Implement roles-based authorization

Using products from the IBM community software portfolio and the open source community

Scott Crowther
Abe Guerra
Tamer Nassar

April 14, 2009

Learn how to implement a dynamic user interface through user authentication. Authentication is often the requirement for applications with multiple groups of users. Each group may require access to application functionality that may need to be withheld from other groups. The authentication mechanism must validate user credentials and control access to application functionality based on the user's credentials. This article shows how to implement a basic authentication mechanism using OpenLDAP and Tomcat. It compares the OpenLDAP and Tomcat implementation to an OpenLDAP and WASCE implementation. And finally, code examples show the implementation of the dynamic UI using Java code and JSTL.

Introduction

Web applications usually need to control the functionality made available to certain groups of users. For example, an application might have a group of expert users that may be allowed to upload data files to the application server. However, the application might need to prevent a group of casual users from doing the same.

Controlling access to application functionality is a two-step process. First, an authentication mechanism is used to identify the current user. Second, an authorization mechanism is used to control access to application functionality. An implementation of this process could involve multiple copies of the application's Web pages, where each group of users has an associated copy of a Web page. Another implementation, and the implementation preferred and used in this article, is to have one copy of a Web page which dynamically renders appropriate functionality through the use of Java and JSTL.

You can use many technologies to implement authorization. Given the sophistication of applications, the implementation in this article uses an application server. In particular, we use Tomcat 6 and WebSphere Application Server Community Edition 2.1 (WASCE). We prefer a
WASCE implementation because of the integration with Apache Geronimo and its associated J2EE platform.

This article’s implementation uses the Lightweight Directory Access Protocol (LDAP). An LDAP implementation provides a robust mechanism for managing the users of an application. There are several LDAP servers that can be used. Our requirement was an LDAP server which was robust, stable, and open source. Our research led us to OpenLDAP as the best server, given these requirements.

Development environment

This article is for a Java-based Web application. In particular, we’re using Java 5, JSF 1.2, and JSTL 1.2. Our IDE is the Eclipse Ganymede release with the Web Tools Platform (WTP). (See Related topics for download links.)

Once you have installed Java and Eclipse, you can begin setting up your development environment. First, you need to create the necessary project libraries by doing the following:

- Click Windows from the menu bar.
- Select Preferences.
- Double click Web in the Preferences window.
- Double click JavaServer Faces Tools (underneath Web).
- Select Libraries (underneath JavaServer Faces).
- Click New to add each of the following three jars:
  - jstl-1.2.jar
  - jsf-api.jar
  - jsf-impl.jar.

Your Preferences window should now look similar to what is shown in Figure 1.
Second, you'll need to create a new Eclipse project by doing the following:

- Click **File** from the menu bar.
- Hover over **New** in the drop-down menu.
- Select **Dynamic Web Project** from the secondary drop-down menu (you may need to find the option through the **Other** option if Dynamic Web Project is not listed).

### Using Tomcat 6

Since we're doing a Tomcat 6 and WASCE deployment, we'll need to split at this point to account for the different servers. The first step in creating the project is providing a name, target runtime, module version, and configuration. It should look similar to what is shown in Figure 2 (based on your setup and Tomcat in this instance).
Figure 2. New dynamic Web project

The second step will ask you to configure your Web module setting. You can leave the default settings on this step. The third step will ask you to define your project’s JSF capabilities. These should be set similar to Figure 3.
Security

Now let’s start defining some security. You will need to define a security constraint for the Web application. We’ll define a constraint which secures everything inside /WebContent/private against unauthenticated users (we’ll let the UI handle roles). To create this constraint, add the code in Listing 1 to /WebContent/WEB-INF/web.xml:

Figure 3. JSF capabilities
Listing 1. Sample code of web.xml

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>
            Areas with authentication required
        </web-resource-name>
        <url-pattern> /faces/private/* </url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name> * </role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee> NONE </transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

Next, you need to define a log-in mechanism. We’ll define form-based authentication and will define the form’s location. Add the code in Listing 2 to web.xml:

Listing 2. (Continued) Sample code of web.xml

```
<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>LDAP</realm-name>
    <form-login-config>
        <form-login-page>/public/login.html</form-login-page>
        <form-error-page>/public/loginError.html</form-error-page>
    </form-login-config>
</login-config>
```

The last update we need to make to web.xml is for security roles. However, we are handling roles through JSTL, so we can define the constraint for all roles by adding the code in Listing 3.

Listing 3. (Continued) Sample code of web.xml

```
<security-role>
    <description>Authenticated Users</description>
    <role-name>*</role-name>
</security-role>
```

Now that you’ve defined the security for your application, you need to connect it to an LDAP solution which manages the users of your Web application. You can define your LDAP server in /WebContent/META-INF/context.xml. The file should look like Listing 4.

Listing 4. context.xml

```
<?xml version='1.0' encoding='utf-8'?>
<Context reloadable="true" useNaming="true" debug="3">
    <Realm className="org.apache.catalina.realm.JNDIRealm" debug="99" connectionURL="ldap://hostname:389"
           userBase="ou=people,dc=tnc,dc=org" userSubtree="true"
           userSearch="(&(uid={0})(objectclass=inetOrgPerson))"
           roleBase="(uniqueMember={0})"
           roleSearchSubtree="false"
           />
</Context>
```
Using WASCE

Now we will create the roles-based navigation project using WASCE. In the following steps we will show you how to create and configure the project and how to set up Security LDAP realm in WASCE.

Create a new project by doing the following:

- Click **File** from the menu bar.
- Select **Dynamic Web Project**.
- Type the project name: **RolesBasedNavWASCE**
- Select **IBM WASCE** as runtime.
- Under Dynamic Web module, select Version 2.5
- Choose JavaServer Faces for the configuration field.

**Figure 4. New dynamic Web project**

- Click **Modify** and add **WASCE Deployment**.
Figure 5. Project facets

Define your project’s JSF capabilities. It should be set similar to this:

Figure 6. JSF

Setting up Security LDAP realm in WASCE

Below is how you configure WASCE to connect to LDAP. LDAP could be installed on the same server or on a different server. WASCE will be able to connect to it as long as you specify the right connection URL.

- Access your WASCE admin console. Go to http://SERVERNAME:8080/console
• Log in as username/password: system/manager.
• Click **Security Realms**.

**Figure 7. Security**

• Type a name for the Security realm: `ldap-realm`, and pick **LDAP Realm** as the Realm Type.
• Click **Next**.
You need to enter your LDAP information in the Security Realm page, so WASCE can communicate with LDAP. Here are the fields values we entered based on our LDAP setup.

- **Initial Context Factory**: `com.sun.jndi.ldap.LdapCtxFactory`
- **Connection URL**: "it should point to the LDAP server"
- **Connect Username**: `cn=ldaproot,dc=tnc,dc=org`
- **Connect Password**: `xxxxxxxx`
• Confirm password: xxxxxxx
• Connect Protocol: "LEAVE IT BLANK"
• Authentication: simple
• User Base: ou=people,dc=tnc,dc=org
• User Search Matching: uid={0}
• User Search Subtree:false
• Role Base: ou=people,dc=tnc,dc=org
• Role Name: cn
• Role User Search String: (objectClass={0})
• Role Search Subtree: false

Note: (Some of these values might change based on your LDAP setup.)

• Click Next.
• To test your LDAP connection, you will need to enter a username/password for a created LDAP user.
• After a successful test, click Deploy Realm.

To complete your security configuration, you need to define a constraint which secures everything inside /WebContent/private. To create this constraint, add the following to the web.xml:

**Listing 5. Security constraint inside web.xml**

```xml
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>Not required for FORM auth</realm-name>
  <form-login-config>
    <form-login-page>/faces/public/login.jsp</form-login-page>
    <form-error-page>/faces/public/loginerror.jsp</form-error-page>
  </form-login-config>
</login-config>

<security-role>
  <description>Authenticated users</description>
  <role-name>*</role-name>
</security-role>

<security-constraint>
  <web-resource-collection>
    <web-resource-name>Areas with authentication required</web-resource-name>
    <url-pattern>/faces/private/*</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>*</role-name>
  </auth-constraint>
</security-constraint>
```

Also, you need to add the security-realm to geronimo-web.xml as shown in Listing 6.
Listing 6. geronimo-web.xml

```xml
<security-realm-name>ldap-realm</security-realm-name>
<security>
  <role-mappings>
    <role role-name="*">
      <principal name="admin" class="org.apache.geronimo.security.realm.providers.GeronimoGroupPrincipal"/>
    </role>
  </role-mappings>
</security>
```

The project and file structure will be the same as the Tomcat project, but the main difference is that you are taking advantage of using WASCE to authenticate with open LDAP.

The advantages of using WASCE

Figure 9. WASCE

There are several advantages to using WASCE over Tomcat, including:

- WAS CE supports the full J2EE stack which includes: a Servlet container, an EJB container, a messaging provider (JMS), and a Java Connector (JCA) container.
- Services such as Security and Authentication, Web Services, JMS, and so on are provided by the Geronimo Plug-in Kernel.
- EJB and JPA are supported.
- Additional plug-ins can be applied to the Kernel and accessed rather than embedded in application code.
- Clustering provides increased reliability and performance.
- Transactional database (Apache Derby) is included.
- Centralized configuration and control
- Runtime customization
- Centralized user management
- Eclipse plug-in allows for deployment and debugging (even remotely).
- Migration articles are available to help move from Tomcat/JBoss to WASCE.
- IBM support is available.

User Interface

The application pages will exist in two directories. The private pages, requiring security, will exist in /WebContent/private. The public pages, with no required security, will exist in /WebContent/public.
The public directory will contain html pages which require no authentication. They include the log-in page and error pages. Create the log-in page by doing the following:

- Right click public directory.
- Hover over New in the menu.
- Click HTML on the second menu.
- Set the name to log in, and click Finish.

Use the default authentication mechanism, which requires you to define the log-in form’s action as j_security_check, and use j_username and j_password. The resulting html file should look like Listing 7:

**Listing 7. Log-in page**

```html
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Login</title>
</head>
<body>
<form method="POST" action="j_security_check">
<table border="0" cellspacing="5">
<tr>
<td align="right">User ID</td>
<td align="left"><input type="text" name="j_username"></td>
</tr>
<tr>
<td colspan="2" align="center"><input type="submit" value="Submit"></td>
</tr>
</table>
</form>
</body>
</html>
```

Similarly, create another HTML file in the public directory and name it loginError.html. This will be the page which is displayed when a user attempts to log in with invalid credentials. It can be as simple as Listing 8:

**Listing 8. Log-in error page**

```html
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Login Error</title>
</head>
<body>
Your login attempt has failed.
</body>
</html>
```

You'll also want to create a similar page to display an error message when a user attempts to access functionality for which he or she is not authorized. It, too, can be a very simple page, as shown in Listing 9.
In this example, we're going to use a simple home page that will simulate links to application functionality. We'll also create .jsp pages which simulate the different types of functionality in the application. All of these files should exist in the protected area (/WebContent/private). The home page will use tag libraries, which are imported like Listing 10:

```
<%@taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
```

The .jsp pages also use a resource bundle so content can be externalized:

```
<f:loadBundle basename="com.ibm.test.messages" var="msg" />
```

The .jsp pages are dependent on back-end Java code for the authentication of users. Once users are authenticated against the LDAP directory, their credentials are persisted using Java beans. The .jsp can reference the bean like Listing 11:

```
<jsp:useBean id="userBean" scope="session" class="com.ibm.test.UserBean" />
<h:outputText value="#{msg.welcome} #{userBean.user.firstName} #{userBean.user.lastName}" />
```

In the above snippet, a welcome message is displayed by combining content from the resource bundle (referenced by 'msg') and the first and last names from the Java bean (referenced by userBean). At this point, you can start filtering based on user roles. The user's role will also be contained in the Java bean. This allows the .jsp page to check the user role and take appropriate action. For example, you can control the functionality links that are displayed based on the current user's role.

```
<c:if test="${userBean.user.userRole == 'Admin'}">
  <br /><a href="../private/admin.jsp">Admin Functionality</a>
</c:if>
```

The code snippet above in Listing 12 checks if the current user's role is 'Admin'. If so, it displays the link to the admin functionality. If not, it doesn't display the link. We'll add additional checks in the .jsp pages which simulate functionality to protect against attempts to directly access the functionality pages. The resulting home page should look like Listing 13:
**Listing 13. Home page**

```html
<%@taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@taglib uri="http://java.sun.com/jsf/core" prefix="f"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Home</title>
</head>
<body>
<f:view>
  <f:loadBundle basename="com.ibm.test.messages" var="msg"/>
  <jsp:useBean id="userBean" scope="session" class="com.ibm.test.UserBean" />
  <h:outputText value="#{msg.welcome} \#{userBean.user.firstName} \#{userBean.user.lastname}" />
  <br /><a href="../private/casual.jsp">Casual Functionality</a>
  <c:if test="${userBean.user.userRole == 'Expert' || userBean.user.userRole == 'Admin'}">
    <br /><a href="../private/expert.jsp">Expert Functionality</a>
  </c:if>
  <c:if test="${userBean.user.userRole == 'Admin'}">
    <br /><a href="../private/admin.jsp">Admin Functionality</a>
  </c:if>
</f:view>
</body>
</html>
```

We'll create three other .jsp pages in the private directory, one for each of the simulated functionalities: casual, expert, and admin. As mentioned above, we'll need to do an additional check in the .jsp pages to protect ourselves against attempts to directly access the .jsp pages. The check can be done like Listing 14:

**Listing 14. Authorization**

```html
<c:if test="${userBean.user.userRole != 'Admin'}">
  <c:redirect url="/public/authError.html"/>
</c:if>
```

In the above code snippet, we check for the expected user role. If the user role is not what was expected, then we redirect the user to an error page. No such check is required for the casual functionality because it's open to all authenticated users.

The casual functionality page should look like Listing 15:
Listing 15. Casual user functionality

```html
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
Casual user functionality is accessible here.
</body>
</html>
```

The expert functionality page should look like Listing 16:

Listing 16. Expert user functionality

```html
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
<jsp:useBean id="userBean" scope="session" class="com.ibm.test.UserBean" />
<c:if test="$\{userBean.user.userRole != 'Expert'}">
  <c:if test="$\{userBean.userRole != 'Admin'}">
    
  </c:if>
</c:if>
Expert user functionality is accessible here.
</body>
</html>
```

The admin functionality should look like Listing 17:

Listing 17. Admin user functionality

```html
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
<jsp:useBean id="userBean" scope="session" class="com.ibm.test.UserBean" />
<c:if test="$\{userBean.user.userRole != 'Admin'}">
  <c:redirect url="/public/authError.html"/>
</c:if>
Admin user functionality is accessible here.
</body>
</html>
```
Your directory structure should now look like the following navigation pane:

**Figure 10. Project structure**

![Project structure diagram]

**Authentication and persistence**

For the purposes of this article, we can put all the back-end code in a single package which we'll call com.ibm.test. You can create it by doing the following:

- Right click **Java Resources: src** from the left navigation pane.
- Select **New** and then select **Package**.
- Enter `com.ibm.test` in the Name field and click **Finish**.

We'll be using a resource bundle, so let's create the resource file first by doing the following:

- Right click on the newly created package.
- Select **New** and then select **Other**.
- Double click **General** in the New pop-up window.
- Select **File** and then click **Next**.
- Enter `messages.properties` into the File Name field and click **Finish**.

The example in this article only uses the resource bundle for the welcome message, and therefore, it only needs to have the following line:
However, the file could include all of the content for the application.

Application authentication is handled by a Java class called UserBean. Create it by doing the following:

- Right click the `com.ibm.test package`
- Select `New` and then select `Class`.
- Enter `UserBean` in the Name field and click `Finish`.

The UserBean class will utilize two Java design patterns: Data Access Object (DAO) and Transfer Object (TO). The DAO will be responsible for accessing the user's data from the LDAP directory. The TO will be responsible for containing the data. The UserBean class is instantiated upon the reference from `home.jsp`:

```jsp
<jsp:useBean id="userBean" scope="session" class="com.ibm.test.UserBean" />
```

The constructor of the `UserBean` class needs to get the user ID of the authenticated user. Remember the authentication mechanism we're using is Tomcat's default form authentication. For the purposes of this article, we're going to let Tomcat handle authentication and use the `UserBean` class to grab and persist the authenticated user's data. We can grab the user ID from the `UserBean` class like this:

```java
ExternalContext context = FacesContext.getCurrentInstance().getExternalContext();
String userID = context.getRemoteUser();
```

We can now use the authenticated user ID to retrieve all the user's data from the LDAP directory. This is done through the `OpenLDAPDAO` class. Create it by doing the following:

- Right click the `com.ibm.test package`
- Select `New` and then select `Class`.
- Enter `OpenLDAPDAO` in the Name field and click `Finish`.

The `OpenLDAPDAO` class will contain the details of the LDAP directory in class variables. Its constructor will create a connection to the LDAP directory. It will also contain a method for retrieving the attributes of a specified user. The `UserBean` class maintains the user data through the `UserTransferObject` class. This class is a container for the user data. A copy of all the classes is included in the attached archive.

Your Java directory structure should look like the structure in the navigation pane in Figure 11.
Figure 11. Project Java structure

So there it is! Start your server and give it a test drive.

An unauthenticated attempt to access the home page will redirect you to the log-in page, shown in Figure 12.

Figure 12. Log-in page

Logging in as a casual user will only list one of the functionality links as in Figure 13.

Figure 13. Casual user

If the user attempts to directly access the .jsp page for an unauthorized functionality, a user error message is displayed as in Figure 14.

Figure 14. Authentication error

A login with admin privileges will load all the functionality links as in Figure 15.
Conclusion

Congratulations! In a short amount of time you have set up a skeleton application that implements authorization. You can now build an application around this that can control access to functionality based on the user roles you have defined in your LDAP directory. You can continue developing your application on Tomcat or WASCE, where you can take advantage of its additional functionality. The application skeleton also implements JSF, which we didn't explore in this article but is a great point of exploration.

Another point of exploration is OpenLDAP. Setting up OpenLDAP is an article in itself. The OpenLDAP Web site is a great resource. Third-party software can be used to facilitate the management of your LDAP server. Jxplorer is a tool we found to be very good. However, you may want to implement an administrative section to your application where administrators can manage the LDAP server through the application. This could easily be built into the application skeleton using the existing patterns along with JSF.
### Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tomcat version of the app</td>
<td>RoleBasedNavTomcat6.war</td>
<td>1.42MB</td>
</tr>
<tr>
<td>WASCE version of the app</td>
<td>RolesBasedNavWASCE.war</td>
<td>373KB</td>
</tr>
</tbody>
</table>
Related topics

- Download Eclipse (Ganymede), the Integrated Development Environment we use in this article.
- Download JSTL 1.2.
- Learn more about the JavaServer Pages Standard Tag Library (version 1.2) we used to help render our UI.
- We use IBM WebSphere Application Server Community Edition as an alternative to Tomcat. Download WASCE.
- We manage our application users through an open source implementation of the Lightweight Directory Access Protocol called OpenLDAP. Download LDAP.
- We administer our LDAP server through an open source LDAP browser called JXplorer. Download JXplorer.
- Learn more about authentication, authorization, and access control.
- Learn about migrating from Tomcat to WASCE.
- Learn about configuring Web application security in WASCE V2.0.
- Find General WASCE resources.
- Access WASCE User guides.