

Strategies for building solid business applications

*SAMPLE CHAPTER*



# STRUTS RECIPES

George Franciscus

Danilo Gurovich

 MANNING




***Struts Recipes***  
by George Franciscus  
and Danilo Gurovich  
**Sample Chapter 7**

Copyright 2004 Manning Publications

# *brief contents*

---

- 1 ■ Basic recipes 1
- 2 ■ Forms and form elements 42
- 3 ■ Struts tag libraries 96
- 4 ■ The Struts-Layout tag library 170
- 5 ■ Validation within the Struts framework 242
- 6 ■ Internationalization 294
- 7 ■ Logging in, security, and guarding 317
- 8 ■ Advanced recipes 348
- 9 ■ Testing 424



*Logging in, security,  
and guarding*

---

*Security is mostly a superstition. It does not exist in nature, nor do the children of men as a whole experience it. Avoiding danger is no safer in the long run than outright exposure. Life is either a daring adventure, or nothing.*

—Helen Keller

Are you who you are? Since you *are* who you are, who are you in relation to my application?

If you understand the above line completely, then you know the difference between authentication and authorization. It's really not that simple when you delve deeply into the workings of an application. There are many questions that need to be answered, both at the level of the intent and nature of the application and at the architectural underpinnings you wish to employ.

Let's look at authentication first. We'll take the example out of the electronic world of 1s and 0s and use a human interaction example. Let's say you're sitting in a crowded coffee house at a table with two chairs. You're reading the paper and not interacting with anyone, and one of the few seats left in the entire establishment is at your table. Somebody comes up with a cup of coffee and asks if they can sit at your table since you have an empty seat. You say, "Sure, go ahead. If you'd like to read some of my paper, I'm through with the Sports and Entertainment sections." You are now sharing information with an anonymous user.

Time marches on, and you're sipping your coffee, and you look up from your paper about the same time as your tablemate. After an awkward second, the stranger says "Thanks so much for the spot at your table, it was very crowded here today and I needed to sit for a minute. My name is Dean." Dean is no longer anonymous, and you now know something about him. At this point you need to decide whether to trust him or not, and for the purposes of sharing a table and a paper, Dean's name is about all you need to know. You have a pleasant conversation.

A few days later you go to the coffee shop, order your usual latté and sit down with your paper. Dean comes in a few minutes later, and although the place is not very busy, he comes over to your table, and brings a different paper and offers to share. As you two sit quietly and read your papers, you trade a few comments and get to know each other a little better, and it becomes routine to sit together over coffee and conversation whenever you both are at the coffee shop at the same time. You trade personal information and details over a period of weeks, and you now know quite a bit about Dean, so when you see him, the interaction between you is more personalized, and there are fewer "barriers" than if Dean were a complete stranger. You might buy the coffee, or Dean might pick up an extra biscotti. You now know Dean from other strangers, and he is welcome at your table at any time. He is "authenticated" as Dean, and "authorized" as someone with whom you share a table, paper, and conversation.

You've made quite a few decisions in the above scenario: You've decided to let someone share your space and information; further, you've decided to expand

this decision into a specific person and interact with him based upon who he is and what he does. Let's now draw some parallels with a web application.

In our “coffee shop” web application, we have a web site with information available to all users (strangers that ask to sit at the table and share the newspaper). As the person gives a few details and requests, we offer more information. At this point we've decided the level of “security” that we need to have is a name, as the person is not asking to borrow a thousand dollars and we're not offering it anyway. As we decide to share more information, we know who “Dean” is, and because we know more about him than just his name, we have also “authorized” him to share in more information based upon “who” he is *specifically*. If another person asked to sit at your table, he or she might have to go through the same procedure as Dean.

In this chapter we're going to reveal different ways to implement security strategies for general and specific areas of an application. We'll involve ourselves in the following four areas:

- A simple log-in/log out authentication strategy—This is a lightweight security model, as it has low security and is very generic in nature. This is a very basic model that is not container based, and is expandable and scalable.
- Integrating domain authentication with Struts—Becoming more common and a little more “industrial strength,” container-based authentication would compare to our coffee shop being a members-only establishment: anyone getting in has to be authenticated following an understandable and agreed-upon policy, and is then given a name badge by the management. When Dean asks to sit at your table, you already “know” his name and that he's an “authentic” member of the coffee shop.
- Protecting individual fields from a group of users—This is similar to meeting someone for the first time. As you get to know them more, they have “access” to more information. Even if you have many people at your table, they *might* have access to different information with respect to a conversation's context.
- Protecting areas on a page from certain groups of users—A similar recipe to the above field-level approach, but at the page level.
- Using Struts to protect your `Actions` from unauthorized access. You don't want a stranger at the coffee shop to drink from your mug or take a bite out of your cookie! We've devised a couple of different recipes to address specific methods for guarding `Actions` with various mechanisms to do this.

The recipes in this chapter give the user a choice—from very simple to quite complex—of methods to control access, protect information, and guide the user. While authentication and authorization is usually implemented to keep out unwanted or malicious users, it's also well advised to think of these strategies as a way to “guide” specific users to information that is more usable and pertinent to their needs. More often than not a user isn't malicious, they are just looking for the most germane information, and authorization and authentication is an accepted standard to make this happen.

## 7.1 Tomcat domain authentication and Struts

---

### ◆ **Problem**

You want to integrate domain authentication with Struts.

### ◆ **Background**

Security is a hot topic with respect to web applications. It is not uncommon for many enterprise-level projects to expose sensitive business logic and processes to a browser. It is important that this type of information be kept from malicious or “wandering” users.

Tomcat and other servlet containers have an excellent domain-level authentication system that can be integrated with databases or a flat file to authenticate the user and establish the level of access to different areas through the “role” assigned to the user. This system is part of the Servlet 2.2 specification:

There are currently four separate mechanisms for user authentication in a servlet container.

- *HTTP Basic*—This is by far the simplest type of authentication to implement. As a browser makes a request to an area protected by the container, a system-level authentication box appears, requesting a user name and password (users of many web-based mail applications will recognize this dialog box immediately). This box is triggered through a response from the container in the form of a “401 unauthorized” message, which triggers the dialog box from any HTTP-compliant browser. The log-in and password information is sent back to the server via Base64 authentication. Once this information is processed, access is granted to the page and all pages are protected by the container and the level of access set up in the container's configuration. HTTP Basic is extremely easy to set up and its support is

universal, but its look and feel is fixed and although it is Base64 encoded, this is not a secure encryption method.

- *HTTP Form*—Consider this type of authentication as HTTP Basic authentication with a “friendly face.” Instead of the system-level box, the user is presented with an HTML page with a log-in and password text box containing an HTML form (this is outside of the Struts application) with specific form and field names for posting to the server. If authentication fails, a specified error page is displayed. It is important to note that this method should only be implemented when a session is maintained using cookies or HTTPS.
- *HTTPS Client*—This is HTTP authentication through a Secure Sockets Layer. All data is transmitted using public-key cryptography and developers don’t have to know about the encryption to implement the system. It requires a certificate from an authority such as VeriSign. More information is available on this type of authentication in the last recipe of this chapter.
- *HTTP Digest*—This is the same as HTTP Basic except that the password is encrypted as well as encoded. This provides the advantages of Basic authentication with more security, but is only supported on Internet Explorer 5 and newer versions. Not all servlet containers implement this, as it is not mandated by the Servlet specifications.

This recipe will discuss Basic and Form-based authentication and their integration into the Struts framework. Other recipes in this chapter will expand on this information.

A servlet container can secure an entire application or part of it from users through the use of user names and the roles assigned to them. This definition is created in the web application *and* in the servlet container’s configuration files. The servlet container can be configured to use any type of persistence layer available, from a simple XML file to an open source database or fully functional LDAP that conforms to the Servlet security model. The directories and pages secured by the application are defined in the web.xml file at the container or application level, whichever is applicable. If you are securing one application or many, you have a consistent and well-defined method in which to complete your task.

With respect to Struts, security using this method is accessed from outside of the framework. With Basic Authentication you can secure your entire application. When using a form-based mechanism it is important to create directories that are not secured by the container’s configuration, as any servlets, images, CSS files, or HTML pages will not render if they are “behind” the secured areas of your application, and, in most cases, you wouldn’t want them to be.

**◆ Recipe**

Two methods are discussed here. First is the HTTP Basic authentication method, which opens an Alert dialog box with a log-in and password text line, similar to the box in figure 7.1.

This method eschews any HTML code and immediately loads the index or designated welcome file as soon as Tomcat decides who you are and what role you have. For our second method (HTTP Form-based authentication), you need to create a log-in and log-in error page using standard HTML. When any protected page is requested by an unauthorized user, the log-in page is requested and the designated welcome file is displayed. If the user fails to authenticate, the error page you've created is sent to the browser.

We must first decide what we wish to protect. If you wish to protect an entire application with absolutely no access until the user is known, you must use the Basic method, as it uses an Alert box and needs nothing in the way of HTML pages, images, or links to do its job. If we wish to use the Form method, then two simple HTML or JSP pages must be created, and any files (such as images, includes, and so on) that we want to use on these pages must also be made available in an unprotected area.



Figure 7.1 Dialog box used with Basic authentication.

**NOTE** This recipe shows the use of a flat-file authentication. More robust applications are simple to adapt but are out of the scope of a Struts recipe. Many sources have information as to how to configure your specific data source to the container's system.

To begin, open the `web.xml` file in the `WEB-INF` directory of your application. Find the end of your file at the `</web-app>` tag and insert the following information (tailored to your directory needs) in front of it. Both Basic and Form types are shown below in listing 7.1:

**Listing 7.1 Basic authentication in web.xml file**

```
</taglib>
<!--Security for entire application-->
<security-constraint>
  <web-resource-collection>
    <web-resource-name>My web application</web-resource-name>
    <url-pattern>*/</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
    <http-method>PUT</http-method>
    <http-method>DELETE</http-method>
    <http-method>HEAD</http-method>
    <http-method>OPTIONS</http-method>
    <http-method>TRACE</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>applicationuser</role-name>
    <!--more roles can be added here -->
  </auth-constraint>
</security-constraint>
<login-config>
  <auth-method>BASIC</auth-method>
</login-config>
</web-app>
```

The above method protects your entire directory, but there is no custom log-in or error screen provided. To provide your log-in and error screen, it is crucial to specify the directories and files that are protected, as shown below in listing 7.2:

**Listing 7.2 Form authentication listing in web.xml file**

```
</taglib>
<!--Security for application with custom log-in and error pages -->
<security-constraint>
  <web-resource-collection>
```

```

<web-resource-name>My Custom Page Application</web-resource-name>
<url-pattern>/index.jsp</url-pattern>
<url-pattern>/pages/*</url-pattern>
<http-method>POST</http-method>
<http-method>PUT</http-method>
<http-method>DELETE</http-method>
<http-method>HEAD</http-method>
<http-method>OPTIONS</http-method>
<http-method>TRACE</http-method>
</web-resource-collection>
<auth-constraint>
<!--more roles can be added here -->
<role-name>customuser</role-name>
</auth-constraint>
</security-constraint>
<login-config>
<auth-method>FORM</auth-method>
<form-login-config>
<!--These are the two pages that must be created, if you attach
images, css, or include files it is important that they be in a
non-protected area or your results will not be what you intend -->
<form-login-page>/login.html</form-login-page>
<form-error-page>/error.html</form-error-page>
</form-login-config>
</login-config>
</web-app>

```

Next, let's take a look at the `server.xml` file in the `{Tomcat-directory}/conf/` folder. It is heavily commented, but we are looking for the section after the `<engine>` and `<logger>` tags in the default file. For flat-file realm authentication, uncomment the line:

```
<Realm className="org.apache.catalina.realm.MemoryRealm"/>
```

Make sure that any other realm tags are commented out. Note that this uncommenting procedure is unnecessary when using Tomcat 4.1+, since the `UserDatabaseRealm` is enabled by default. You are now ready to name your users in the `tomcat-users.xml` file, also located in the `{Tomcat-directory}/conf/` folder. Listing 7.3 demonstrates a sample `tomcat-users.xml` file that creates a couple of users for the Form-based login described in the above `web.xml` file listed in 7.2:

#### Listing 7.3 tomcat-users.xml file employing form-based authentication

```

<?xml version="1.0" encoding="utf-8"?>
<tomcat-users>
  <role rolename="customuser"/>
  <user username="aguy" password="thisPassword" roles="customuser"/>

```

```
<!-- you may comma delimit more roles in the "roles" attribute-->
<user username="anotherguy" password="thisGuysPwd" roles="customuser"/>
</tomcat-users>
```

Finally, let's look at the JSP log-in and error pages in listings 7.3 and 7.4. The bold variable names in the HTML form used here are recognized by your container and **must** be used.

#### Listing 7.4 Login.html for form-based authentication

```
<html>
<head>
<title>My Log In Page</title>
</head>
<body>
<form action="j_security_check">
<table style="width: 50%">
  <tr>
    <td>Username:</td>
    <td><input type="text" name="j_username" /></td>
  </tr>
  <tr>
    <td>Password:</td>
    <td><input type="password" name="j_password" /></td>
  </tr>
  <td>&nbsp;</td>
  <td><input type="submit" value="Login" /></td>
</tr>
</table>
</form>
</body>
</html>
```

#### Listing 7.5 error.html page for handling any login errors for above login page

```
<html>
<head>
<title>Log in error</title>
</head>
<body>
<p style="font-size: 125%; color: red">Log-In Error</p>
<p>The user name and/or password are incorrect.</p>
<p><a href="login.html">try again</a></p>
</body>
</html>
```

**◆ Discussion**

There might be a situation where you just wish to protect your `Actions` instead of—or in addition to—individual pages. This is simply done by adding

```
<url-pattern>*.do</url-pattern>
```

within the `<web-resource-collection/>` xml tags. This implementation protects your methods, but a security hole exists without additional pattern-mappings, as a user could type a URL in directly—`http://www.mysite.com/pages/myFooPage.jsp` and possibly access it.

This is a great place to discuss the options for solving that particular problem:

- 1 Place all JSPs under `WEB-INF`.
- 2 Secure `*.jsp` with a constraint that *no* user has permission to use, thereby preventing direct access.

Although the limitations of the flat file are apparent when compared to what can be achieved with a database, it really depends upon the size and scope of the application you're creating. As more roles and realms are needed, it is possible to protect areas in your site from certain roles as they are created. This is explained and extended in the recipes related to this one.

Finally, you might want to secure your entire site by listing some or all of your JSP files inside the `WEB-INF` directory. This type of security allows you to use the `*.do` security constraint with ease, as only those files that you wish to protect can be accessed through the Struts `Action`. Please see “Best Practice: Use the container to protect resources” for details.

**◆ Related**

- 7.2—Handling log out
- 7.4—Secure an action mapping using the container
- 7.5—Customized action mapping security
- 7.6—Protect areas on a page
- 7.7—Protect fields

## 7.2 Handling log out

---

**◆ Problem**

You want to implement log out in your application.

### ◆ **Background**

We've discussed logging into an application, but once you're there, you're eventually going to need to make a clean break with the page upon leaving. It's important to judge the level of security you want for your web application; not just to prevent hacking, but to control the user's `Actions` within it. Just sticking a logout button on a page and round-tripping it to the welcome screen might be good for a test application, but let's look at the handling of some real-world problems.

First, let's decide in advance just how far out you need to go. A great deal of this decision depends upon the security your application needs. If you are interested only in capturing a user name, authenticating and recording entry and exit, and controlling "how" a user can come into your application or bookmark it, then that's a lighter-weight solution than one that regulates what a user sees, does, or accesses.

But if you're implementing container-managed security through your application server or a servlet container such as Tomcat, then you need to make a more robust and secure solution that not only ensures that entry into your site is controlled, but it also makes sure that the user can't just "hit the back button" and waddle back into the application.

### ◆ **Recipe**

Either way, there's not a lot of code to logging out, it just depends on what you want to do and what you get in return for your trouble. First, you might just want to control a small application by using `<logic:present/>` or `<logic:if>` tags to evaluate whether a user is logged in or not. This is often done by detecting the presence of a user variable, or by detecting a flag with a role, group, or Boolean variable. With such a simple log-in or security method as this is, it is only a matter of using an `Action` class with the following code in the `execute()` method to set the user's name and any user-specific session information to null.

```
// we already have access to HttpServletRequest
HttpSession session = request.getSession();
session.setAttribute(Constants.YOUR_LOGIN_FLAG_OR_USER_KEY, null);

//you may want to set other session and/or request variables to null here
//forward to a log-out page or back to welcome or whatever...

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

If you are using a J2EE-compliant application server or servlet container and wish to use container-managed security, you need a more "industrial-strength" solution. Interestingly, such a solution involves writing less code:

```
HttpSession session = request.getSession();
session.invalidate();

//forward to a log-out page or back to welcome or whatever...

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

The above code completely invalidates the session and forwards to a logout page. When the method requesting a page is detected, the user cannot enter back into any part of the defined realm without having to completely log back in.

#### ◆ **Discussion**

The first method prevents the user from backing into an application, or walking into a side door. It also requires the developer to add some security-based code into every page he or she creates, or extend the `Action` servlet and create a custom security model. While this can be handled quite easily through the use of `Tiles`, `includes`, and other similar methods, it still requires some conscious effort. This effort is rewarded by very tight control and the ability to create small-to-large views based upon developer-defined schemes. By using this method, you “own” your security model and may configure it to your application in the manner you see fit—creating views, controllers and applications that discriminate between roles, users, and other custom groupings.

The J2EE container-managed security is much more global, as you can control access to one or many applications in the servlet container, and is much more common with web applications because of its ease of implementation and the fact that you don’t have to implement any page-level security measures. Everything is handled by the server as it tracks your session. Of course, you give up tight control on the view and navigation without resorting to additional measures that might have some overlap and redundancy, but are often needed and required. These types of controls are explained further in the remaining recipes of this chapter.

It just depends upon what you need. Sometimes a little security is just fine, sometimes your requirements are global, and then again you might need something very specific. The remaining recipes in this chapter discuss securing applications in simple and complex manners. There are many more ways of securing an application than there are for logging out, and the above recipe takes care of most of a developer’s needs as long as the strategies implemented are not very esoteric.

◆ **Related**

- 1.3—What is “jsessionid” and why do I need it?
- 7.1—Tomcat domain authentication and Struts

## 7.3 Switch to SSL and back again

---

◆ **Problem**

You want to use SSL in your application.

◆ **Background**

Most organizations consider information their most valuable asset. Guarding information is more than a full-time job for developers and administrators alike. The protection of information must be approached from many different angles. The database, user interface, and business layer are just a few of the critical areas that need your attention. Even the door to the server room must be kept locked. More importantly, once the data has left the building via the Internet, it can be compromised by some very talented people electronically poking and prodding data on the wires.

In this recipe we demonstrate how to enable Secure Socket Layer (SSL) in your container. In addition, we introduce you to the Struts SSL extension for HTTP/HTTPS switching (`ssl-ext`) library—a package allowing you to declare which `Actions` are SSL enabled. This very easy-to-use package allows you to convert between SSL-enabled `Actions` and disabled ones with ease.

Before diving into the recipe, let’s cover some SSL basics. SSL is a network protocol originally developed by Netscape to provide authentication and transmission security. The SSL protocol resides in a layer above TCP/IP, but below application protocols such as HTTP, LDAP, and IMAP. Prior to transmission, the client verifies the identity of the server. Similarly, the server might optionally verify the identity of the client. To thwart the efforts of digital eavesdroppers, the two hosts agree upon a cryptographic algorithm, commonly called a cipher, to encrypt and decrypt transmissions. In addition to providing confidentiality, SSL audits the transmission to ensure it remains untampered enroute. The entire workings of SSL are beyond the scope of this recipe, but we encourage you to explore SSL using some of the resources mentioned in the resource section.

Most containers, including Tomcat, provide SSL services. Enabling SSL on your container ensures your communication is secure and not tampered with. The container is able to recognize SSL communication by inspecting the URL scheme. The URL scheme is the portion of the URL preceding the colon. An HTTP unsecured protocol is identified with “http” before the colon. For example, `http://127.0.0.1` indicates an unsecured HTTP protocol. An “s” is placed after http to indicate the request wishes to communicate using SSL. For example, `https://127.0.0.1` is the appropriate URL for an SSL HTTP request.

Many applications have both SSL and non-SSL links. The challenge is to format your links for SSL when you want a secure communication. By default, the Struts link tag uses the same URI scheme as the current page you are viewing. So, how can you build a link from a secured page to an unsecured one, and vice versa? One option is to build your own link tag to draw SSL links, but this can quickly lead to a maintenance burden whenever you need to convert from one to the other. Now, if you could do this declaratively in the `struts-config.xml` file, then you would be a happy camper. You could use tags to paint your link, but you could change them declaratively in the `struts-config.xml` file as things change. Fortunately, that’s exactly what the Struts SSL extension for HTTP/HTTPS switching (`ssl-ext`) library does for you. In this recipe we show you how to set up SSL with Tomcat. We then show you how to set up and use `ssl-ext` in your application to encrypt your data over the wire. A complete explanation of SSL authentication is beyond the scope of this book; instead, we focus on the Struts development aspect of SSL.

### ◆ **Recipe**

Implementing this recipe is divided into three sections. In the first section we show you how to enable SSL on Tomcat. Next, we describe exactly what needs to be done to install `ssl-ext` in your application. Lastly, we apply `ssl-ext` into your code. Let’s get started!

#### **Step 1: enable SSL in Tomcat**

- 1 Download and install JSSE 1.0.2 (or later) from <http://java.sun.com/products/jsse/>. This web site tells you everything you need to know to install JSSE. Note that the Java 2 SDK Standard Edition v 1.4 has prebundled JSSE for you. If you are using JDK v 1.4, then you can skip this step.

- 2 Create a certificate keystore. Enter one of the following commands:

For Windows: %JAVA\_HOME%\bin\keytool -genkey -alias tomcat -keyalg RSA

For UNIX: \$JAVA\_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA

You will be asked a number of questions. For the purposes of this exercise, simply enter whatever makes sense to you. See your system administrator when the time comes to implement into a controlled environment.

- 3 Uncomment the SSL HTTP/1.1 Connector from conf/server.xml. The uncommented connector should look like this (take special note of the port number, you are going to need to know it later):

```
<Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
  port="8443"
  minProcessors="5"
  maxProcessors="75"
  enableLookups="true"
  acceptCount="10"
  debug="0"
  scheme="https"
  secure="true"
  useURIVValidationHack="false">
<Factory
  className="org.apache.coyote.tomcat4.CoyoteServerSocketFactory"
  clientAuth="false"
  protocol="TLS" />
</Connector>
```

- 4 Restart Tomcat. Point your browser at <https://127.0.0.1:8443> to make sure everything went as planned. If Tomcat renders the page, then you are successful.

Congratulations, you have enabled SSL on Tomcat.

## Step 2: set up Struts SSL extension for HTTP/HTTPS switching (ssl-ext) for your application

Because ssl-ext is a Struts extension library, it's not surprising that its installation is similar to Struts. If you have installed Struts, the following instructions will seem familiar.

- 1 Download the ssl-ext binaries from <http://sslext.sourceforge.net>. The download contains a working sample application. Unzip the download to the Tomcat webapps directory. In the following steps you will copy over many of the files from the sample application to create your own ssl-ext application.

- 2 Copy over `sslext.jar` from the sample app to your `WEB-INF/lib` directory.
- 3 Copy over `sslext.tld` from the sample app to your `WEB-INF/lib` directory.
- 4 Place the following snippet with the other taglibs in the `web.xml` file:

```
<taglib>
  <taglib-uri>/WEB-INF/sslext.tld</taglib-uri>
  <taglib-location>/WEB-INF/sslext.tld</taglib-location>
</taglib>
```

- 5 Add the plug-in tag to the `struts-config.xml` file. Take special note of the `httpsPort` property. It must be the same one used to configure Tomcat (step 1, instruction 3). As you might expect, the `httpPort` property should match the Tomcat configuration found in the `conf/server.xml`. The two values defined below are the Tomcat default values.

```
<plug-in className="org.apache.struts.action.SecurePlugIn">
  <set-property property="httpPort" value="8080"/>
  <set-property property="httpsPort" value="8443"/>
  <set-property property="enable" value="true"/>
</plug-in>
```

That's it! You are ready to start using `ssl-ext`.

### Step 3: build an application using `ssl-ext`

Let's build an application to demonstrate `ssl-ext` in action. In this step, we'll show how `ssl-ext` is used to format the URL for links targeting SSL-secured pages. Conversely, we'll demonstrate URL formatting for links targeting ordinary non-SSL pages. We'll see that securing the page is done by making a small change to the `struts-config.xml` file. Listing 7.6 shows the `struts-config.xml` action mapping for a sample application.

**Listing 7.6** `Struts-config.xml`

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

<!DOCTYPE struts-config PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
  "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>

  <global-forwards type="org.apache.struts.action.ActionForward">
    <forward name="unsecured" path="/unsecured.do"/>
    <forward name="secured" path="/secured.do"/>
    <forward name="menu" path="/menu.do"/>
  </global-forwards>

  <form-beans>
```

```
<form-bean name="dummyForm" type="com.strutsrecipes.ssl.forms.Dummy" />
</form-beans>

<action-mappings type="org.apache.struts.config.SecureActionConfig"> ❶
  <action path="/menu"
    type="org.apache.struts.actions.ForwardAction"
    parameter="/WEB-INF/pages/menu.jsp">
    <set-property property="secure" value="false"/>
  </action>

  <action path="/unsecured"
    type="org.apache.struts.actions.ForwardAction"
    parameter="/WEB-INF/pages/unsecured.jsp">
    <set-property property="secure" value="false"/> ❷
  </action>

  <action path="/secured"
    type="org.apache.struts.actions.ForwardAction"
    parameter="/WEB-INF/pages/secured.jsp">
    <set-property property="secure" value="true"/> ❸
  </action>

  <action path="/seuresubmit"
    type="org.apache.struts.actions.ForwardAction"
    name="dummyForm"
    parameter="/WEB-INF/pages/seuresubmit.jsp">
    <set-property property="secure" value="true"/> ❹
  </action>

  <action path="/unseuresubmit"
    type="org.apache.struts.actions.ForwardAction"
    name="dummyForm"
    parameter="/WEB-INF/pages/unseuresubmit.jsp">
    <set-property property="secure" value="false"/> ❺
  </action>
</action-mappings>

<plug-in className="org.apache.struts.action.SecurePlugIn">
  <set-property property="httpPort" value="8080"/>
  <set-property property="httpsPort" value="8443"/>
  <set-property property="enable" value="true"/>
</plug-in>
</struts-config>
```

To secure an Action with SSL, you nest a `<set-property>` tag inside the Action tag. The value of the property attribute is always `secure`. A `true` value of the value attribute indicates we want URLs to issue an SSL request by setting the URI scheme to “https”. Similarly, a `false` value indicates the page is an ordinary, unsecured page; and the URI scheme should be “http”. A value of any defaults to the current page’s scheme. Although you can achieve the same effect using an

ordinary Struts link tag, the any value let you change it to true or false by making a change to the struts-config file.

Because the `secure` property is not supported by the default action mapping, we need to override it at ❶. Listing 7.7 demonstrates how `ssl-ext` link tags are used to generate a protocol formatted URL.

**Listing 7.7 menu.jsp: JSP using `ssl-ext` link tags**

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

<html:html>
<h1>SSL: Menu</h1>

<h2>Links</h2>
<br><sslext:link forward="unsecured">unsecured</sslext:link> ❷
<br><sslext:link forward="secured">secured</sslext:link> ❸

<h2>submit to secured</h2>
<sslext:form action="/securesubmit" > ❹
<br><html:text property="name" value="" />
<html:submit />
</sslext:form>

<h2>submit to unsecured</h2>
<sslext:form action="/unsecuresubmit" > ❺
<br><html:text property="name" value="" />
<html:submit />
</sslext:form>

</html:html>
```

The links to secured pages are prefixed with `https`, and the links to unsecured are formatted with the usual `http`. For example, the link at ❷ is rendered “unsecured” as `http://127.0.0.1:8080/ssl/unsecured.do`, whereas the link at ❸ is rendered “secured” as `https://127.0.0.1:8443/ssl/secured.do`.

The most striking observation in listing 7.7 is that we never specify whether or not the page is secured with SSL. That information is defined in the `struts-config.xml` file. The link tag at ❷ in listing 7.7 maps to ❷ in listing 7.6. Analogously, ❸ maps to ❸, ❹ maps to ❹, and ❺ maps to ❺. The `ssl:ext` link tag consults the `struts-config.xml` file when painting the URL protocol segment. To use the `ssl-ext` taglib you need to declare it as we have done at ❸.

Submits work in much the same way, except you must use the `form` tag from the `ssl-ext` namespace ❹❺.

There you have it. Applying `ssl-ext` is completely unraveled in the preceding steps.

#### ◆ **Discussion**

Because Step 1 and Step 2 are self-explanatory we dispense with reviewing those sections. Step 3 is far more interesting and worthy of closer inspection. The JSP work is straightforward, and despite the `sslext` namespace, it looks pretty much the same as any Struts JSP. Instead of painting the links with the Struts link tag, we use the same tag from the `sslext` namespace. Behind the scenes, the tag is using the action mapping to set the URI scheme to either `http` or `https`. To paint the URL with `https`, you set the `secure` property on the action mapping to `true`. To paint the URL as “`http`”, you set it to `false`. The submit works in the same way, except you use the form tag from the `sslext` namespace instead of the Struts `html` namespace.

There is one more tag we haven’t discussed. The `pageScheme` tag uses the `secure` attribute to force a redirect to `http` or `https`. For example, an `https` request to a page with `<sslext:pageScheme secure="false">` redirects the request to `http`, despite the fact the request was made to `https`.

The job of ensuring the URL is prefixed with the right scheme would certainly be more difficult if it were not for `ssl-ext`. Instead of tackling this problem in the JSP by tailoring each link for SSL, we can manage it from the `struts-conf.xml` file. The maintenance payback is substantial. For example, if we had 20 links to the same Struts `Action`, then 20 JSP changes would be required to change a link to use SSL. The identical change under `ssl-ext` requires just one change to the `struts-config.xml`! Too err is human, to avoid errors is divine.

#### **BEST PRACTICE**

*Use SSL judiciously*—SSL is a proven, reliable, flexible, and popular way to authenticate and secure applications, but you should use it only when you need it. All SSL transmissions encrypt and decrypt data. Depending on the chosen cipher, these operations can impact your performance.

#### ◆ **Reference**

- [SSLD0C] SSL Overview
- [SSLEXT] SSL Extension for Struts HTTP/HTTPS switching
- [SSLSPEC] SSL Specification
- [SSLTOM] SSL Tomcat Configuration
- [URI] W3 URI Specification

## 7.4 Secure an action mapping using the container

---

### ◆ **Problem**

You want to protect an action mapping from unauthorized access.

### ◆ **Background**

Suppose you want to restrict update access to users with a supervisor role, but provide inquiry access to all other users. Removing the user's ability to navigate to the unauthorized page impedes their ability to submit the update using the application, but a clever user can spoof a URL using the web browser's location bar. Another solution is to use the container to define security constraints in the web.xml file. However, as the number of URLs grows, this strategy becomes unwieldy and difficult to maintain. This recipe shows you how to use inherent Struts functionality to protect your action mappings by registering roles against them.

**REQUEST PROCESSOR** Struts supports the Model-View-Controller (MVC) design paradigm. MVC demands that all requests are dispatched to a controller. The controller becomes the single point of entry to the entire application, forcing all requests to be handled in a consistent manner. The Struts `ActionServlet` fulfills the MVC controller responsibilities, but delegates the real work to the `RequestProcessor`. The default `RequestProcessor` is responsible for finding the appropriate `ActionMapping`, populating the `ActionForm` with request parameters, validating the `ActionForm` when required, and invoking the `Action` before forwarding the response to the appropriate `ActionForward`. In addition to this busy work load, the `RequestProcessor` ensures request processing is restricted to authorized users.

### ◆ **Recipe**

First, you need to configure the container to challenge the user for their user name and password when they attempt to access the application. Once authenticated, the container has the means to help you determine a user's role. See recipe 7.1, "Tomcat domain authentication and Struts," to add users and roles to the container's security mechanism.

To restrict access to an action mapping for a list of roles, you simply add a list of comma-delimited roles to the `roles` attribute on the `Action` tag inside the `struts-config.xml` file. The `roles` attribute specifies which roles are authorized to access the action mapping. The omission of the `roles` attribute leaves the action mapping unprotected. It's that simple!

**Listing 7.8 The Action tag roles attribute**

```
<action-mappings>
  <action path="/unprotected" ❶
    type="org.apache.struts.actions.ForwardAction"
    parameter="/WEB-INF/pages/unprotected.jsp"/>
  <action path="/protected" ❷
    roles="supervisor,appladmin"
    type="org.apache.struts.actions.ForwardAction"
    parameter="/WEB-INF/pages/protected.jsp"/>
  <action path="/menu"
    type="org.apache.struts.actions.ForwardAction"
    parameter="/WEB-INF/pages/menu.jsp"/>
</action-mappings>
```

Let's step through listing 7.8.

At ❶ we show an unprotected action mapping. At ❷ we protect the action mapping by adding a `roles` attribute. Users possessing either `supervisor` or `appladmin` are authorized to access this action mapping.

**◆ Discussion**

A good practice to secure your JSPs from direct client access is to place them under the `WEB-INF` directory. Once secured, the `ActionServlet` becomes your vehicle to access your JSPs. Whenever a request is received by the `ActionServlet`, the `RequestProcessor` is called into duty to ensure the user is authorized to invoke the requested `Action`. The `RequestProcessor` consults the container's security mechanism by calling the `isRoles()` on the request object to validate the user's container-defined role against the list of roles specified in the `Action` tag's `roles` attribute. Authorized users are cleared to process the `Action` in the usual way. Those users denied access are presented with the standard "HTTP Status 400 – User is not authorized to access *resource name*."

This recipe secures action mapping from request spoofing. It provides an effective means of securing action mappings using the container's built-in security mechanism with very little effort on your part.

Although this recipe is an effective strategy of securing JSPs and pages, there are a couple of limitations. First, you are tied to the container's security mechanism to define your users and roles. Second, users precipitating security violations are presented with the standard server error screen and left without navigability back to the main application. These limitations are addressed in recipe 7.5, "Customized action mapping security."

This recipe takes a coarse-grained approach by securing the entire page. See recipe 7.6, “Protect areas on a page,” and recipe 7.7, “Protect fields,” for a fine-grained approach to a similar problem.

**BEST PRACTICE** *Use the container to protect resources*—If your container supports Servlet specification 2.3, an excellent practice is to force all client requests through the `ActionServlet`. Simply place all your JSPs under the `WEB-INF` directory. The servlet engine denies client access to all resources stored below the `WEB-INF` directory. With your JSPs protected by the servlet engine, all requests must be marshaled through the `ActionServlet` and processed by the `RequestProcessor`.

◆ **Related**

- 7.1—Tomcat domain authentication and Struts
- 7.5—Customized action mapping security
- 7.6—Protect areas on a page
- 7.7—Protect fields

◆ **Reference**

- [CORE] Core J2EE Patterns, Best Practices, and Design Strategies, Controlling Client Access

## 7.5 Customized action mapping security

---

◆ **Problem**

You want to use Struts to protect your `Actions` from unauthorized access, but you want to use your own security mechanism. Furthermore, you want to send the user to a page of your choice when security violations are detected.

◆ **Background**

In recipe 7.4, “Secure an action mapping using the container,” we explored inherent Struts functionality to detect unauthorized access of `Actions`. Using this Struts feature can be very effective, but there are some limitations. First, you are forced to use the container’s security mechanism. Second, the user was left without navigability back to the application. In this recipe we show you how to address these two limitations by extending the `RequestProcessor` and leveraging Struts’ ability to process exceptions declaratively. See recipe 5.4, “Use declarative exception handling.”

Let's review the controller. The Struts `ActionServlet` receives all requests, but delegates the real work to the `RequestProcessor`. The default `RequestProcessor` is responsible for finding the appropriate `ActionMapping`, populating the `ActionForm` with request parameters, validating the `ActionForm` when required, and invoking the `Action` before forwarding the response to the appropriate `ActionForward`. Moreover, the `RequestProcessor` ensures request processing is restricted to authorized users. For a description of the `RequestProcessor` and its relationship to the `ActionServlet`, we recommend you read recipe 7.4.

The hallmark of a good framework is one that allows you to tailor its behavior by extending and overriding *hot spots*. Hot spots are framework methods which can be overridden to implement customize behavior. Struts 1.1 refactored the `ActionServlet` to introduce the `RequestProcessor` to empower developers with 15 hot spots. In this recipe we override the `RequestProcessor` `processRoles` method to change the behavior of `Action` authorization.

#### ◆ **Recipe**

Let's begin by creating a brand new `RequestProcessor` to utilize a different authorization mechanism than the one provided by the default `RequestProcessor`. Because we want to keep most of the default implementation intact, we extend `RequestProcessor` and override the `processRoles` method, as shown in listing 7.9.

**Listing 7.9** `SecurityRequestProcessor`

```
package com.strutsrecipes.customguardaction.controller;

import java.io.IOException;

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

import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.RequestProcessor;

import com.strutsrecipes.customguardaction.business.SecurityComponent;
import
    com.strutsrecipes.customguardaction.exceptions.SecurityViolationException;
public class SecurityRequestProcessor extends RequestProcessor {

    protected boolean processRoles (
        HttpServletRequest request,
        HttpServletResponse response,
        ActionMapping mapping)
```

```

throws IOException, ServletException {

    // Obtain the list of roles from action config
    String roles [] = mapping.getRoleNames();
    if ((roles == null) || (roles.length < 1)) {
        return (true);
    }

    // verify user possesses role
    for (int i = 0; i < roles.length; i++) {
        if (isUserInRole (request.getRemoteUser(), roles [i])) {
            return (true);
        }
    }

    // invoke declarative exception handling
    ActionForward forward = processException(request,response,
        new SecurityViolationException(),null,mapping);
    // forward user
    if (forward != null) {
        process ForwardConfig(request, response, forward);
    }
    return (false);
}

protected boolean isUserInRole (String user, String role) {
    // invoke proprietary security mechanism
    return new SecurityComponent().isAuthorized(user, role);
};
}

```

Now, let's register `SecurityRequestProcessor` with the `ActionServlet` by adding the controller tag to the `struts-config.xml` file, as shown in listing 7.10.

#### Listing 7.10 Controller tag

```

<controller processorClass=
    "com.strutsrecipes.customguardaction.controller.SecurityRequestProcessor"/>

```

To declare the response our application should take when a `SecurityViolationException` is encountered, we register an exception with exception handler by nesting an exception tag under the `Action` tag in the `struts-config.xml` file.

Finally, we register roles against the `Action` tag.

Listing 7.11 &lt;action-mapping&gt;

```
<action-mappings>
  <action    path="/unprotected"
            type="org.apache.struts.actions.ForwardAction"
            parameter="/WEB-INF/pages/unprotected.jsp"/>

  <action    path="/protected"
            roles="supervisor,appladmin" ❸
            type="org.apache.struts.actions.ForwardAction"
            parameter="/WEB-INF/pages/protected.jsp">
    // declare exception to exception handler
    <exception key="error.security"
              path="/menu.do"
              scope="request"
              type="com.strutsrecipes.
                customguardaction.exceptions.
                SecurityViolationException"/>

  </action>

  <action    path="/menu"
            type="org.apache.struts.actions.ForwardAction"
            parameter="/WEB-INF/pages/menu.jsp"/>
</action-mappings>
```

Let's step through the code to see how it all works. We start by creating a new RequestProcessor. By extending SecurityRequestProcessor from RequestProcessor, all of the default RequestProcessor hot spots execute, except processRoles. At ❶ we obtain a list of all the roles we declared at ❸. At ❷ we verify that the user possesses at least one of the roles by calling the isUserInRole private method. The isUserInRole method validates the user and role by invoking our proprietary security mechanism.

In the event that the user does not possess the authority to invoke the Action, we call the processException method ❹ to obtain the ActionForward defined declaratively to the exception handler ❺. The default RequestProcessor normally calls this method to deal with exceptions returning when processing an Action, but we are using it to deal with SecurityViolationException. We then invoke the processActionForward method ❻ to direct the response to the appropriate ActionForward.

All of the information required to handle the exception is defined in the exception tag ❺. The type attribute is used to identify the exception to be handled, and the path attribute identifies the destination page. The exception handler automatically places an ActionError in context using the value declared in

the key attribute. The receiving page can use `<html:errors/>` to present an error message to the user.

We tell the `ActionServlet` to use `SecurityRequestProcessor` instead of the default `RequestProcessor` by creating a controller tag with a `processorClass` attribute (listing 7.10).

Roles are registered against the `Action` by adding a `roles` attribute to the `Action` tag nested inside the `<action-mappings>` tag 5 (listing 7.11).

#### ◆ Discussion

The `ActionServlet` intercepts the request in the usual way. However, the `ActionServlet` employs the `SecurityRequestProcessor` instead of the default `RequestProcessor` to process the request. The `RequestProcessor` invokes all the hot spots in the usual way, except that our `SecurityRequestProcessor` hijacks the way `Actions` are authorized. Action authorization has been modified to use our own proprietary security mechanism instead of the container. Whenever a security violation is detected, the Struts exception handler is invoked, using a newly minted `SecurityViolationException`. The exception handler determines the appropriate `ActionForward` used for the request response. The user is informed of the security violation because the exception handler automatically creates an `ActionError`. The menu page picks up the error through the `<html:errors>` tag and renders the error message on to the page. All of the limitations have been addressed! We can use our proprietary security mechanism to authorize our `Actions`, and the user is sent to the menu page to attempt his request once more.

One “gotcha” rears its ugly head. The exception handler handles all exceptions the same, irrespective of their origin. This means that a `SecurityViolationException` originating from the `SecurityRequestProcessor` is treated the same as one originating from the `Action`. If you need to handle these two scenarios differently, then you need to throw a unique exception from `SecurityRequestProcessor`.

This recipe shows you how to override `Action` authorization to provide a clean way of using your own authentication mechanism. It also presents an alternative way to employ declarative exception handling.

#### ◆ Related

- 5.4—Use declarative exception handling
- 7.1—Tomcat domain authentication and Struts
- 7.4—Secure an action mapping using the container

◆ **Reference**

- [CORE] Core J2EE Patterns, Best Practices, and Design Strategies, Controlling Client Access

## 7.6 Protect areas on a page

---

◆ **Problem**

You want to protect areas on the page. Some areas on the screen need to be hidden from certain groups of users.

◆ **Background**

Web pages can contain diverse sets of information; you might not want all users to see all information. You want all users to access the page, but you would like to hide certain portions of the page from various types of users. For example, you want to hide employment information from some users, benefit information from another group of users; but make basic information available to all users.

One possible solution is to build a page for each type of user and use the `Action` to forward the user to the appropriate page for their security role. This might be fine for a small application with a very small number of roles, but it doesn't scale well as the number of roles and areas increase. As the number of pages increases, the maintenance effort quickly becomes unmanageable. A better solution is to create a single page and declare which areas are accessible by which roles. This solution reduces your maintenance effort to a single page.

The Struts Tiles framework provides an easy way to divide your screen into smaller, reusable pieces. Tiles allows you to create a mosaic of tiles which can be assembled into a larger tile. Each tile is declared with a definition tag. You specify accessibility of the tile by adding a `role` attribute to the definition tag. Only users possessing the appropriate role can see the portion of the screen rendered from a Tiles definition.

To refresh your memory on Tiles basics, we recommend you read recipe 3.12, "Creating a basic Struts Tiles page."

◆ **Recipe**

The first thing we need to do is add roles and user IDs to your container's configuration. See recipe 7.1, "Tomcat domain authentication and Struts," for complete instructions on how to do this.

The next step is to create a web page using Tiles by dividing your page into the areas you want to protect (see listing 7.12).

**Listing 7.12** user.jsp using tiles

```
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles" %>
<%@ taglib uri="/WEB-INF/struts-form.tld" prefix="form" %>
<%@ 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>
  <HEAD>
    <title><tiles:getAsString name="title"/></title>
  </HEAD>
  <h1><tiles:getAsString name="title"/>: User Information</h1>
  <body>
    <tiles:insert attribute='basic' />
    <tiles:insert attribute='personal' />
    <tiles:insert attribute='employment' />
    <tiles:insert attribute='benefits' />
    <p>
      <html:link forward="search">search</html:link>
    </p>
  </body>
</html>
```

Each of these areas becomes a tile. This example illustrates four areas; basic, personal, employment, and benefits. Each of these areas is encapsulated in a tile. In listing 7.13 we create Tile definitions.

**Listing 7.13** tiles-def.xml file

```
<!DOCTYPE tiles-definitions PUBLIC
  "-//Apache Software Foundation//DTD Tiles Configuration//EN"
  "http://jakarta.apache.org/struts/dtds/tiles-config.dtd">

<tiles-definitions>
<definition name="site.user" path="/layouts/user.jsp"> ❶
  <put name="title" value="Protect Area Recipe" />
  <put name="basic" value="site.basic" />
  <put name="personal" value="site.personal" />
  <put name="employment" value="site.employment" />
  <put name="benefits" value="site.benefits" />
</definition>
<definition name="site.basic" path="/tiles/basic.jsp" />
<definition name="site.personal" path="/tiles/personal.jsp" />
<definition name="site.employment" path="/tiles/employment.jsp"
  role="mgr"> ❷ ❸
```

```
<definition name="site.benefits"    path="/tiles/benefits.jsp"
                                role="hr"/>
</tiles-definitions>
```

In listing 7.13 we declare each tile in the `tiles-def.xml` file and assemble them into the `site.user` tile ❶. The tiles are inserted into the main tile at ❷. To limit the access to a user role we add a `role` attribute to definition tag ❸. The absence of a `role` attribute leaves the tile unprotected. That's it! You can easily add or remove roles from the tiles by modifying the definition tag in the `tiles-def.xml` file.

#### ◆ Discussion

The implementation of this recipe is simple and straightforward. First you need to analyze your page to identify the areas you want to protect. Each of these areas is encapsulated in a tile, and a master tile is used to assemble the smaller tiles. Whenever, the master tile is referenced in an `ActionForward`, the user's container defined roles are consulted before rendering the tile to the user.

There is a “gotcha” lurking in the shadows. Notice that the `role` attribute name is “role” and not “roles”. This means you cannot provide a comma-delimited list of roles. If you do, Tiles thinks that the comma is part of the role name and access is denied. You can work around this limitation by creating additional roles for the combinations you require. Note that this is not consistent with the `roles` attribute on the `Action` tag in the `struts-config.xml` file.

There are two limitations worth noting. First, you are restricted to the container security mechanism. Second, you cannot practically limit access to individual fields.

The Tiles framework is instrumental in providing modularity and reuse in the View [MVC], but this recipe shows you how the Tiles definition tag's `role` attribute implements security in a maintainable and robust fashion.

#### ◆ Related

- 3.12—Creating a basic Struts tile page
- 3.13—Using Tiles with XML definitions
- 7.1—Tomcat domain authentication
- 7.7—Protect fields

#### ◆ Reference

- [MVC] Model-View-Controller

## 7.7 Protect fields

---

### ◆ Problem

You want to protect individual fields from a group of users. You also want to make some fields private so they are only visible to the user logged on.

### ◆ Background

Circumstances arise in which you need to protect an individual field. Often you want to restrict access to a set of users possessing a security role. One option is to use the approach outlined in recipe 7.6. Using tiles certainly does the job, but it might lead to a plethora of tiles. This recipe will show you a better way to deal with fine-grained protection of fields.

In addition to showing you how to restrict field access to a group of users, we will also show you how to provide private fields. Private fields are only visible to the user logged into the application.

### ◆ Recipe

Add roles and user IDs to your container's configuration. See recipe 7.1 for complete instructions on how to do this.

In listing 7.14 we use the Struts `<logic:present>` tag to restrict access to users possessing the "mgr" role.

**Listing 7.14** Restrict access to a group of users

```
<logic:present role="mgr">
  Phone number: <bean:write name="user" property="phoneNumber"/><br>
</logic:present>
```

In listing 7.15 we use the Struts `<logic:present>` tag to restrict access to the user logged on.

**Listing 7.15** Private fields restrict access to the user logged on

```
<bean:define id="userName" name="user" ❶
  property="userId"
  type="java.lang.String"/>
<logic:present user="<%= userName %>" ❷
  Password: <bean:write name="user" property="password"/><br>
</logic:present>
```

### ◆ **Discussion**

The code samples in the recipe section present two techniques for protecting individual fields. Listing 7.14 uses the `<logic:present>` tag to ensure only managers can access the phone number field. The `present` tag `role` attribute consults the container security mechanism to ensure the user viewing the page possesses the “mgr” role. All other users will not see the phone number. The obvious limitation is that you must use the container security mechanism to authenticate the user.

Listing 7.15 presents a technique to limit access to the user currently logged on. The user’s name, password, and other information is created by the business tier and encapsulated in a user bean. This bean contains the information we want to display and the name of the user who owns that information. At ❶ we create a scripting variable to hold the owner’s user name. Again, as in listing 7.14, we use the `<logic:present>` tag to implement security. This time we use the `user` attribute ❷ to validate that the user name, represented with the `userName` scripting variable, is the same as the user logged on. The `<logic:present>` tag consults the container’s security mechanism to determine the name of the user logged on. If the name on the user bean matches the logged on user, the field is presented. Once again, this technique requires that you use the container’s security mechanism to authenticate the user.

This recipe demonstrates two new techniques for your security toolbox. Armed with these techniques, you are able to deliver fine-grained security access at the field level.

### ◆ **Related**

- 7.1—Tomcat domain authentication and Struts
- 7.6—Protect areas on a page

### ◆ **Reference**

- [CORE] Core J2EE Patterns, Best Practices, and Design Strategies, Controlling Client Access

# STRUTS RECIPES

George Franciscus and Danilo Gurovich

**T**his book is for developers and architects with real Struts work to do. It offers a broad collection of practical solutions—complete with detailed code listings—that will save you time and money. Each recipe clearly defines the problem it solves, gives you the background you need, and discusses the practical implications of adopting the solution. Many recipes point out little-known “gotchas,” which will save you from needless grief. This book introduces you to Struts best practices so you can make your applications secure, robust, and maintainable.

Techniques presented in this book have gone through the trial by fire of real-life enterprise development and deployment—you can rely on them with confidence.

## A Sampling of What's Inside

- How to automate your projects using Ant builds
- Cross validate your forms with a pluggable validator
- Unit test your app for function, performance, and coverage
- Make your application secure
- How to deal with exceptions
- Generate alternate views using PDF and XSL
- Refine your UI with Struts-Layout
- How to design a layered Struts application
- Understand the ins and outs of the Tiles Controller
- Use a DB in your message-resource
- Integration with Hibernate

**George Franciscus** is a J2EE consultant, Struts authority, and co-author of *Struts in Action*. **Danilo Gurovich** is a Manager of Web Engineering at an eCommerce Company. He has designed eCommerce and ERP/EAI Struts applications, and led teams who built them.

“A gold mine of useful information.”

—Patrick Peak, Chief Technology Officer, BrowserMedia

“Sooner or later you’re going to need every recipe—keep it by your side.”

—Cedric Dumoulin, Struts Project Management Committee Member

“... presents best practices that will turn every developer on your team into an expert.”

—Hemesh Surana, Senior Software Engineer, SeeBeyond Corporation

“... solves real-life Struts problems for you ... takes your applications to the next level.”

—Vivek Awasthi, CTO & Head of Research, ISTS Infotech Solutions



Ask the Authors



Ebook edition

[www.manning.com/franciscus](http://www.manning.com/franciscus)



ISBN 1-932394-24-9