs nothing and disappears from the output page:

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

The `<html:form>` tag produces an HTML form for data entry. It also generates a simple JavaScript to move the focus to the first field on the form. The `action` property is a reference to an `ActionMapping` in the Struts configuration. This tells

the form which JavaBean helper to use to populate the HTML controls. The JavaBean helpers are based on a Struts framework class, ActionForm:

```
<html:form action="/LogonSubmit" focus="username">
```

The `<html:text>` tag creates an HTML input control for a text field. It will also populate the field with the `username` property of the JavaBean helper for this form:

```
<TR><TH align="right">Username: </TH><TD align="left">
  <html:text property="username"/></TD>
```

So, if the form were being returned for validation, and the last username submitted was Ted, the tag would then output:

```
<input type="text" name="username" value="Ted">
```

Otherwise, the tag would use the initial default value for `username` as specified by the JavaBean helper class. Usually, this is `null`, but it could be any value.

Likewise, the `<html:password>` tag creates an HTML input control:

```
<TR><TH align="right">Password: </TH>
<TD align="left"><html:password property="password"/></TD>
```

The password control is like a text field but displays asterisks instead of the characters input. If the form is being returned for validation, by default the password tag will rewrite the prior value so that it doesn't need to be entered again. If you would prefer that the password be input each time, you can turn off `redisplay`.

If the initial logon attempt fails, this code keeps the password out of the browser's cache and require the password to be input again, even if it passed validation:

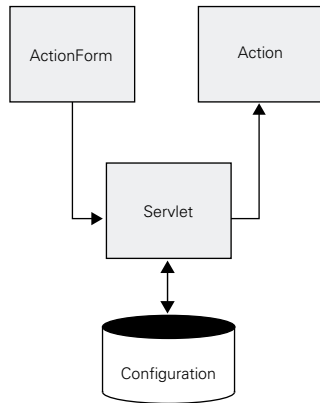
```
<html:password property="password" redisplay="false"/>
```

These tags create standard HTML Submit and Reset buttons:

```
<TD align="right"><html:submit/></TD>
<TD align="left"><html:reset/></TD>
```

When the form is submitted, two framework objects come into play: the ActionForm and the Action. Both of these objects must be created by the developer to include the details for their application. As shown in figure 3.6, the ActionServlet uses the Struts configuration to decide which ActionForm or Action subclass to use.

Let's look at the Struts configuration for the logon screen's ActionForm and Action. Then we can look at the source for these classes.



**Figure 3.6**  
The configuration determines which ActionForm and Action to use.

### 3.3.5 The configuration source for the logon screen

The logon screen itself refers to only one element in the Struts configuration file: the `/LogonSubmit` ActionMapping. This element in turn refers to two other objects, `app.LogonForm` and `app.LogonAction`. All three elements are shown in table 3.4. Let's explore each element in turn.

**Table 3.4** The logon screen configuration elements

Element	Description
<code>/LogonSubmit</code> ActionMapping	Encapsulates several details needed when building and submitting an HTML form with the Struts framework
<code>app.LogonForm</code>	Describes properties used by the HTML form
<code>app.LogonAction</code>	Completes the processing of the submitted form

### 3.3.6 The LogonSubmit source

In the previous section, we mentioned that the `<html:form>` tag works closely with the Struts configuration to make HTML forms more useful:

```
<html:form action="/LogonSubmit" focus="username">
```

The action parameter tells the `<html:form>` tag which ActionMapping to use. In this case, the Struts configuration mapping would look like this:

```
<action
  path="/LogonSubmit"
  type="app.LogonAction"
  name="logonForm">
```

```

scope="request"
validate="true"
input="/pages/Logon.jsp"/>

```

Table 3.5 provides an index to what the settings on this mapping mean.

**Table 3.5** ActionMapping settings

Property	Purpose
path	A unique identifier for this mapping. It is included in the web address, as in <i>http://localhost:8080/logon/LogonSubmit.do</i> .
type	The Action object to call when the path is requested.
name	The JavaBean helper (ActionForm) to use with an HTML form.
scope	A property that specifies whether to store the helper in the request or the session.
validate	A property that specifies whether to call the standard <code>validate</code> method on the form bean helper (specified by <b>name</b> ) before calling the Action object (specified by <b>type</b> ).
input	A property that specifies where to send control if the <code>validate</code> method returns false.

As we mentioned in chapter 2, many of the object and property names used by the Struts framework are vague. For example, the `name` property is *not* the name of the mapping; it's the name of the JavaBean helper, or ActionForm bean, to be used with this mapping.

The same form beans are also specified in the configuration:

```

<form-bean
  name="logonForm"
  type="app.LogonForm"/>

```

This element relates the logical name `logonForm` with a specific Java class, `app.LogonForm`. This will be a subclass of a Struts ActionForm. The ActionForm class provides standard methods for the framework to use, including the `validate` method.

Let's take a look at the source for the `LogonForm` and then come back to the `LogonAction`.

### 3.3.7 The LogonForm source

While HTML forms give users a place to enter data, they do not give applications a place to put it. When the user clicks Submit, the browser collects the data in the form and sends it up to the server as a list of *name-values* pairs (or couplets). So, if a user enters a username and a password into the logon page and clicks Submit, this is what our application sees:

```
username=Ted  
password=LetMeIn
```

The browser submits everything as a string of characters. You can put in JavaScript validations to force people to enter only numerals into a given field or to use a prescribed format for dates, but that's just smoke and mirrors. Everything is still going to be transferred to your application as a string—*not* as a binary object ready to pass to a Java method.

It's important to remember that this is the way the browsers and HTML work. Web applications cannot control this. Frameworks such as Struts exist to make the best of what we have to work with. The Struts solution to HTTP data-entry snarls is the `ActionForm`.

In an environment like Swing, data-entry controls have a built-in text buffer that can validate characters as they are entered. When the user leaves the control, the buffer can be converted to a binary type, ready for delivery to the business layer.

Unfortunately, the HTTP/HTML platform doesn't provide a component that can buffer, validate, and convert input. So, the Struts framework offers the `ActionForm` (`org.apache.struts.action.ActionForm`) to bridge the gap between web browser and business object. `ActionForms` provide the missing buffer/validate/convert mechanism we need to ensure that users enter what they are supposed to enter.

When an HTML form is submitted, the name-value couplets are caught by the Struts controller and applied to an `ActionForm`. The `ActionForm` is a `JavaBean` with properties that correspond to the controls on an HTML form. Struts compares the names of the `ActionForm` properties with the names of the incoming couplets. When they match, the controller sets the property to the value of the corresponding couplet. Extra properties are ignored. Missing properties retain their default value (usually null or false).

Here are the public properties from our `LogonForm`:

```
private String password = null;  
public String getPassword() {  
    return (this.password);  
}  
public void setPassword(String password) {  
    this.password = password;  
}  
  
private String username = null;  
public String getUsername() {  
    return (this.username);  
}
```

```
public void setUsername(String username) {
    this.username = username;
}
```

The properties of most Struts ActionForms look just like this. Thrifty developers can create them with a macro that simply prompts them for the property name. Others may use code skeletons and the search-and-replace feature of their code editors. Struts code generators are also available that create ActionForms by parsing HTML or JSPs.

---

**NOTE** In Struts 1.1, creating ActionForms is even simpler if you use a Dynamic ActionForm or Map-backed ActionForm. See chapter 5 for details.

---

The base ActionForm also includes two standard methods—`reset` and `validate`. The `reset` method is helpful when you are using ActionForms as part of a wizard workflow. This method doesn't need to be implemented if the mapping is set to request scope.

When the mapping is set to `validate=true`, the `validate` method is called after the form is populated from the HTTP request. The `validate` method is most often used as a *prima facie* validation. It just checks that the data “looks” correct and that all required fields have been submitted. Again, this is something that a Swing control would do internally before passing the data along to the application. You can do these checks by hand, or use something like the *ValidatorForm* (see chapter 12), which can be programmed from a configuration file.

Here's the `validate` method from our LogonForm. It checks that both fields have something entered into them. If your application had any rules regarding the length of a username or password, you could enforce those rules here as well.

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

    if ((username == null) || (username.length() < 1))
        errors.add("username",
            new ActionError("error.username.required"));

    if ((password == null) || (password.length() < 1))
        errors.add("password",
            new ActionError("error.password.required"));

    return errors;
}
```

The `ActionErrors` object returned by `validate` is another framework class. If `validate` does not return null, then the controller will save the `ActionErrors` object in the request context under a known key. The `<html:errors>` tag knows the key and will render the error messages when they exist, or do nothing when they do not.

The tokens `error.username.required` and `error.password.required` are also keys. They are used to look up the actual messages from the Struts message resources file. Each locale can have its own resource file, which makes the messages easy to localize.

The Struts message resources file uses the common name-value format. The entries for our messages look like this:

```
error.username.required=<li>Username is required</li>
error.password.required=<li>Password is required</li>
```

---

**NOTE** In Struts 1.1 there are ways to keep the markup out of the messages. A new `errors.prefix/error.suffix` feature can be used to specify that `<li>` and `</li>` should wrap each message. A new set of message tags is also available that can be used in place of the original `<html:error>` tag. The message tags make it easy to keep the markup in the presentation page (where it belongs). See chapter 10 for more about the Struts JSP tags.

---

Even when localization is not being used, the Struts application resource file collects all the messages into a single place where they can be reviewed and revised, without touching the Java source code.

### 3.3.8 The LogonAction source

After collecting the data entry into an `ActionForm` and performing any initial validations, the controller passes the form along to the Action class given by the mapping.

The Struts architecture expects that you will use your own Java classes to do most of the request processing. A JSP page may *render* the result, but the Action *obtains* the result. As you saw in chapter 2, this is known as an MVC or Model 2 approach, where the Action serves as a request dispatcher.

When a request for an Action is sent to the Struts servlet, it invokes (or dispatches) the Action by calling its `perform` (or `execute`) method.

---

**NOTE** There is an alternative entry method in Struts 1.1, named `execute`. This method provides for better exception handling but is otherwise the same as the Struts 1.0 `perform` method. We will refer to the `perform` method in this chapter so the code will work with both versions. Other applications in the book are based on Struts 1.1 and make good use of the new features.

---

Listing 3.5 contains the source in its entirety.

**Listing 3.5** The Java source for the `LogonAction` class  
(`/WEB-INF/src/java/app/LogonAction.java`)

```
package app;

import java.io.IOException;
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;

public final class LogonAction extends Action {

    // Validate credentials with business tier
    public boolean isUserLogon (String username,
                               String password) throws UserDirectoryException {

        return (UserDirectory.getInstance().
                isValidPassword(username,password));

    } // end isUserLogon

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

        // Obtain username and password from web tier
        String username = ((LogonForm) form).getUsername();
        String password = ((LogonForm) form).getPassword();

        // Validate credentials
        boolean validated = false;
        try {
            validated = isUserLogon(username,password);
        }
    }
}
```

```

}
catch (UserDirectoryException ude) {
// couldn't connect to user directory
    ActionErrors errors = new ActionErrors();
    errors.add (ActionErrors.GLOBAL_ERROR,
        new ActionError("error.logon.connect"));
    saveErrors(request,errors);
    // return to input page
    return (new ActionForward (mapping.getInput()));
}

// Save our logged-in user in the session,
// because we use it again later.
HttpSession session = request.getSession();
session.setAttribute(Constants.USER_KEY, form);

// Log this event, if appropriate
if (servlet.getDebug() >= Constants.DEBUG) {
    StringBuffer message =
        new StringBuffer("LogonAction: User ");
    message.append(username);
    message.append("' logged on in session ");
    message.append(session.getId());
    servlet.log(message.toString);
}

// Return success
return (mapping.findForward (Constants.WELCOME));

} // end perform
} // end LogonAction

```

And now the blow by blow.

The Action is at the top of the Struts food chain and so imports several classes. We've specified each class used here, so that you can see where everything comes from:

```

package app;
import java.io.IOException;
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;

```

If we were lazy, this block could also be expressed as:

```
package app;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.struts.action.*;
```

But that would not be as instructive. Like most Apache products, the Struts source code follows best practices and doesn't cut corners. We follow suit in our own code. While we've omitted the JavaDoc here, both our code and the Struts code are fully documented.

Next, we use a helper method to call a business tier method. We could have put this same code into the Action's `perform` method, but it is always a good idea to strongly separate the generic business code from the Struts controller code. If you let even one line in, that soon turns into three, then five, and before long your Actions are a big ball of mud [Foote]. The best way to avoid "code creep" is to always encapsulate business tier code in a helper method before calling it from an Action:

```
// Validate credentials with business tier
public boolean isUserLogon (String username,
    String password) throws UserDirectoryException {

    return (UserDirectory.getInstance().
        isValidPassword(username,password));

} // end isUserLogon
```

As we've mentioned elsewhere, Struts 1.1 prefers the new `execute` method over the original `perform` method, but either one still works. We used `perform` in this application so the code will work with either version:

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

The purpose of an Action is to ferry input from the web tier to the business tier, where the rest of your application lives. Here, we extract the username and password from the ActionForm (JavaBean helper) and save them as plain Strings:

```
// Obtain username and password from web tier
String username = ((LogonForm) form).getUsername();
String password = ((LogonForm) form).getPassword();
```

We can then pass the username and password Strings to a business tier function to see if they validate. Here, we take care to encapsulate the call in a separate method and not bury the code in the Action's perform method. This call is to another method in this class, but it could just as easily be to any method in any Java class:

```
// Validate credentials
boolean validated = false;
try {
    validated = isUserLogon(username,password);
}
catch (UserDirectoryException ude) {
// couldn't connect to user directory
    ActionErrors errors = new ActionErrors();
    errors.add (ActionErrors.GLOBAL_ERROR,
    new ActionError("error.logon.connect"));
    saveErrors(request,errors);
    // return to input page
    return (new ActionForward (mapping.getInput()));
}
}
```

The API for the `isUserLogon` method specifies that it return true if the credentials match, false if they don't, and throw an exception if it doesn't know (say, for instance, because it couldn't connect to the directory to find out). Should the exception occur, our Action catches it, converts the event into an `ActionError`, and forwards back to the input page.

If the business tier comes back and says the logon is no good, the Action posts the error message and routes control back to the input page. This is the same thing that happens when the `validate` method on the `ActionForm` fails (see section 3.3.7):

```
if (!validated) {
// post the error
    ActionErrors errors = new ActionErrors();
    errors.add (ActionErrors.GLOBAL_ERROR,
        new ActionError("error.logon.invalid"));
    saveErrors(request,errors);
    // return to input page
    return (new ActionForward (mapping.getInput()));
}
}
```

Since the error does not pertain to a particular property, we log the error under the generic `ActionErrors.GLOBAL_ERROR` flag instead of a property name. To indicate that the logon itself is invalid, we also specify a different error message than `validate`. In the Struts application resources, this is shown as:

```
error.logon.invalid=<li>Username/password combination is invalid</li>
```

The presentation layer substitutes the `error.login.invalid` token for the proper message when it is displayed. If there is a separate message for a user's locale, then the user will receive the localized version of this message.

If the business tier says the logon is good, then we can tell the web tier to retain the user's credentials. The Java Servlet framework provides a user session for exactly this purpose. Here we store the user's `logonForm` in their session context. Each user has a context, maintained by the servlet container. Users who have a `logonForm` stored in their session context are logged in. Otherwise, the user is not logged in:

```
// Save our logged-in user in the session,
// because we use it again later.
HttpSession session = request.getSession();
session.setAttribute(Constants.USER_KEY, form);
```

This strategy is known as application-based security. Many Java web applications use this approach, since it is portable and easy to implement. Any approach to authentication can be used in your own applications.

The Struts framework relies on the container's default logging system. Here, we log an event only if the debug level for the servlet was set high enough in the web deployment descriptor (`web.xml`):

```
// Log this event, if appropriate
if (servlet.getDebug() >= Constants.DEBUG) {
    StringBuffer message =
        new StringBuffer("LogonAction: User ");
    message.append(username);
    message.append(" logged on in session ");
    message.append(session.getId());
    servlet.log(message.toString());
}
```

We set this with an `init-param`, like this:

```
<init-param>
  <param-name>debug</param-name>
  <param-value>2</param-value>
</init-param>
```

In a production application, you can set `debug` to 0, and entries like this one won't appear. To plug in another logging package, developers subclass the Struts `ActionServlet` class and override the `log` method.

---

**NOTE** In Struts 1.1, using alternate logging packages is made even easier through support of the Jakarta Commons Logging Component [ASF Commons].

---

When all is said and done, the `perform` method returns an `ActionForward` to the controller (`ActionServlet`). Here, we send control to the success forward:

```
// return success
return (mapping.findForward (Constants.SUCCESS));
}
```

This is defined in the Struts configuration as:

```
<forward
  name="success"
  path="/pages/Welcome.jsp"/>
```

Now that we are logged in, the presentation of this page will vary slightly. A logoff link will be available.

### 3.3.9 The LogoffAction source

Look back to figure 3.5 to see how the welcome page changes once the user is logged in. In `Welcome.jsp`, the `<logic:present>` tag sees the user bean placed into the session context by the `LogonAction`

```
<logic:present name="user">
```

and exposes an `<html:link>` tag that references the `logoff` forward:

```
<html:link forward="logoff">Sign out</html:link>
```

In the Struts configuration, the `logoff` forward is defined as:

```
<forward
  name="logoff"
  path="/logoff.do"/>
```

The path here refers to a `.do` file. There should be a corresponding `/logoff` `ActionMapping` elsewhere in the Struts configuration:

```
<action
  path="/logoff"
  type="app.LogoffAction"/>
```

As you can see, `/logoff` is an extremely simple mapping; it simply passes control to the `LogoffAction` class, without any special parameters or settings. The job of the `LogoffAction` class is also very simple. It just removes the user's `logonForm`

object from the session context. If there is no `logonForm` in the session context, then the user is considered to be logged out. Let's have a look at the `LogoffAction` source, which is the last class remaining in our walkthrough (see listing 3.6).

**Listing 3.6** The Java source for `LogoffAction` class  
(`/WEB-INF/src/java/app/LogoffAction.java`)

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

    // Extract attributes we will need
    HttpSession session = request.getSession();
    LogonForm user = (LogonForm)
        session.getAttribute(Constants.USER_KEY);

    // Log this user off
    if (user != null) {

        if (servlet.getDebug() >= Constants.DEBUG) {
            StringBuffer message =
                new StringBuffer("LogoffAction: User ");
            message.append(user.getUsername());
            message.append("' logged off in session ");
            message.append(session.getId());
            servlet.log(message.toString());
        }
    }
    else {

        if (servlet.getDebug() >= Constants.DEBUG) {
            StringBuffer message =
                new StringBuffer("LogoffAction: User ");
            message.append(session.getId());
            servlet.log(message.toString());
        }
    }

    // Remove user login
    session.removeAttribute(Constants.USER_KEY);

    // Return success
    return (mapping.findForward (Constants.SUCCESS));
}

} // end LogoffAction
```

First, we obtain the user's logon object. The convention of this application is to store the user's logon object in the session context, under the key given by `Constants.USER_KEY`, so that's where we look:

```
// Extract attributes we will need
HttpSession session = request.getSession();
LogonForm user = (LogonForm)

// Log this user off
if (user != null) {

    if (servlet.getDebug() >= Constants.DEBUG) {
        StringBuffer message =
            new StringBuffer("LogoffAction: User ");
        message.append(user.getUsername());
        message.append("' logged off in session ");
        message.append(session.getId());
        servlet.log(message.toString());
    }
}
```

As before, we log some detail if the debug level has been set high enough in the web deployment descriptor (`web.xml`).

This is the core operation in the class. We remove any object stored under the `USER_KEY`, and voila, the user is logged out:

```
// Remove user login
session.removeAttribute(Constants.USER_KEY);
```

If we wanted to remove everything from the session that might be stored for the user, we could simply invalidate the session instead:

```
session.invalidate();
```

But this also destroys objects such as the user's locale, which is used to display localized messages.

As with logon, when the operation completes, we return to the welcome page:

```
// Return success
return (mapping.findForward (Constants.SUCCESS));
```

In the next section, we will step back and show how this application would be built from scratch. So far, we've been stepping through the guts of the pages and classes. Now we will take a wider look at the application by moving the focus to the Struts configuration file and the source code tree.

### 3.4 Constructing an application

---

We've taken the application for a drive, kicked the tires, and taken a good look under the hood. We know what it is supposed to do and how it does it, but where would you start building your own? In this section, we go back to square one and show how you can build an application like this from beginning to end.

Since we have a good grasp of what we need the application to do, we can start with a practical set of requirements. From this, we can create a whiteboard plan that includes the obvious objects and what we will call them. Then, we'll start coding the objects, refining and expanding on the plan as we go. This type of plan/code, refine-plan/refine-code approach to software development is often called a "spiral" methodology.

Of course, there are many other ways to approach software development. Any methodology should work fine with Struts. But the goal of this section is not to explore software methodologies. We're here to demonstrate what it is like to construct a simple Struts application. So, let's have at it...

#### 3.4.1 Defining the requirements

*Requirements are the effects that the computer is to exert in the problem domain, by virtue of the computer's programming*

—*Practical Software Requirements*, by Benjamin L. Kovitz

Although we have a sound working understanding of what the application needs to do, it's always a good practice to start from a set of requirements. Following suit with the rest of this chapter, we'll just draw up the simplest possible set of useful requirements.

Our simple requirements document will have three main sections: goal, requirements, and specification, as shown in table 3.6.

**Table 3.6** Headings in our requirements document

Heading	Purpose
Goal	What result we need to achieve within the problem domain
Domain requirements	What we need to accomplish to realize the goal
Program specifications	What we need to do to realize the requirements

#### **Goal**

- Allow privileged users to identify themselves to the application

**Domain requirements**

- Allow users to present their credentials (username and password)
- Verify that the credentials presented are valid
- Allow correction of invalid credentials
- Inform user when credentials are verified
- Allow validated users access to privileged features
- Allow user to invalidate access on demand

**Program specifications**

- Be accessible from a standard web browser, with or without JavaScript enabled
- Offer logon to new visitors from a welcome page
- Allow entry of credentials (username and password) on a logon page
- Require that each credential contain 1 to 20 characters, inclusive
- Require that both username and password be entered
- Submit credentials to business tier method for validation
- Return invalid credentials to user for correction
- Log on user if credentials are valid
- Customize welcome page with username when logged on
- Allow validated users to log off from welcome page

Of course, this is very simple specification for a very simple application. Many specifications consume reams of paper and are embellished with diagrams, data tables, screen definitions, and detailed descriptions of the problem domain. For more about writing specifications for your own applications, we recommend *Practical Software Requirements*, by Benjamin L. Kovitz [Kovitz].

**3.4.2 Planning the application**

With our requirements in hand, we can start to sketch the application and plan which objects we will need to realize the program specification. One way to approach this is to list the specifications and the components that would help realize them. Often, a team will do something like this on a large whiteboard as part of an initial design meeting.

---

**NOTE** An important point to note in this process is that there is often not a 1:1 correlation between the specification items and the components that realize them. While both the specification and the program serve the same goal, they approach the goal from different perspectives. So, a list like this will not be “normalized.” Some specifications will appear more than once, as will some components.

---

### **View**

In practice, many applications start out as storyboards. JSPs define the visible parts of our application. Table 3.7 outlines our requirements for the presentation layer.

**Table 3.7** Our “whiteboard” view plan

Specification	JavaServer Pages
Offers logon to new visitors from a welcome page	Welcome.jsp
Allows entry of credentials (username and password) on a logon page	Welcome.jsp
Returns invalid credentials to user for correction	Logon.jsp
Customizes welcome page with username when logged on	Welcome.jsp
Allows validated users to log off from welcome page	Welcome.jsp
Is accessible from a standard web browser with or without JavaScript enabled	Logon.jsp; Welcome.jsp
Directs users to welcome activity	index.jsp

Note that we added a specification of our own at the end of this list. A good trick in a Struts application is to have the application’s welcome page redirect to a Struts action. This puts the control flow into the framework as soon as possible and helps to minimize change as the application grows.

### **Controller**

In a strongly layered application (see chapter 2), all requests for pages or data pass through the control layer. Table 3.8 outlines our requirements for the controller (or “Front Controller” [Go3]).

**Table 3.8** Our “whiteboard” controller plan

Specification	ActionForms
Allows entry of credentials (username and password) on a logon page	LogonForm
	<b>Actions</b>
Validates credentials with business tier method	LogonAction
Returns invalid credentials to user for correction	LogonForm; LogonAction
Logs on user if credentials are valid	LogonAction
Allows validated users to log off from welcome page	LogoffAction
	<b>ActionForwards</b>
Offers logon to new visitors from a welcome page	welcome; logon
Allows validated users to log off from welcome page	Logoff
	<b>ActionMappings</b>
Submits credentials to business tier method for validation	LogonSubmit
	<b>Utility</b>
Documents all internal constants	Constants

Note that we added another specification of our own, “Document all internal constants.” This is especially important in a layered application, where some of the constants will be “loosely bound.” The Java compiler can’t validate tokens that we use in an XML configuration file, so it’s important we carefully track whatever tokens we use.

### **Model**

We have only one requirement for our data access layer, shown in table 3.9.

**Table 3.9** Our “whiteboard” model plan

Specification	Method interface
Submits credentials to business tier method for validation	<code>boolean isUserLogon(String username, String password);</code>

### 3.4.3 Planning the source tree

With a baseline plan for the application in place, we can sketch a source tree for the application. Since our application is very simple, we can use a single subdirectory for the pages and a single package for Java classes. Our tree is shown in figure 3.7.

---

**NOTE** Struts expects there to be an Application Resources bundle on your classpath. The logon application places its property file in its own “resources” package. Internationalized applications will have several property files. Giving them their own package helps keep things organized. Our build file copies these to the classes folder so that they will be on the classpath at runtime. Just be sure to rebuild the application after any change to a resource file.

---

If you are following along and building your own logon application, a good way to get a jumpstart on the tree and the Struts classes is to deploy the Blank application:

- Download the Blank application from the book site [Husted].
- Copy the blank.war file as logon.war.
- Put the WAR in your container’s autodeployment folder (usually webapps).

This is why the Blank application is provided. It’s meant as a generic template for other applications. We present the base Blank infrastructure in sections 3.4.4 through 3.4.8. Then, we begin work on the source for our logon application.

To modify and rebuild the application, you may need to install some development tools.

### 3.4.4 Setting up your development tools

Aside from the Java Development Kit and a web container, you need two other pieces to be able to create and deploy web applications: a build tool and a

```

[logon]
index.jsp
|_pages
|_Welcome.jsp
|_Logon.jsp
|_[...]
|_WEB-INF
|_build.xml
|_web.xml
|_conf
|_struts-config.xml
|_[...]
|_doc
|_index.html
|_[...]
|_lib
|_struts.jar
|_struts-bean.tld
|_struts-html.tld
|_struts-logic.tld
|_src
|_java
|_app
|_Constants.java
|_LogoffAction.java
|_LogonAction.java
|_LogonForm.java
|_[...]
|_resources
|_application.properties

```

**Figure 3.7** The source tree for our logon application

programmer's editor. Like many development teams, the Struts Committers use Jakarta's Ant to build the Struts distribution and its sample applications. Other tools could be used, but Ant is quickly becoming the de facto standard build tool for Java applications.

The choice of a programming editor is still a subjective decision. Any programming editor can be used with Struts (and probably is). If you do not have a preference, a likely starting point is the open source programmer's editor, jEdit.

### **Installing Ant**

Deploying applications generally involves many steps. Automating those steps is your best chance that things will go as planned. Ant is an Apache XML scripting tool designed to ensure that deployments are quick and free from human error. Ant uses an XML build file to run a series of tasks. Ant predefines the most common tasks. If the predefined tasks don't meet your needs, you can also create your own.

To install Ant, you need to:

- 1 Download from Jakarta [ASF, Ant].
- 2 Unzip the download to the directory of your choice.
- 3 Set up three environment variables: ANT\_HOME, JAVA\_HOME, and PATH.
- 4 Set ANT\_HOME=<location of the unzipped download>.
- 5 Set JAVA\_HOME=<location of the JDK>.
- 6 Set PATH=%PATH%;%ANT\_HOME%\bin.

As part of constructing our application, we provide a build.xml file to use with Ant.

### **Installing jEdit**

If you do not already have a preferred programming editor, you can download and install jEdit to get started with Struts. The Java-based installer bundled with jEdit makes it as simple to install as the latest versions of the JDK and Tomcat (see chapter 1).

Let's step through the installation process:

- 1 Download jEdit from SourceForge [jEdit].
- 2 If the installer doesn't run automatically, you can start it from a DOS prompt using:  

```
java -jar <downloaded jar file>
```
- 3 The first screen you will see is a welcome screen.

- 4 Clicking Next will present you with the GNU General Public License.
- 5 Clicking Next will provide you with the opportunity to select an installation directory.
- 6 Clicking Next will present you with a list of components to install. We recommend that you select them all.
- 7 Clicking Next again will initiate the install.
- 8 After the installation completes, you can launch jEdit by navigating through the Windows Start menu.

A number of jEdit plug-ins are available that allow you to edit over FTP, build Ant projects, edit XML files, and much more.

#### **3.4.5 Setting up the *build.xml* file**

Like many Java products these days, Struts expects that the Jakarta Ant tool [ASF, Ant] will be used as part of the build process. Ant also uses an XML configuration file, named *build.xml*. Typically, you can set up a stock build file for your application that does not change throughout. In chapter 4, we present the build file used by the logon application.

#### **3.4.6 Setting up the *web.xml* file**

The Java 2 Servlet framework uses a configuration file to help set up your application. The web deployment descriptor, or *web.xml*, identifies the servlets you'll need, and other settings for your application. The format is prescribed by the servlet specification [Sun, JST]. Most Struts applications need to deploy only a single servlet and the tag libraries, and tend to be relatively simple. We present the *web.xml* file used by the logon application in chapter 4.

#### **3.4.7 Setting up the *struts-config.xml* file**

Much like the web deployment descriptor, Struts also has an XML configuration file. This is where your application registers its ActionForms, ActionForwards, and ActionMappings. Each class has its own section in the file where you can define the default objects to be created at startup. Listing 3.7 shows our starter Struts configuration file.

**Listing 3.7** The Struts configuration file (/WEB-INF/conf/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-beans>
    <!-- ... -->
  </form-beans>
  <global-forwards>
    <forward
      name="welcome"
      path=" /Welcome.do"/>
    <!-- ... -->
  </global-forwards>
  <action-mappings>
    <action
      path="/Welcome"
      type="org.apache.struts.actions.ForwardAction"
      parameter="/pages/Welcome.jsp"/>
    <!-- ... -->
  </action-mappings>
</struts-config>
```

When you set up your application, you can start with a blank configuration file, like this one, and add the objects you need as you go along. We'll do just that through the balance of the chapter so you can see how the Struts configuration is used in practice. Chapter 4 covers the Struts configuration files in depth. You may have noticed that our starter configuration is not totally blank. A default `welcome` forward has been provided for your convenience.

### **The welcome action**

Usually, it's helpful to route the page flow through the Struts controller as soon as possible. This keeps the big picture in the Struts configuration. You can adjust the control flow for the entire application from a single point. Unfortunately, the containers require a physical page for the welcome page. Listing a Struts action URI as a welcome page in the web deployment descriptor (`web.xml`) doesn't work.

The best all-around solution is to put in a stub `index.jsp` that redirects to your welcome action. The `struts-blank` provides one such stub. This is a very simple utility page, with just two lines:

```
<%@ taglib uri="/tags/struts-logic" prefix="logic" %>
<logic:redirect forward="welcome"/>
```

The Blank application provides both the `index.jsp` forwarding page and a default welcome page. We will continue to use the `index.jsp` as is but will be making some changes to the welcome page. However, before we do anything else, let's test our deployment.

### 3.4.8 Testing the deployment

To be sure all is well before testing a new application, it's helpful to open a working application as a baseline. The Struts Blank application is a good choice for a baseline application. Its default welcome page includes some basic system checks to see that configuration files are loading properly, that the tag extensions can be found, and that the message resources are available.

The WAR file for the Struts Blank application can be found in the Struts distribution or on this book's website. Just place the `blank.war` file in your container's autodeploy folder and restart it if necessary. You can then open the application's welcome page using a URL such as

```
http://localhost:8080/blank
```

If all is well, a page like the one shown in figure 3.8 should display.

## Welcome!

To get started on your own application, copy the `struts-blank.war` to a new WAR file using the name for your application. Place it in your container's "webapp" folder (or equivalent), and let your container auto-deploy the application. Edit the skeleton configuration files as needed, reload Struts or restart your container, and you are on your way! (You can find the `application.properties` file with this message in the `/Web-INF/SCR/java/resources` folder.)

Powered by  
**Struts**

**Figure 3.8** The welcome screen of the Blank application

The source for this page appears in listing 3.8.

### Listing 3.8 The default welcome page for the Blank application

```
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/struts-logic" prefix="logic" %>
<html:html locale="true">
<head>
<title><bean:message key="welcome.title"/></title>
<html:base/>
</head>
```

```
<body>
<logic:notPresent name="org.apache.struts.action.MESSAGE"
  scope="application">

<b>ERROR: Application resources not loaded -- check servlet container
  logs for error messages.</b>

</logic:notPresent>
<h3><bean:message key="welcome.heading"/></h3>
<p><bean:message key="welcome.message"/></p>
</body>
</html:html>
```

### 3.4.9 Constructing our welcome page

A basic tenet of most software methodologies is to get a working prototype up and running as soon as possible. If we follow that advice, then the first thing we should do is put up the welcome page called for by our specification. An early version of our welcome page, without the conditional logic, might look like the one shown in listing 3.9.

**Listing 3.9** An early version of the welcome page

```
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<html>
<head>
<title>Welcome World!!</title>
<html:base/>
</head>
<body>
<ul>
<li><html:link forward="logon">Sign in</html:link</li>
</ul>
<img src='struts-power.gif' alt='Powered by Struts'>
</body>
</html>
```

Since this refers to the logon ActionForward, we need to add that to our Struts configuration. We can also change the default welcome page from Index.jsp to Welcome.jsp:

```
<global-forwards>
  <forward
    name="logon"
    path "/Logon.do"/>
  </forward>
```

```

        name="welcome"
        path /Welcome.do"/>
    <!-- ... -->
</global-forwards

```

At this point, we can restart the container to reload the Struts configuration. Some containers, like Tomcat, let you reload a single application.

---

**1.0 vs 1.1** In Struts 1.0, there were number of administrative Actions available, including one to reload the Struts configuration. These were removed in Struts 1.1 because they conflicted with the support for multiple application modules.

---

Once the new configuration is loaded, we can try opening our new welcome page:

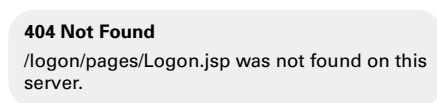
```
http://localhost:8080/logon/
```

You should see the screen shown in figure 3.9.



**Figure 3.9**  
The welcome screen before logon

However, if you were to try and click on the link, you wouldn't get very far, as shown in figure 3.10.



**Figure 3.10**  
The file not found error

To fix this error, we need to move on to the next object and construct the logon page.

### 3.4.10 Constructing the logon page

Looking back at our whiteboard in section 3.4.2, we see that our logon page needs to collect the username and password, and submit them to a mapping named `/LogonSubmit`. This means that we need to create a Struts form that specifies the `/LogonSubmit` action, with input controls for a text field and a password field, as shown in listing 3.10.

**Listing 3.10** The JSP source for our logon page (/pages/logon.jsp)

```
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<html><head><title>Sign in, Please!</title></head>
<body>
<html:errors/>
<html:form action="/LogonSubmit" focus="username">
<table border="0" width="100%">
<tr>
<th align="right">Username: </th>
<td align="left"><html:text property="username"/></td>
</tr>
<tr><th align="right">Password: </th>
<td align="left"><html:password property="password"/></td>
</tr>
<tr>
<td align="right"><html:submit property="submit" value="Submit"/></td>
<td align="left"><html:reset/></td>
</tr>
</table>
</html:form>
</body>
</html>
```

---

The `<html:form>` tag refers to an `ActionMapping` object, which in turns refers to other objects (`org.apache.struts.action.ActionMapping`). Let's write the `ActionMapping` first and then the objects that go with it:

```
<action-mappings
  <action
    path="/LogonSubmit"
    name="logonForm"
    scope="request"
    validate="true"
    input="/pages/Logon.jsp"/>
  <!-- ... -->
</action-mappings>
```

The two related objects are the `logonForm` form bean and the `LogonAction`. We also need to register the `ActionForm` beans in the Struts configuration. The name we use becomes the default attribute name for the object when it is created in the request or session context:

```
<form-beans
  <form-bean
    name="logonForm"
    type="app.LogonForm"/>
  <!-- ... -->
</form-beans>
```

This brings us to adding the two specified Java classes, LogonForm and LogonAction.

### 3.4.11 Constructing the Constants class

While not strictly required, documenting the ActionForward names and other magic tokens is strongly recommended. This is simple to do and can make your codebase much easier to manage over time. When we present code, we usually omit the JavaDoc comments. But in this case we will leave them in. Why? Because the whole point of this class is to *document* the constants. So, in this case, the documentation *is* the code. Listing 3.11 contains the Java source for the Constants class.

**Listing 3.11** Java source for Constants class (`/WEB-INF/src/java/app/Constants.java`)

```
package app;
public final class Constants {

    /**
     * The session scope attribute under which the Username
     * for the currently logged in user is stored.
     */
    public static final String USER_KEY = "user";

    /**
     * The value to indicate debug logging.
     */
    public static final int DEBUG = 1;

    /**
     * The value to indicate normal logging.
     */
    public static final int NORMAL = 0;

    /**
     * The token that represents a nominal outcome
     * in an ActionForward.
     */
    public static final String SUCCESS= "success";

    /**
     * The token that represents the logon activity
     * in an ActionForward.
     */
    public static final String LOGON = "logon";

    /**
     * The token that represents the welcome activity
     * in an ActionForward.
     */
    public static final String WELCOME = "welcome";
}
```

### 3.4.12 Constructing the other classes

We presented the source for the LogonAction and LogonForm classes in sections 3.3.8 and 3.3.9. We also need to include the UserDirectory and UserDirectoryException classes introduced in chapter 1. We can add all of these to our new application unchanged. Our source tree from section 3.4 places them under /WEB-INF/src/java/app/, as shown in figure 3.11.

```
[logon]
|_WEB-INF
|_src
|_java
|_app
|_LogonAction.java
|_LogonForm.java
|_ [...]
```

**Figure 3.11**  
The location of LogonAction,  
LogonForm, and other Java files

The LogonAction also refers to a Constants class. We need to add that before the source will compile.

### 3.4.13 Creating the user directory

In chapter 1, we introduced a simple registration application that stored a user ID and password. These logon accounts are stored in a standard Properties file named user.properties. This can be brought over from that application or re-created under WEB-INF/src/java/resources, as shown in figure 3.12.

```
[logon]
|_WEB-INF
|_src
|_java
|_app
|_resources
|_application.properties
|_user.properties
```

**Figure 3.12**  
The location of user.properties  
and other resource files

Properties files are simple text files. Here's an example that uses the first names of this book's authors as the user ID and their last names as a password:

```
TED=Husted
CEDRIC=Dumoulin
```

```
GEORGE=Franciscus
DAVID=Winterfeldt
CRAIG=McClanahan
```

If you like, you can just type these, or the logins of your choice, into a text file and save it under `/WEB-INF/src/java/resources/user.properties`. Just be sure to enter the user IDs in all uppercase letters, since this is required by the business logic.

Of course, your application can just as easily validate logins against a JNDI service or a database, or use the container's security realm. We cover using data services with Struts in chapter 14.

### 3.4.14 Configuring the ActionErrors

As you will remember, both the `LogonForm` and `LogonAction` may generate error messages. The `ActionError` system is integrated with the application messages. Before putting `LogonForm` and `LogonAction` to the test, we need to add these messages to the `application.properties` document:

```
errors.header=<H3><font color="red">Validation Error</font></H3>You must
  correct the following error(s) before proceeding:<UL>
errors.footer=</UL><HR>
error.username.required=<LI>Username is required</LI>
error.password.required=<LI>Password is required</LI>
error.logon.invalid=<LI>Username and password provided not found in user
  directory. Password must match exactly, including any lower or upper case
  characters.</LI>
```

---

**1.0 vs 1.1** New tags in Struts 1.1 allow you to omit markup from the message. See chapter 10 for more about the Struts JSP tags.

---

As part of the build process, we copy the application resource documents from `/WEB-INF/src/java/resource` to a `resources` package under the `classes` folder where the `ActionServlet` can find them. Be sure to edit that copy and to launch the build process before running any tests.

### 3.4.15 Compiling and testing the logon page

In section 3.4.8, we created the JSP page for our logon form. But to make this work, we need to add the related configuration elements and Java classes, as shown in table 3.10.

**Table 3.10** Logon page configuration elements

Configuration elements	Java classes
<i>LogonSubmit</i> action-mapping <i>logonForm</i> form-bean	<i>LogonAction</i> , subclass of <i>Action</i> <i>LogonForm</i> , subclass of <i>ActionForm</i>

Since these are now in place, we can compile the application and test the logon page. There is a stock `build.xml` in the `WEB-INF` directory that you can use with Ant. The default build target, `compile`, will build the Java classes from the Java source files and copy the application resources message file into the classes directory.

When the build is successful, you can enter the application and follow the link to the logon page. (Depending on how well your container reloads Java classes, you may need to restart the container after a build. When in doubt, restart.)

The logon application should now behave much as it did during our original tour (see sections 3.3.3 through 3.3.6). The difference is that the welcome page does not change after we have logged in, nor does it offer the opportunity to log out. We can fix that in the next section, and our initial application will be complete.

### 3.4.16 Amending the welcome page

Our original draft of the welcome page omitted the conditional logic regarding whether the user was logged in. Now that people can log in, we can add that back so it matches the version from section 3.3.2 (see listing 3.12). The lines we are adding appear in bold.

**Listing 3.12** The revised source for the welcome page (`/pages/Welcome.jsp`)

```
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/struts-logic" prefix="logic" %>
<html>
<head>
<title>Welcome!</title>
<html:base/>
</head>
<body>
<logic:present name="user">
```

```

<h3>Welcome <bean:write name="user" property="username"/>!/</h3>
</logic:present>
<logic:notPresent scope="session" name="user"/>
<h3>Welcome World!</h3>
</logic:notPresent>
<html:errors/>
<ul>
<li><html:link forward="logon">Sign in</html:link></li>
<logic:present name="user">
<li><html:link forward="logoff">Sign out</html:link></li>
</logic:present>
</ul>
<img src='struts-power.gif' alt='Powered by Struts'>
</body>
</html>

```

As shown in figure 3.13, this puts us back where we started. When guests arrive, they are invited to sign in. Once they log on, they are greeted by name and can then sign out. That's progress for you!



**Figure 3.13** The welcome screen before and after logon

### 3.4.17 The Struts ActionForward Action

If you've kept an eye on your browser's location bar, you may have noticed that we never reveal the location of our JSP pages. Many applications don't bother with this nicety, but if you would like to use a strict Model-View-Controller architecture (as described in chapter 2), you may not want to expose any implementation details regarding your View, including whether you are using JSP pages or where you happen to store them. Ideally, all navigation should pass through `.do` Actions that are managed by the controller.

Of course, many times there really isn't anything for the controller to, well, do. This was the case for our logon and welcome pages. They don't require any information from the model and can be displayed by linking directly to the JSP page. But this allows people to bookmark the location of the page. Later, you may need to perform some background action before displaying the logon page, or you may want to move or rename the JSP pages. If people have bookmarked the JSP page, they will try to go back to the old location and either bypass your logic or generate a file not found error. In practice, this usually leads to putting legacy checks into the server page and redirects into the web server—more ways for things to go wrong and more code to maintain.

The moral? We must “virtualize” as many navigation details as possible; otherwise, we will be forced to continually compensate for what the browser may (or may not) cache or store. Instead of linking directly to a “physical” JSP, we should always link to a “virtual” Struts Action, which can then provide the appropriate page.

Of course, writing a custom Action for every page, whether or not it needed one, would be a lot of busy work. A more efficient solution is to deploy a single utility Action that can be customized in the Struts configuration and reused whenever it is needed. Since Struts creates one multithreaded instance of each Action class, this is a very efficient way to ensure that control stays with the controller. All we need to do is pass the Action and the path to the page.

Happily, you can do this using the standard `ForwardAction` bundled in the `struts.jar`. You simply pass the target path as the `parameter` property of the `ActionMapping`:

```
<action
  path="/Welcome"
  type="org.apache.struts.action.ForwardAction"
  parameter="/pages/Welcome.jsp"/>
```

The `ForwardAction` will then look up the target path from the mapping and use it to return an `ActionForward` to the servlet.

The practical upshot is that instead of

```
http://localhost:8080/logon/pages/Welcome.jsp
```

appearing on the browser's address bar, where it could be bookmarked for direct access, the Action URI appears instead:

```
http://localhost:8080/logon/Welcome.do
```

Users can still bookmark this address, but you have much more control now and can change the implementation of the logon activity without worrying about what

some browser has bookmarked. Before, you would have had to consider what should happen if they try to go directly to the old server page.

In a Model-View-Controller architecture, the actions are your API. The server pages are an implementation detail. If the JSPs are exposed to the browser as part of the navigation system, then the Controller and View layers become mixed, and the benefits of MVC are diluted.

We can add other instances of the `ForwardAction` whenever we want to go directly to a page. Since only one `ForwardAction` will be instantiated for the application, all we are really adding is an `ActionMapping` object. If the usual MVC reasons for using `ForwardAction` weren't enough, the modular application feature introduced in Struts 1.1 *requires* that all JSP requests go through an Action. This is because each module has its own configuration context and control has to pass through the `ActionServlet` controller in order to select the configuration for the JSP page. This is not a requirement if you are using a single, default application. But if you follow this practice from the beginning, you can make your application a module without making any changes.

### 3.5 Summary

---

Regardless of what role you play on a development team—engineer, designer, architect, QA—it's helpful to have the big picture of how the application works as a whole. In this chapter, we took a comprehensive look at a small but useful application. By touring, dissecting, and then constructing a logon application, we were able to show you how the Struts framework actually works. As part of the construction phase, we created a design document to outline our goals, client requirements, and program specifications. Given a design to work from, we configured the application's `web.xml`, our Ant `build.xml` script, and the Struts config file. With the right infrastructure in place, we built each component in the order they were needed. Along the way, we pointed out some best practices and emphasized the importance of separating the model, view, and controller.

In chapter 4, we take a closer look at the Struts configuration file. As we have seen here, the configuration plays a strong role in Struts and makes applications easier to design and maintain.