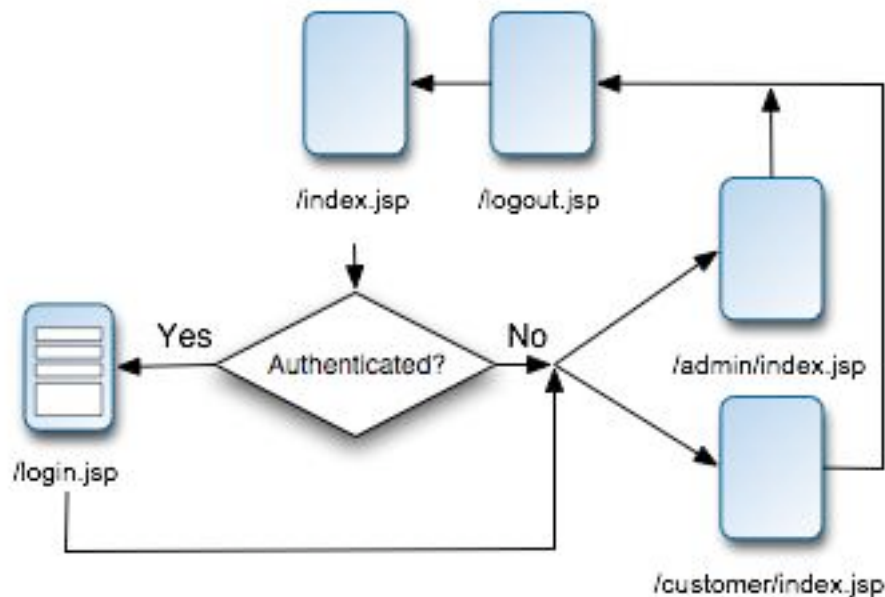


9 JAAS in Web Applications

Though the Servlet spec doesn't officially integrate with JAAS, by convention, most Servlet containers provide several JAAS-related functions: restricting access to pages in a web application, providing a framework to authenticate users, and methods to access authentication information. Pages restrictions are specified by URL patterns and a list of "role" names that the requesting user must have to access the URLs. How these role names map to `Principals` is not specified, but in Tomcat, the role names are simply the `String` names of `Principals`. The framework for authenticating users can be used to create login screens that gather a user's username and password, and then associate the authenticated user with the session. The methods for accessing authentication information allow you to programmatically verify which `Principals` a user is in, retrieve their Servlet-centric `Principals`, and access other security-related state.

9.1 The Web Application



This chapter uses a simple web application, diagramed above, with a handful of pages to demonstrate each of the above integration points between JAAS and Servlets. The web application provides a home page with links to an admin page, a customer page, and support pages to log in users, log out users, and an error page. As their names suggest, a user must be authenticated as an admin to access the admin page and a customer to access the customer



page. Also, this chapter discusses a simple custom tag library that displays its body content based on an authenticated user's `Principal` set.

9.2 Configuring JAAS with Servlets

To enable JAAS in a web application, three things must be configured. First, the web container must be configured to create a “realm” that will be used to authenticate users. The Servlet spec does not specify how this configuration is done, so it's different for each web container. Once a realm is setup, the `web.xml` file must be modified to enable the authentication framework and to include mappings of URL patterns to the `Principal` names required to access those URLs. Finally, JAAS must itself be configured to specify the `LoginModule` implementations to use when authenticating a user.

9.2.1 Configuring Realms

A realm has one responsibility: authenticate a user based on a username and password, adding “roles” to that user if authentication was successful. The Servlet spec doesn't specify how this responsibility is implemented, or very many other semantics of realms except that a realm must be able to represent roles with `String` names. Because of this looseness, each web container implementation is able to provide many different realm implementations: simple flat-file based realms, LDAP or other directory-based realms, OS authentication realms, and many other methods. Practically every web container also provides a way to use JAAS as a realm. In the instances when JAAS is used as a Servlet realm, the web container gathers a user's username and password credentials, and delegates authenticating the user to the JAAS authentication framework, using a `LoginContext` and `LoginModule` implementations.

In this chapter, we use Tomcat 5.0.28 as our web container. Tomcat is the reference implementation for the Servlet 2.4 specification, and it provides a simple way to use JAAS realms. Realms are configured in Tomcat by modifying either the system-wide `server.xml`, or the web application's uniquely named `server.xml`. In our example, we modify the second to keep our application as self-contained as possible.

Modifying `server.xml`

Web application `server.xml` files are stored in `<tomcat dir>/conf/Catalina/localhost/` and follow the convention of being named the same as their corresponding web application. Our web application is named `jaas-book-chp09`, so the `server.xml` file we're interested in is found at `<tomcat dir>/conf/Catalina/localhost/jaas-book-chp09.xml`. The content of the file is below:

```
<?xml version="1.0"?>
<Context path="/jaas-book-chp09" docBase="~/tomcat/webapps/jaas-book-
chp09"
    debug="0" reloadable="true">
    <Realm className="org.apache.catalina.realm.JAASRealm" #1
        appName="chp09" #2
```



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5
License: <http://creativecommons.org/licenses/by-nc/2.5/>

```
userClassNames="chp09.UserPrincipal" |#3
roleClassNames="chp04.UserGroupPrincipal" |#3
useContextClassLoader="false"/> #4
</Context>
```

(annotation) <#1: The realm tag specifies that Tomcat's `JAASRealm` will be used to authenticate users.

(annotation) <#2: `appName` is used to specify which `LoginModule` group will be used to authenticate users. "chp09" is the application name that will be passed into the `LoginContext` constructor. So, we'll have to ensure that our `javax.security.login.Configuration` can return an `AppConfigurationEntry` array for that application name.>

(annotation) <#3: The `userClassNames` and `roleClassNames` attributes specify which `Principal` implementations will be used to represent the user `Principal` and the role `Principals`. Once a user has been authenticated, creating a `Subject` with `Principals`, the user `Principal` is used when the web container looks up the `Subject`'s user, for example, when looking up the value for `HttpServletRequest`'s `getRemoteUser()` or `getUserPrincipal()`. The role `Principals` are used to lookup the `Subject`'s roles, for example, when resolving if a user is in a role for `HttpServletRequest`'s `isUserInRole`.>

(annotation) <#4: setting this attribute to `false` tells Tomcat to use the web application's class loader instead of the server class loader. The `LoginModule` implementation we'll be using (`DbLoginModule` and `TomcatLoginModule`) will be stored in the web application's lib directory, meaning that the server level class loader will not be able to find it.>

With `chp09-server.xml` in place, Tomcat will create an authentication realm that will be used once we configure `web.xml` to enable security.

9.2.2 Configuring web.xml

As with practically every other feature in Servlets, enabling authentication and authorization is done by modifying the `web.xml` file. Three tags are used to enable authentication, specify URL access restrictions, and define the available user roles, or `Principals`.

Enabling Authentication

Web containers may provide five types of authentication schemes: BASIC and DIGEST, which use the built in username and password dialog box for HTTP; FORM, which uses custom JSP pages with standard form action and element names; CLIENT-CERT, which uses digital certificates; and any proprietary mechanisms that the web container provides. In this book, we'll only cover the use of the FORM method because it covers the widest range of cases and allows for a fair amount of customization.

In our example web application, configuring authorization in `web.xml` is done with the following element:

```
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>Chp09 Realm</realm-name> #1
  <form-login-config> #2
    <form-login-page>/login.jsp</form-login-page>
    <form-error-page>/login-error.jsp</form-error-page>
  </form-login-config>
</login-config>
```

(annotation <#1: the realm name is used purely for display purposes, mostly for web application development tools.>



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

(annotation) <#2: this element and its sub-elements specify the location of the login page to use and JSP page to use when authentication errors occur. The error page is used for invalid login attempts and when errors occur logging in. A different page, covered below, is used when a user attempts to access a URL they're not authorized to view.>

Locking Down URLs with security-constraints

To specify access control for parts of your web application, you use any number of `security-constraint` elements. The `security-constraint` element specifies one or more URL patterns and the Principals, represented by role names, a user is required to have to access those URLs. When an unauthenticated user attempts to request one of the protected URL, the web container redirects the request to the login page specified in the `login-config` element in `web.xml`.

A URL pattern can be an exact match, like `/admin/userlist.jsp`, or a pattern, like `/admin/*`. The first pattern specifies a single page, while the second specifies any URLs that begin with `/admin`. The patterns are all relative to the web application context.

The role names specified may either be the String name of a Principal, or the special role name `*`, which is shorthand for any role. When a user requests a URL specified by the `security-constraints` URL patterns, the user's Subject must have one of the roles specified, or access is denied.

The `security-constraint` elements used in our example `web.xml` are below:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Admin Page (Chp09)</web-resource-name>
    <url-pattern>/admin/*</url-pattern> #1
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name> #2
  </auth-constraint>
</security-constraint>

<security-constraint>
  <web-resource-collection>
    <web-resource-name>Customer Page (Chp09)</web-resource-name>
    <url-pattern>/customer/*</url-pattern> #1
  </web-resource-collection>
  <auth-constraint>
    <role-name>customer</role-name> #2
  </auth-constraint>
</security-constraint>
```

(annotation) <#1: these tags specify the URLs that have access restrictions. The URL patterns used in this example cover all URLs that begin with `/admin` and all URLs that begin with `/customer`. Each URL is relative to the web application context.>

(annotation) <#2: the Principals required to access restricted URLs are specified in the `auth-constraint` element by `role-name` elements. Any number of `role-name` elements may be specified. Each `role-name` element specifies one Principal, by name, that may access the restricted URL. If none are specified, then no users may access the restricted URLs, preventing all access to the URLs.>

Specifying the roles Used



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

Finally, you must specify all of the role names that are referenced in `web.xml` with one `security-role` element per role. Specifying each role that's used seems tedious, but it allows tools to get a list of roles used and provides crude referential integrity¹.

The `security-role` elements for our example web application are below:

```
<security-role>
  <description>
    Role required to see admin pages.
  </description>
  <role-name>admin</role-name>
</security-role>

<security-role>
  <description>
    Role required to see customer pages.
  </description>
  <role-name>customer</role-name>
</security-role>
```

Other Settings and Entire `web.xml` Listing

The entire `web.xml` listing is below, with code notations for the remainder of the settings:

```
<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
  <display-name>jaas-book</display-name>
  <description>JAAS Book, Chapter 9</description>

  <servlet>
    <servlet-name>InitServlet</servlet-name> #1
    <servlet-class>chp09.StartupServlet</servlet-class>
    <load-on-startup>0</load-on-startup>
  </servlet>

  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

  <error-page> #2
    <error-code>403</error-code>
    <location>/access-denied.jsp</location>
  </error-page>
```

¹ Tomcat allows you to skip specifying the `security-role` element, but logs an error if you omit them. Other web containers may not be so forgiving.



```
<taglib> #3
  <taglib-uri>auth-tags</taglib-uri>
  <taglib-location>/WEB-INF/auth-tags.tld</taglib-location>
</taglib>

<security-constraint>
  <web-resource-collection>
    <web-resource-name>Admin Page (Chp09)</web-resource-name>
    <url-pattern>/admin/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name>
  </auth-constraint>
</security-constraint>

<security-constraint>
  <web-resource-collection>
    <web-resource-name>Customer Page (Chp09)</web-resource-name>
    <url-pattern>/customer/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>customer</role-name>
  </auth-constraint>
</security-constraint>

<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>Chp09 Realm</realm-name>
  <form-login-config>
    <form-login-page>/login.jsp</form-login-page>
    <form-error-page>/login-error.jsp</form-error-page>
  </form-login-config>
</login-config>

<security-role>
  <description>
    Role required to see admin pages.
  </description>
  <role-name>admin</role-name>
</security-role>

<security-role>
  <description>
    Role required to see customer pages.
  </description>
  <role-name>customer</role-name>
</security-role>

</web-app>
```

(annotation) <#1 [InitServlet]: the `InitServlet` is used to configure `DbConfiguration` and configure logging. See section XXX below for further discussion of the `InitServlet`.



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License: <http://creativecommons.org/licenses/by-nc/2.5/>

(annotation) <#2 [error-page]: this `error-page` element specified the page to use when a user attempts to access a URL they are not authorized to view. If this page is not specified, a generic error page is used instead. We'll see this page in action below when we walk through the web application's pages.>
(annotation) <#3 [taglib]: this specifies the custom tag library that contains the role display tag, seen later in this chapter.>

9.2.3 The JSP Pages

Our `web.xml` file references several JSP pages: `login.jsp`, `login-error.jsp`, and `access-denied.jsp`. The `login.jsp` page is used to gather username and password credentials, and the `login-error.jsp` when a user fails authentication or an error occurs authenticating a user. The last page, `access-denied.jsp`, is used when an authenticated user attempts to access a page that requires a Principal the user doesn't have.

In addition to these 3 pages, 3 other pages are included in the example web application: `logout.jsp` which invalidates the user's session, thus logging out the user; the top level `index.jsp` which has links to the Admin and Customer page, and a link to `logout.jsp`; and an `index.jsp` pages for the admin and customer sub-directories.

Of these JSP pages, the only noteworthy pages are `login.jsp` and `logout.jsp`. At the end of this chapter, in the `RoleTag` section, we'll go over the top-level `index.jsp`. The other JSPs are available in the accompanying source, and we won't go over them here.

login.jsp

The `login.jsp` page contains the custom login form used in our web application. The Servlet spec requires that the action for the login form be `j_security_check`. Also, the form input field for the username must have the name `j_username`, and the password input field must have the name `j_password`. Requiring these names makes implementing Servlet security a little easier for web container vendors, and isn't too much of an inconvenience for developers.

The source for `login.jsp` is listed below:

```
<html>
<head><title>Chapter 09 Login</title></head>
<body>

<form method="POST" action="j_security_check">
<p>Username: <input type="text" name="j_username"/></p>
<p>Password: <input type="password" name="j_password"/></p>
<input type="submit" value="Login"/>
</form>

</body>
</html>
```

logout.jsp

`logout.jsp` is interesting because it contains code that logs out the currently authenticated user. The convention for logging out users is to invalidate the user's session. The `logout.jsp` does this with a small inline code fragment; this code could be done in a Servlet, Struts Action, or other non-JSP code just as easily.



The code listing is below:

```
<html>
<head><title>Chp09 Logout</title></head>
<body>

<%
try {
    session.invalidate();
}
catch(IllegalStateException e) {
    // we don't care
}
%>

<p>You've been logged out.</p>
<p><a href="<%= request.getContextPath() %>/index.jsp">Home</a></p>

</body>
</html>
```

9.2.4 Configuring JAAS

JAAS authorization services must be enabled for Tomcat's JAASRealm to work. This is done through the standard methods of using either VM arguments, modifying the VM's security.properties file, or programmatically setting the javax.security.auth.Configuration to use. In our application, we use DbConfiguration, from chapter 4, as our Configuration implementation. To make the web application more self-contained, we programmatically set the Configuration by calling DbConfiguration's init() method in a startup Servlet.

A startup Servlet is a Servlet that the web container loads immediately after loading the web application. By overriding the init() method, you can programmatically configure your web application. We use this pattern to configure DbConfiguration and to configure JDK logging in our web application. The Servlet is specified and configured in web.xml.

The code for the startup Servlet is below:

```
package chp09;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;

import util.LoggerInit;

import chp04.DbConfiguration;

public class StartupServlet
    extends HttpServlet {

    public void init(ServletConfig config) throws ServletException {
```



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License: <http://creativecommons.org/licenses/by-nc/2.5/>


```
    DbConfiguration.init();
    LoggerInit.init();
}
}
```

When our example web application loads, before serving any requests, the above code is executed, configuring `DbConfiguration` to be used by JAAS.

TomcatLoginModule

With JAAS configured, we now need to specify a `LoginModule` to use when authenticating users. We can re-use the functionality of chapter 4's `DbLoginModule`, allowing us to specify and manage users and their role-Principals in a database. However, we need to add an additional `Principal` that represents the authenticated Subject's user. Effectively, this `Principal` simply wraps the username entered in the login page². As mentioned above, this `Principal` is returned from `HttpServletRequest`'s `getUserPrincipal()` method.

To add the user `Principal`, we create a new `LoginModule` implementation, `TomcatLoginModule`, which extends `DbLoginModule` and overrides the `commit()` method. The result is that the `Subject` is given all of the `Principals` assigned to it in the database in addition to the special user `Principal` needed by Tomcat.

```
package chp09;

import javax.security.auth.login.LoginException;

import chp04.DbLoginModule;

public class TomcatLoginModule
    extends DbLoginModule {

    public boolean commit() throws LoginException {
        if (super.commit()) {
            UserPrincipal userP = new UserPrincipal(getUsername()); #1
            getSubject().getPrincipals().add(userP);
            getPrincipalsAdded().add(userP);
            return true;
        } else {
            return false;
        }
    }
}

(annotation) <#1: chp09.UserPrincipal is the Principal we specified when configuring the realm for
Tomcat. The user Principal simply wraps the username provided by the user.>
```

Database Setup

² If a different mapping makes more sense for your web application, the Servlet spec does not require that the user `Principal`'s name is the same as the username. In our example, and most web applications, wrapping the username is sufficient.



To support DbConfiguration and TomcatLoginModule, we setup the same database we used in chapter 4, seeding it with rows for the customer and admin users, and then adding the corresponding RolePrincipal to each:

```
INSERT INTO app_configuration VALUES
('chp09', 'chp09.TomcatLoginModule', 'REQUIRED');
```

```
INSERT INTO db_user VALUES
('admin-user-id', 'admin', 'secret');
```

```
INSERT INTO db_user VALUES
('customer-user-id', 'customer', 'secret');
```

```
INSERT INTO principal VALUES
('admin-principal-id',
'admin',
'chp09.RolePrincipal');
```

```
INSERT INTO principal_db_user VALUES
('admin-user-id', 'admin-principal-id');
```

```
INSERT INTO principal VALUES
('customer-principal-id',
'customer',
'chp09.RolePrincipal');
```

```
INSERT INTO principal_db_user VALUES
('customer-user-id', 'customer-principal-id');
```

9.3 The Web Application

Once the above configurations and other setup are done, we're ready to use our example web application, demonstrating how JAAS can be used with Servlets to restrict access to sections of the web application. This section walks through the pages of the example web application, demonstrating how the web container uses the configuration and code in the preceding sections.

9.3.1 URL Access Control

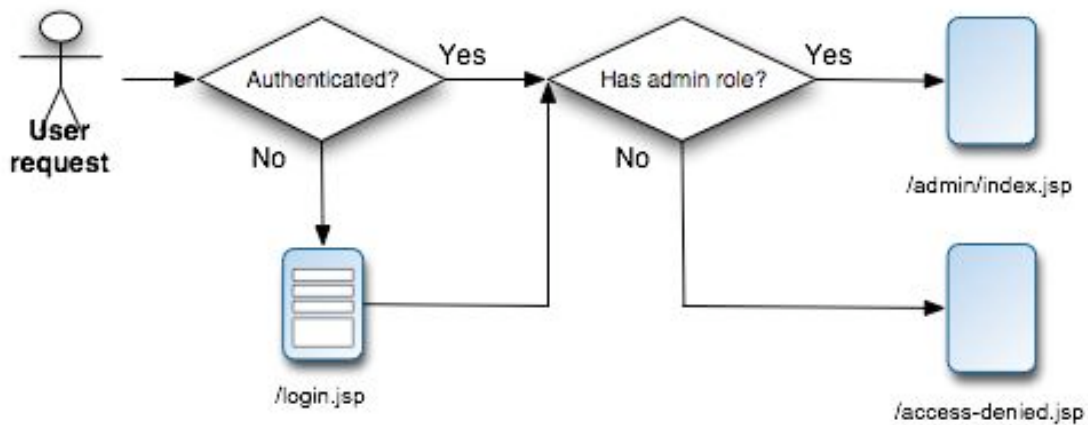
Each time a restricted URL is requested, the web container first ensures that a user is logged in, redirecting the request to the `login.jsp` if there is no user associated with the current session. Once a user has been logged in, the web container will then see if the user belongs to any of the role-Principals that are allowed to access the restricted URL, specified by the `security-constraint` element in `web.xml`. If the user does belong to one of those roles, they're allowed to access the URL. Otherwise, if the user does not belong to one of those roles, they're forwarded to the 403 error page, specified by the `error-page` element in `web.xml`.

The diagram below illustrates this flow:



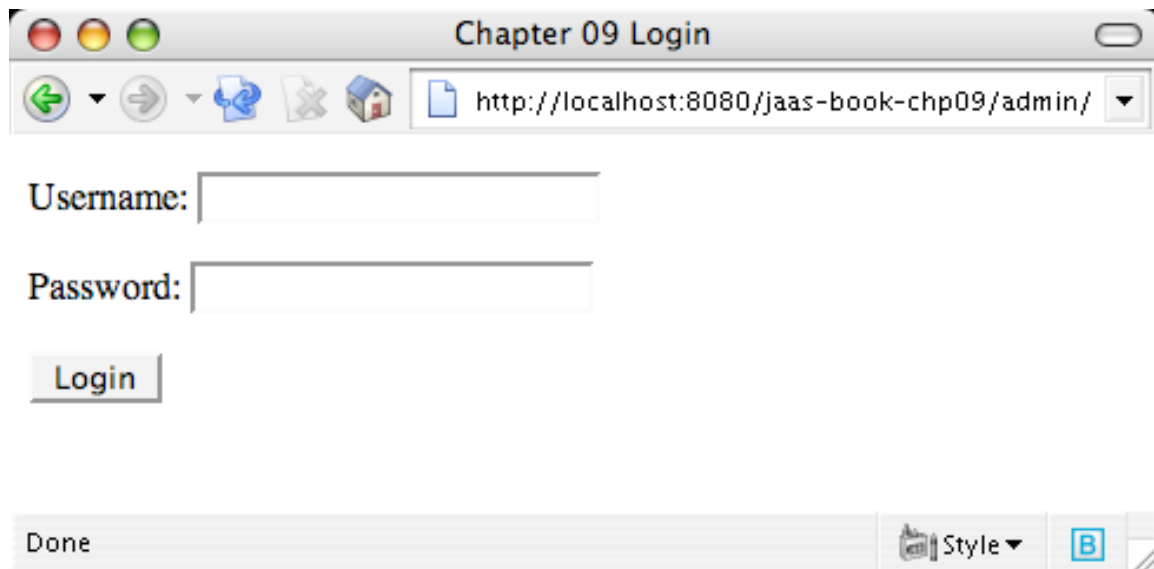
This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>



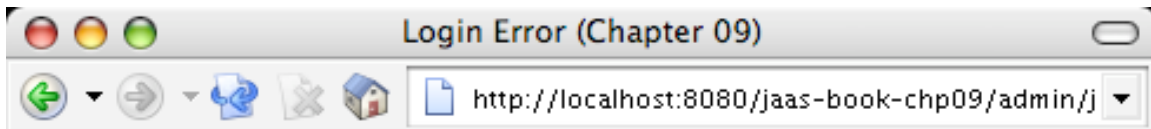
9.3.2 Example: Accessing /admin/index.jsp

Let's suppose that we have a user who wants to access the admin page, /admin/index.jsp in our web application. The user hasn't been authenticated yet, so when he first attempts to access the page, he'll be redirected to login.jsp:

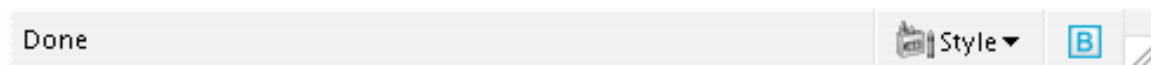


The user types in the correct username, admin, but uses the incorrect password. When TomcatLoginModule is invoked to authenticate the user, it throws a LoginException from its login() method, causing the web container to forward to the login-error.jsp page:

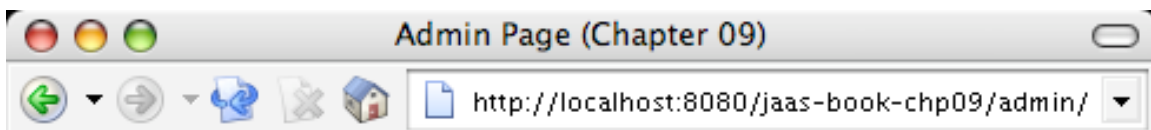




Error logging in. [Back home.](#)



The user realizes their mistake, goes back to the login page, and enters the correct password. With the correct password, TomcatLoginModule's `login()` method returns `true`, causing `commit()` to be called, successfully authenticating the user and adding the required `admin RolePrincipal` to the user's `Subject`. Because the user has now been authenticated and has the required `admin RolePrincipal`, the web container forwards them to access the originally requested page, `/admin/index.jsp`:

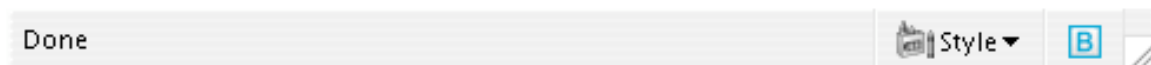


Admin Page

Username: admin

Servlet Principal: `GenericPrincipal[admin(admin,)]`

[Home](#)

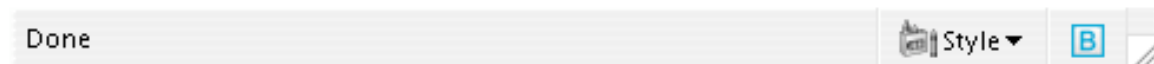
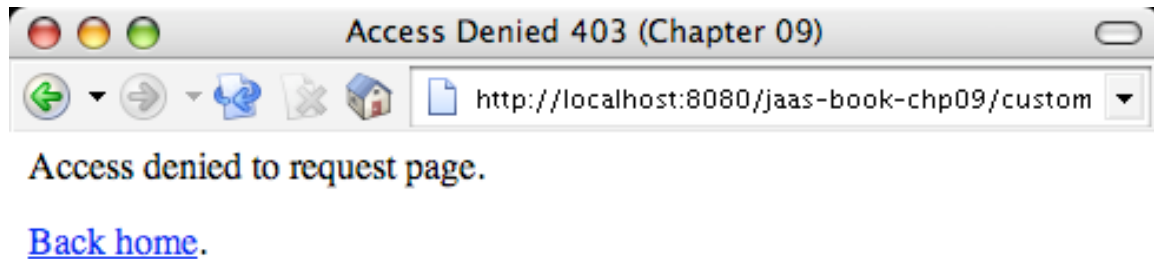


The page displays the username and the `toString()` value of the `Principal` returned from `HttpRequest`'s method `getUserPrincipal()`.



9.3.3 Example: Accessing /customer/index.jsp

Next, the user attempts to access the customer page by going to the URL /customer/index.jsp. The security-constraint for this page requires the user to have the customer RolePrincipal which the admin user does not have. So, the web container redirects the user to 403 error page as specified by the error-page element in web.xml, access-denied.jsp:



9.4 RolesTag

To demonstrate programmatically some of a Servlet's JAAS-related methods, we'll develop a custom tag library that displays the body of the tag only if the authenticated user has one of the Principals the tag specifies. This is a very common scenario. For example, we may have a section of the page that we only want users with the admin RolePrincipal to see.

The tag's only attribute, `roles`, specifies a comma-separated list of Principals that the user must have to see the body of the tag. The body of the tag will be displayed if the authenticated user has at least one of the Principals specified by the `roles` attribute.

9.4.2 A Pure Scriptlet Implementation

To appreciate the utility of having a tag to perform role checks, we'll first look at how we'd check for a user's roles purely with JSP scriptlets:

```
<%@ taglib uri="auth-tags" prefix="auth" %>
<html>
<head><title>Index</title></head>
<body>

<a href="admin">Admin Page</a> |
<a href="customer">Customer Page</a> |
```



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License: <http://creativecommons.org/licenses/by-nc/2.5/>

```
<a href="logout.jsp">Logout</a>

<% if (request.isUserInRole("customer")) { %> #1
<p>Only the <b>customer</b> role sees this.</p>
<% } %>

<% if (request.isUserInRole("admin") ||
      request.isUserInRole("superadmin")) { %> #2
<p>Only the <b>admin</b> role sees this.</p>
<% } %>

<p>Principals: <%= request.getUserPrincipal() %>.</p>

</body>
</html>
```

(annotation) <#1 HttpServletRequest provides the method isUserInRole which returns true if the currently logged in user has the passed in role.>

(annotation) <#2 isUserInRole only accepts one role at a time, so to check for multiple roles, you have to or together a call for each role.>

While using a pure scriptlet approach doesn't require any extra code or configuration files (as the below taglib approach does), it suffers a key disadvantages: lack of abstraction. Instead of layer how our web application does security checks, we're directly coding that method into our JSP page. If we later decide to check for a user's role using a different way than using the isUserInRole() method, we'll have a lot of JSP pages to change. Aside from that, as with most scriptlet code, it just looks ugly.

9.4.1 Using RolesTag

The top-level index.jsp uses demonstrates the use of this tag:

```
<%@ taglib uri="auth-tags" prefix="auth" %>
<html>
<head><title>Chapter 09 Index</title></head>
<body>

<a href="admin">Admin Page</a> |
<a href="customer">Customer Page</a> |
<a href="logout.jsp">Logout</a>

<auth:roles roles="customer">
<p>Only the <b>customer</b> role sees this.</p>
</auth:roles>

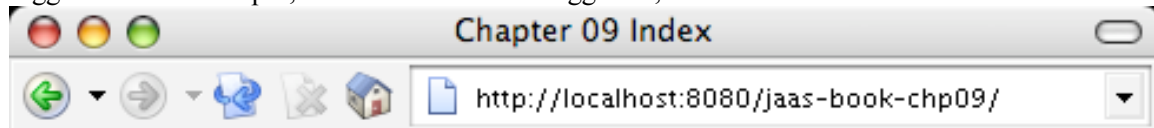
<auth:roles roles="admin">
<p>Only the <b>admin</b> role sees this.</p>
</auth:roles>

<p>Principals: <%= request.getUserPrincipal() %>.</p>
```



```
</body>
</html>
```

The `RolesTag` is used twice in this page. The first instance creates a section of the JSP page that will only be displayed when a user with the `customer RolePrincipal` is logged in, while the second displays its body content only when a user with the `admin RolePrincipal` is logged in. For example, when a customer is logged in, the JSP will be rendered as:



[Admin Page](#) | [Customer Page](#) | [Logout](#)

Only the **customer** role sees this.

Principals: `GenericPrincipal[customer(customer,)]`.



9.4.2 *RolesTag's TLD*

The following tag library descriptor configures the `RolesTag`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library
1.1//EN" "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>auth</shortname>
  <uri>/WEB-INF/auth-tags.tld</uri>
  <tag>
    <name>roles</name>
    <tagclass>chp09.RolesTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <attribute>
      <name>roles</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>
</taglib>
```



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5
License: <http://creativecommons.org/licenses/by-nc/2.5/>

We saw the tag library included with the `taglib` element in the complete listing of the web application's `web.xml` above.

RolesTag Code

Once `RolesTag` verifies that a value for the `roles` attribute was specified, it splits the list of roles in an array of names. `RolesTag` then iterates over this array of names, passing each name to `isUserInRole()` on `HttpServletRequest`. If `isUserInRole()` returns `true`, `RolesTag` returns `INCLUDE_BODY`, causing the body of the tag to be displayed. Otherwise, if the user does not have one of the roles required, `SKIP_BODY` is returned causing the body of the tag to be omitted from the rendered JSP. If the user is not even authenticated, `isUserInRole()` will return `false` each time, causing the body of the tag to be omitted to un-authenticated users as well.

```
package chp09;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.jsp.tagext.TagSupport;

public class RolesTag
    extends TagSupport {

    public int doStartTag() {
        if (roles_ != null || roles_.length() != 0) {
            boolean userHasRole = false;
            HttpServletRequest request = (HttpServletRequest) pageContext
                .getRequest();
            String[] splitRoles = roles_.split(",");
            for (int i = 0; i < splitRoles.length; i++) {
                String role = splitRoles[i];
                if (request.isUserInRole(role.trim())) {
                    return EVAL_BODY_INCLUDE;
                }
            }
        }
        return SKIP_BODY;
    }

    public String getRoles() {
        return roles_;
    }

    public void setRoles(String roles) {
        roles_ = roles;
    }

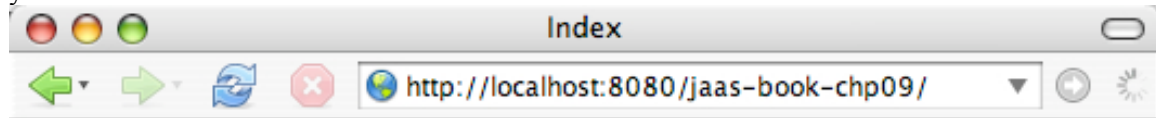
    private String roles_;
}
```



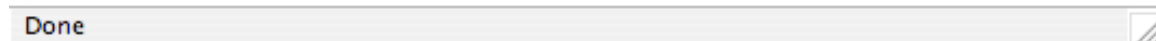
Running the Example Web Application

To deploy the example web application for this chapter, change directories to the source code directory, and execute the command `ant deploy-chp09`. This will configure the database for you, compile the required code, and deploy the web application to Tomcat.

Once you start Tomcat, you'll be able to load the example web application by going to the URL `http://localhost:8080/jaas-book-chp09/` in your browser. The first page you'll see is below:



Principals: null.



From the index page, you can attempt to access both the Admin and Customer page. Once you click on either link, you'll be redirected to the login page which will prompt you for a username and password as seen in the screenshots in the previous sections. To login as an admin, use the credentials `admin/secret`; to login as a customer, use the credentials `customer/secret`.

Summary

With a good understanding of JAAS under our belts, we were ready to start using JAAS in a web application. The first step using JAAS in a web application was modifying the application's `web.xml` file to enable authentication. Once authentication was enabled, we learned how to customize the different JSP pages used by the web container to log a user in and display error messages. With the configuration under our belts, we went over two simple ways to secure parts of any web applications: URL access restrictions and a custom tag library that conditionally displays its JSP body according to the roles the logged in `Subject` has.



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>