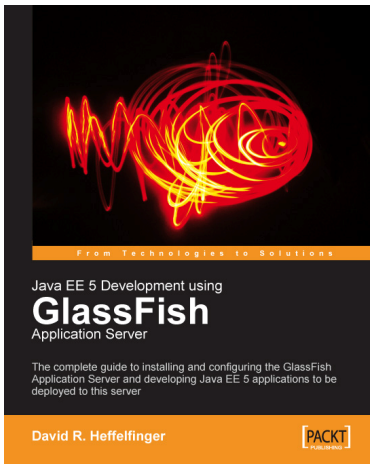
Java EE 5 Development using GlassFish Application Server

The complete guide to installing and configuring the GlassFish Application Server and developing Java EE 5 applications to be deployed to this server

David R. Heffelfinger



Chapter No. 8

"Security"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO. 8 "Security"

A synopsis of the book's content

Information on where to buy this book

About the Author

David Heffelfinger has been developing software professionally since 1995; he has been using Java as his primary programming language since 1996. He has worked on many large-scale projects for several clients including Freddie Mac, Fannie Mae, and the US Department of Defense. He has a Masters degree in Software Engineering from Southern Methodist University. David is editor in chief of Ensode.net (`http://www.ensode.net`), a website about Java, Linux, and other technology topics.

First and foremost, I would like to thank my family for putting up with me spending several hours a day working on this book; without your support, I wouldn't have been able to accomplish this.

I would also like to thank the Packt Publishing staff for their help and support in getting this book published. I am especially grateful to Priyanka Baruah, who first contacted me regarding this book, Patricia Weir for her patience regarding the several changes to the book's outline, Sagara Naik for keeping track of the schedule. I would also like to thank the technical reviewers, Kim Lewis and Meenakshi Verma for providing excellent suggestions. Last but not least, I would also like to thank Douglas Paterson, who gave me the opportunity to get my first book published (and who wished to work on a second book with me) for supporting my decision to work on this book.

Java EE 5 Development using GlassFish Application Server

The complete guide to installing and configuring the GlassFish Application Server and developing Java EE 5 applications to be deployed to this server

Project GlassFish was formally announced at the 2005 JavaOne conference. Version one of the GlassFish application server was released to the public approximately a year later, at the 2006 JavaOne conference. GlassFish version one became the reference implementation for the Java EE 5 specification, and as such, was the first available application server compliant with this specification.

While releasing the first available Java EE 5 application server was a tremendous accomplishment, the first version of GlassFish lacked some enterprise features such as clustering and High Availability. GlassFish version 2, released in September 2007, added these and other enterprise features, in addition to other features such as an enhanced web based administration console.

This book will guide you through the development and deployment of Java EE 5-compliant application on GlassFish version 2. It also covers application development using frameworks that build on top of the Java EE 5 specification, including Facelets, Ajax4jsf, and Seam.

What This Book Covers

Chapter 1 provides an overview of Glassfish, including how to install it, configure it, and verify the installation.

Chapter 2 covers how to develop server-side web applications using the Servlet API.

Chapter 3 explains how to develop web applications using JavaServer Pages (JSPs), including how to develop and use JSP custom tags.

Chapter 4 discusses how to develop Java EE applications that interact with a relational database system through the Java Persistence API (JPA) and through the Java Database Connectivity API (JDBC).

Chapter 5 explains how to use the JSP Standard Tag Library (JSTL) when developing JavaServer Pages.

Chapter 6 covers how to develop applications using the JavaServer Faces (JSF) component framework to build web applications.

Chapter 7 explains how to develop messaging applications though the Java Messaging Service (JMS) API.

Chapter 8 covers securing J2EE applications through the Java Authentication and Authorization Service (JAAS).

Chapter 9 discusses how to develop Enterprise Java Beans that adhere to the EJB 3 specification.

Chapter 10 explains how to develop and deploy web services that conform to the JAX-WS 2.1 specification.

Chapter 11 covers frameworks that build on top of the Java EE 5 specification, including Seam, Facelets, and Ajax4Jsf.

Appendix A covers sending email from Java EE Applications.

Appendix B covers IDE integration.

8
Security

In this chapter, we will cover how to secure Java EE applications by taking advantage of GlassFish's built-in security features. Java EE security relies on the Java Authentication and Authorization Service (JAAS) API. As we shall see, securing Java EE applications requires very little coding; for the most part, securing an application is achieved by setting up users and security groups in a security realm in the application server, then configuring our applications to rely on a specific security realm for authentication and authorization.

Some of the topics we will cover include:

- The Admin realm
- The File realm
- The Certificate realm
    - Creating self-signed security certificates
- The JDBC realm
- Custom Realms

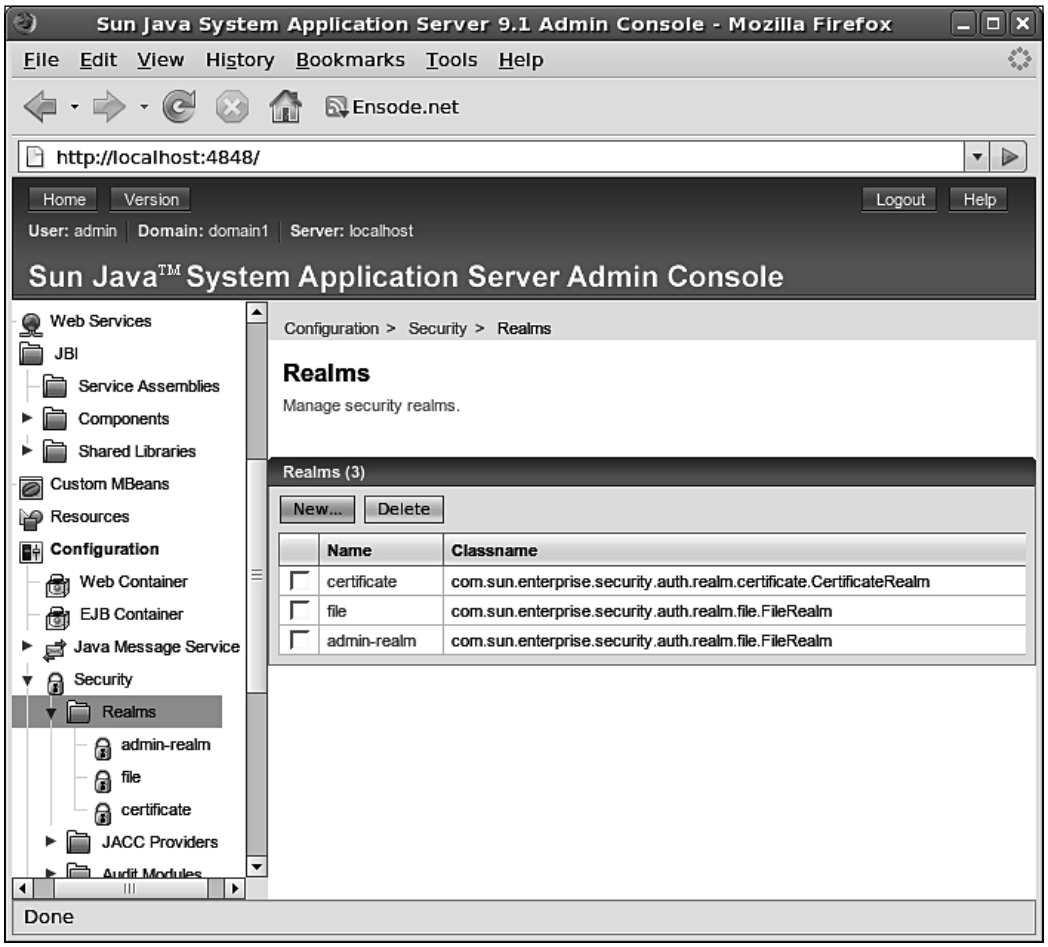
Security Realms

Security realms are, in essence, collections of users and related security groups. Users are application users. A user can belong to one or more security group; the groups that the user belongs to define what actions the system will allow the user to perform. For example, an application can have regular users who can only use the basic application functionality, and it can have administrators who, in addition to being able to use basic application functionality, can add additional users to the system.

Security realms store user information (user name, password, and security groups); applications don't need to implement this functionality, they can simply be configured to obtain this information from a security realm. A security realm can be used by more than one application.

Predefined Security Realms

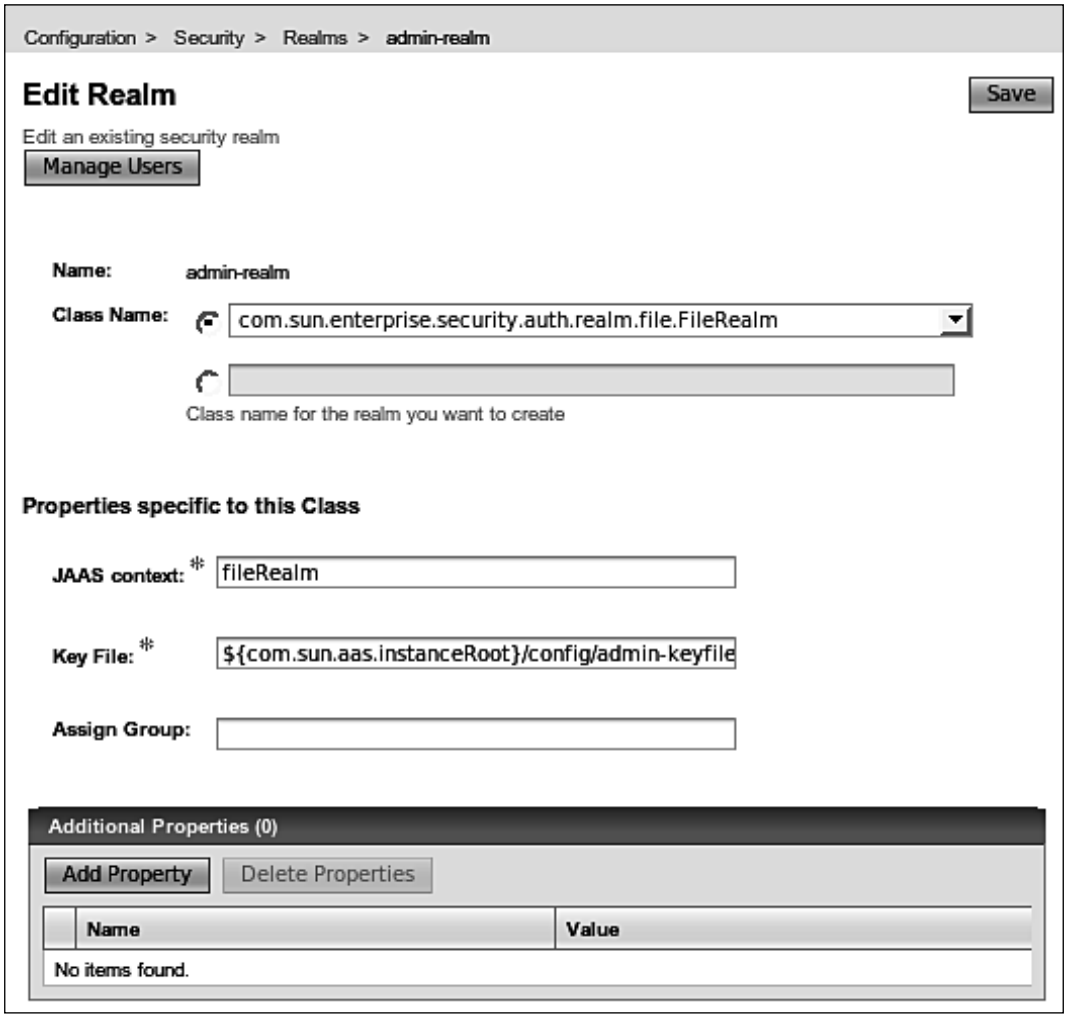
GlassFish comes preconfigured with three predefined security realms: **admin-realm**, the **file realm**, and the **certificate realm**. **admin-realm** is used to manage user's access to the GlassFish web console and shouldn't be used for other applications. The file realm stores user information in a file. The certificate realm looks for a client-side certificate to authenticate the user.



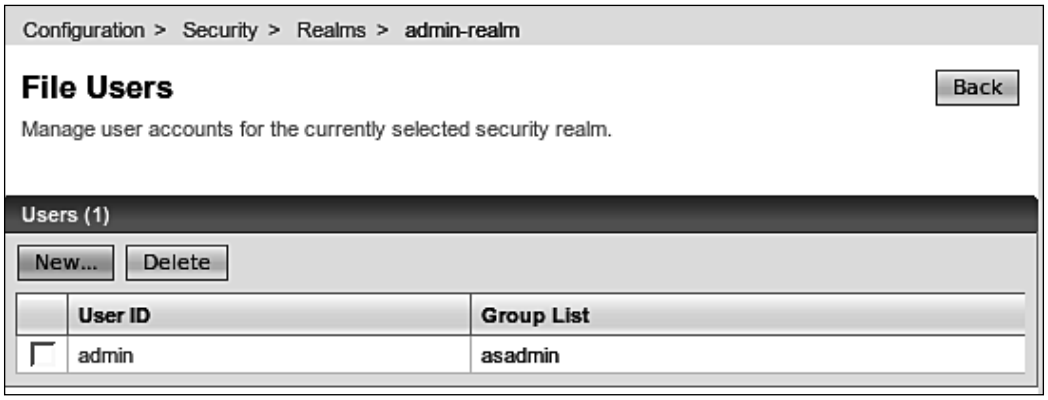
In addition to the predefined security realms, we can add additional realms with very little effort. We will cover how to do this later in this chapter, but first let's discuss GlassFish's predefined security realms.

admin-realm

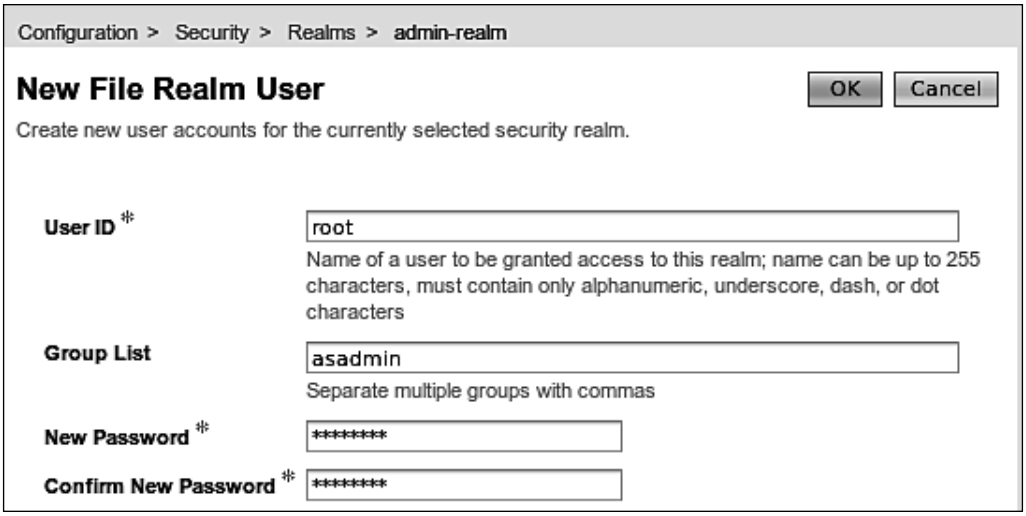
To illustrate how to add users to a realm, let's add a new user to admin-realm. This will allow this additional user to log in to the GlassFish web console. In order to add a user to admin-realm, log in to the GlassFish web console, expand the **Configuation** node at the left-hand side, then expand the **Security** node, then the **Realms** node, and click on **admin-realm**. The main area of the page should look like the following screenshot:



To add a user to the realm, click on the button labeled **Manage Users** at the top left. The main area of the page should now look like this:

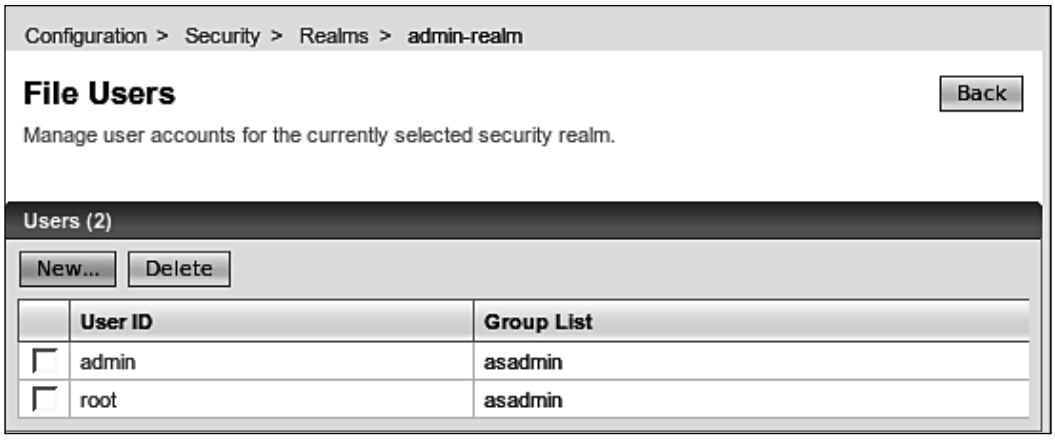


To add a new user to the realm, simply click on the **New...** button at the top left of the screen. Then enter the new user information.



In the above screenshot, we added a new user named "root", added this user to the "asadmin" group, and entered this user's password.

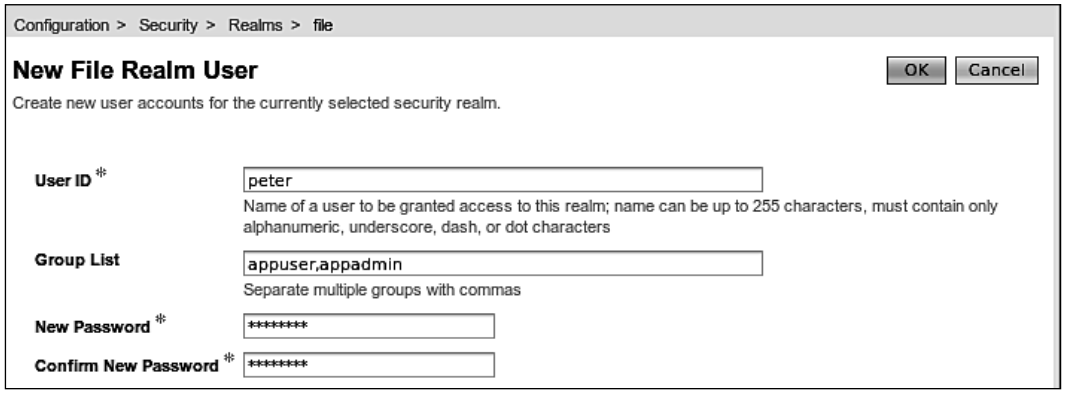
The GlassFish web console will only allow users in the "asadmin" group to log in. Failing to add our user to this security group would prevent him/her from logging in to the console.



We have successfully added a new user for the GlassFish web console. We can test this new account by logging into the console with this new user's credentials.

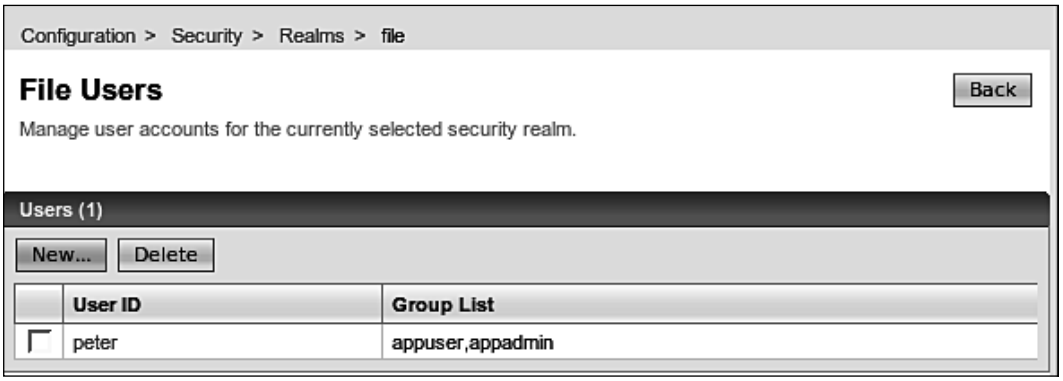
The file Realm

The second predefined realm in GlassFish is the file realm. This realm stores user information encrypted in a text file. Adding users to this realm is very similar to adding users to admin-realm. We can add a user by expanding the **Configuration** node, then expanding the **Security** node, then the **Realms** node, then clicking on **file**, then clicking on the **Manage Users** button and clicking on the **New...** button.

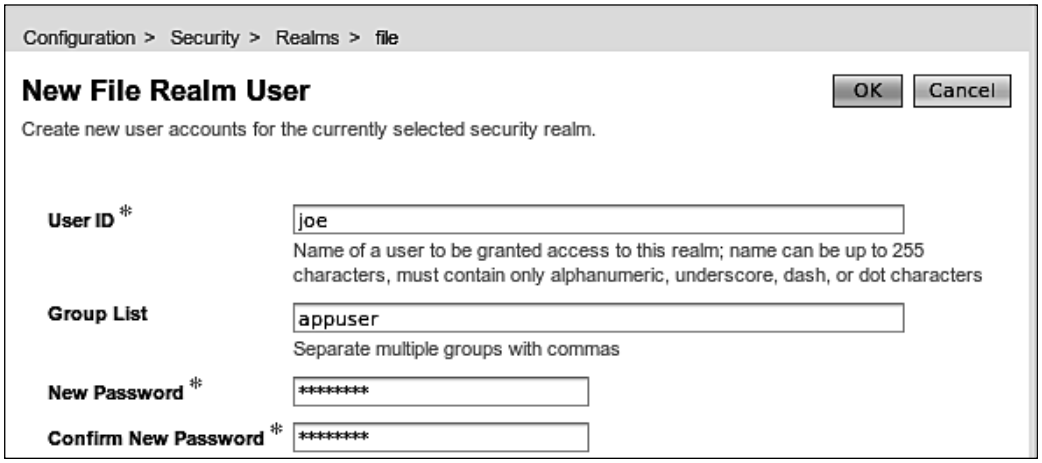


As this realm is meant for us to use for our applications, we can come up with our own groups. In this example, we added a user with a User ID of "peter" to the groups "appuser" and "appadmin".

Clicking the OK button should save the new user and take us to the user list for this realm.



Clicking the **New...** button allows us to add additional users to the realm. Let's add an additional user called "joe" and belonging only to the "appuser" group.



As we have seen in this section, adding users to the file realm is very simple. We will now illustrate how to authenticate and authorize users via the file realm.

File Realm Basic Authentication

In the previous section, we covered how to add users to the file realm and how to assign roles to these users. In this section, we will illustrate how to secure a web application so that only properly authenticated and authorized users can access it. This web application will use the file realm for user access control.

The application will consist of a few very simple JSPs. All authentication logic is taken care of by the application server, therefore the only place we need to make modifications in order to secure the application is in its deployment descriptors, `web.xml` and `sun-web.xml`. We will first discuss `web.xml`, which is shown next.

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="2.5"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.
sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Admin Pages</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>All Pages</web-resource-name>
      <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>user</role-name>
    </auth-constraint>
  </security-constraint>
  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>file</realm-name>
</web-app>
```

The `<security-constraint>` element defines who can access pages matching a certain URL pattern. The URL pattern of the pages is defined inside the `<url-pattern>` element, which, as shown in the example, must be nested inside a `<web-resource-collection>` element. Roles allowed to access the pages are defined in the `<role-name>` element, which must be nested inside an `<auth-constraint>` element.

In the above example, we define two sets of pages to be protected. The first set of pages is any page whose URL starts with `/admin`. These pages can only be accessed by users with the role of admin. The second set of pages is all pages, defined by the URL pattern of `/*`. Only users with the role of user can access these pages. It is

worth noting that the first set of pages is a subset of the second set, that is, any page whose URL matches `/admin/*` also matches `/*`; in cases like this the most specific case "wins". In this particular case, users with a role of user (and without the role of admin) will not be able to access any page whose URL starts with `/admin`.

The next element we need to add to `web.xml` in order to protect our pages is the `<login-config>` element. This element must contain an `<auth-method>` element that defines the authorization method for the application. Valid values for this element include `BASIC`, `DIGEST`, `FORM`, and `CLIENT-CERT`.

`BASIC` indicates that basic authentication will be used. This type of authentication will result in a browser-generated popup, prompting the user for a user name and password, being displayed the first time a user tries to access a protected page. Unless using the HTTPS protocol, when using basic authentication, the user's credentials are Base64 encoded, not encrypted. It would be fairly easy for an attacker to decode these credentials; therefore using basic authentication is not recommended.

`DIGEST` is similar to basic authentication except it uses an MD5 DIGEST to encrypt the user credentials instead of sending them Base64 encoded.

`FORM` uses a custom HTML or JSP page containing an HTML form with user name and password fields. The values in the form are then checked against the security realm for user authentication and authorization. Unless using HTTPS, user credentials are sent in clear text when using form-based authentication, therefore using HTTPS is recommended because it encrypts the data. We will cover setting up GlassFish to use HTTPS, later in this chapter.

`CLIENT-CERT` uses client-side certificates to authenticate and authorize the user.

The `<realm-name>` element of `<login-config>` indicate which security realm to use to authenticate and authorize the user. In this particular example, we are using the file realm.

All of the `web.xml` elements we have discussed in this section can be used with any security realm; they are not tied to the file realm. The only thing that ties our application to the file realm is the value of the `<realm-name>` element. Something else to keep in mind is that not all authentication methods are supported by all realms. The file realm supports only basic and form-based authentication.

Before we can successfully authenticate our users, we need to link the user roles defined in `web.xml` with the groups defined in the realm. We accomplish this in the `sun-web.xml` deployment descriptor.

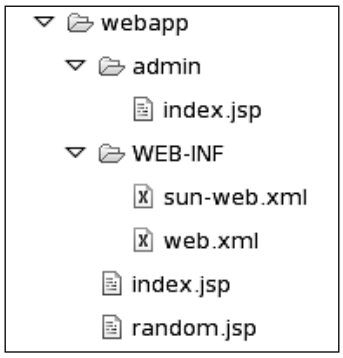
```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD
Application Server 9.0 Servlet 2.5//EN" "http://www.sun.com/software/
```

```
appserver/dtds/sun-web-app_2_5-0.dtd">
<sun-web-app>
  <security-role-mapping>
    <role-name>admin</role-name>
    <group-name>appadmin</group-name>
  </security-role-mapping>
  <security-role-mapping>
    <role-name>user</role-name>
    <group-name>appuser</group-name>
  </security-role-mapping>
</sun-web-app>
```

As can be seen in the example, the `sun-web.xml` deployment descriptor can have one or more `<security-role-mapping>` elements; one of these elements for each role defined in `web.xml` is needed. The `<role-name>` subelement indicates the role to map. Its value must match the value of the corresponding `<role-name>` element in `web.xml`. The `<group-name>` subelement must match the value of a security group in the realm used to authenticate users in the application.

In this example, the first `<security-role-mapping>` element maps the "admin" role defined in the application's `web.xml` deployment descriptor to the "appadmin" group we created when adding users to the file realm earlier in the chapter. The second `<security-role-mapping>` maps the "user" role in `web.xml` to the "appuser" group in the file realm.

As we mentioned earlier, there is nothing we need to do in our code in order to authenticate and authorize users. All we need to do is modify the application's deployment descriptors as described in this section. As our application is nothing but a few simple JSPs, we will not show the source code for them. The structure of our application is shown in the following screenshot:



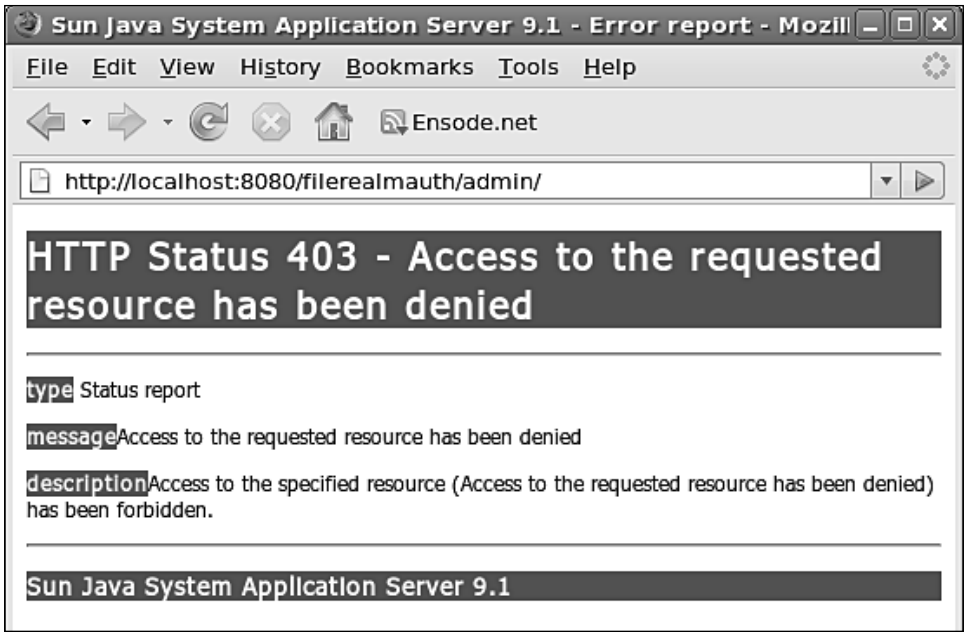
Based on the way we set up our application in the deployment descriptors, users with a role of "user" will be able to access the two JSPs at the root of the application (`index.jsp` and `random.jsp`). Only users with the role of "admin" will be able to access any pages under the "admin" folder, which in this particular case is a single JSP named `index.jsp`.

After packaging and deploying our application and pointing the browser to the URL of any of its pages, we should see a popup asking for a user name and a password.

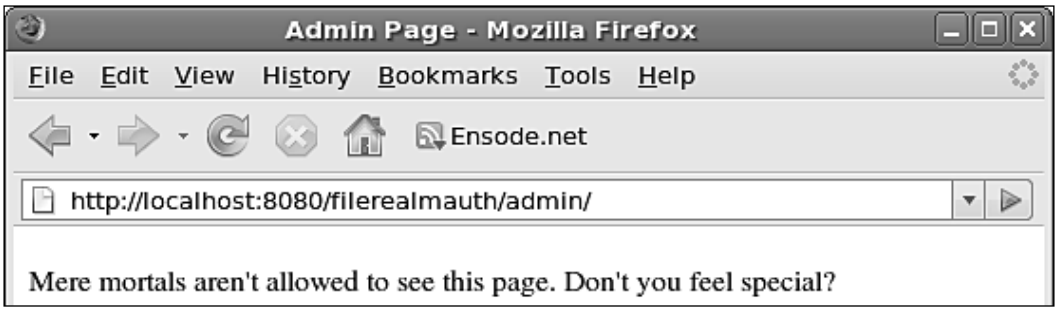
After entering the correct user name and password, we are directed to the page we were attempting to see.

At this point, the user can navigate to any page he or she is allowed to access in the application, either by following links or by typing the URL in the browser, without having to re-enter his/her user name and password.

Notice that we logged in as user joe; this user belongs only to the user role, therefore he does not have access to any page with a URL that starts with `/admin`. If joe tries to access one of these pages, he will see the following error message in the browser.



Only users belonging to the admin role can see pages that match the above URL. When we were adding users to the file realm, we added a user named peter that had this role. If we log in as peter, we will be able to see the requested page. For basic authentication, the only way possible to log out of the application is to close the browser, therefore to log in as peter we need to close and reopen the browser.



As we mentioned before, one disadvantage of the basic authentication method we used in this example is that login information is not encrypted. One way to get around this is to use the HTTPS (HTTP over SSL) protocol; when using this protocol all information between the browser and the server is encrypted.

The easiest way to use HTTPS is by modifying the application `web.xml` deployment descriptor.

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="2.5"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.
sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Admin Pages</web-resource-name>
      <url-pattern>/admin/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>admin</role-name>
    </auth-constraint>
    <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
  </security-constraint>
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>AllPages</web-resource-name>
      <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>user</role-name>
    </auth-constraint>
    <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
  </security-constraint>
  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>file</realm-name>
  </login-config>
</web-app>
```

As we can see, all we need to do to have the application be accessed only through HTTPS is to add a `<user-data-constraint>` element containing a nested `<transport-guarantee>` element to each set of pages we want to encrypt traffic. Sets of pages to be protected are declared in the `<security-constraint>` elements in the `web.xml` deployment descriptor.

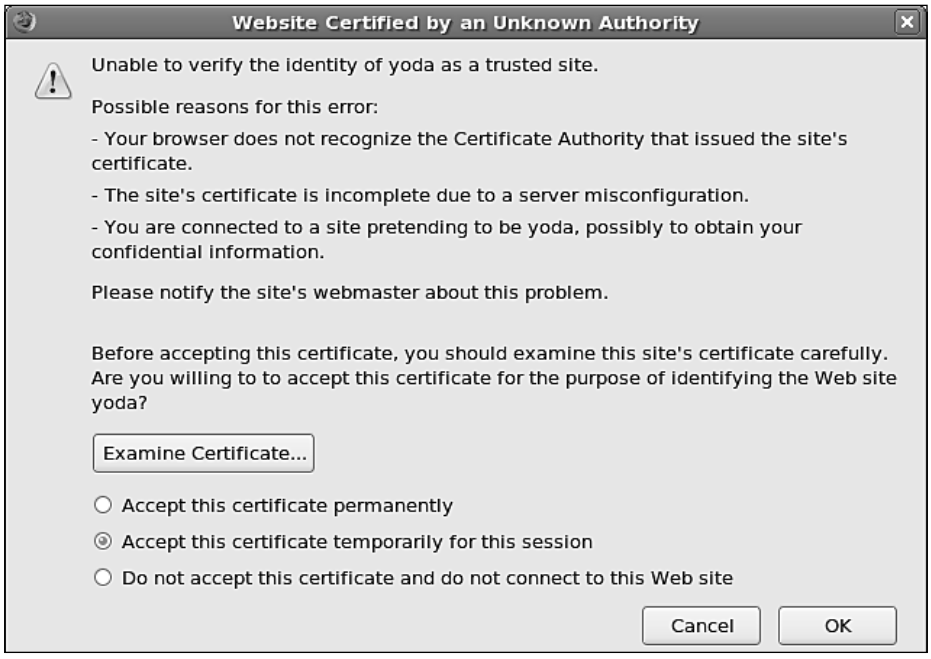
Now, when we access the application through the (unsecure) HTTP port (by default this is 8080), the request is automatically forwarded to the (secure) HTTPS port (default of 8181).

In this example, we set the value of the `<transport-guarantee>` to `CONFIDENTIAL`. This has the effect of encrypting all the data between the browser and the server, also, if the request is made through the unsecured HTTP port, it is automatically forwarded to the secured HTTPS port.

Another valid value for the `<transport-guarantee>` element is `INTEGRAL`. When using this value, the integrity of the data between the browser and the server is guaranteed; in other words, the data cannot be changed in transit. When using this value, requests made over HTTP are not automatically forwarded to HTTPS; if a user attempts to access a secure page via HTTP when this value is used, the browser will deny the request and return a 403 (Access Denied) error.

The third and last valid value for the `<transport-guarantee>` is `NONE`. When using this value, no guarantees are made about the integrity or confidentiality of the data. `NONE` is the default value used when the `<transport-guarantee>` element is not present in the application's `web.xml` deployment descriptor.

After making the above modifications to the `web.xml` deployment descriptor, redeploying the application and pointing the browser to any of the pages in the application, we should see the following.



The reason we see this warning window is that, in order for a server to use the HTTPS protocol, it must have an SSL certificate. Typically, SSL certificates are issued by certificate authorities such as Verisign or Thawte. These certificate authorities digitally sign the certificate; by doing this they certify that the server belongs to the entity to which it claims to belong.

A digital certificate from one of these certificate authorities typically costs around $400 USD, and expires after a year. As the cost of these certificates may be prohibitive for development or testing purposes, GlassFish comes preconfigured with a self-signed SSL certificate. As this certificate has not being signed by a certificate authority, the browser pops up the above warning window when we try to access a secured page via HTTPS. We can simply click OK to accept the certificate.

Once we accept the certificate, we are prompted for a user name and password; after entering the appropriate credentials, we are allowed access to the requested page.

Notice the URL in the above screenshot; the protocol is set to HTTPS, and the port is 8181. The URL we pointed the browser to was `http://localhost:8080/filerealmauthhttps/random.jsp`; because of the modifications we made to the application's `web.xml` deployment descriptor, the request was automatically forwarded to this URL. Of course, users may directly type the secure URL and it will work without a problem.

Any data transferred over HTTPS is encrypted, including the user name and password entered at the pop-up window generated by the browser. Using HTTPS allows us to safely use basic authentication. However, basic authentication has another disadvantage, which is that the only way that a user can log out from the application is to close the browser. If we need to allow users to log out of the application without closing the browser, we need to use form-based authentication.

When using form-based authentication, we need to make some modifications to the application's `web.xml` deployment descriptor.

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="2.5"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.
sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Admin Pages</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>AllPages</web-resource-name>
      <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>user</role-name>
    </auth-constraint>
  </security-constraint>
  <login-config>
    <auth-method>FORM</auth-method>
    <realm-name>file</realm-name>
    <form-login-config>
      <form-login-page>/login.jsp</form-login-page>
      <form-error-page>/loginerror.jsp</form-error-page>
    </form-login-config>
  </login-config>
  <servlet>
    <servlet-name>LogoutServlet</servlet-name>
    <servlet-class>
      net.ensode.glassfishbook.LogoutServlet
    </servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>LogoutServlet</servlet-name>
    <url-pattern>/logout</url-pattern>
  </servlet-mapping>
</web-app>
```

When using form-based authentication, we simply use `FORM` as the value of the `<auth-method>` element in `web.xml`. When using this authentication method, we need to provide a login page and a login error page. We indicate the URLs for the login and login error pages as the values of the `<form-login-page>` and `<form-error-page>` elements, respectively. As can be seen in the example, these elements must be nested inside the `<form-login-config>` element.

The markup for the login page for our application is shown next.

```
<%@ 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>Login</title>
</head>
<body>
<p>Please enter your username and password to access the application</
p>
<form method="POST" action="j_security_check">
<table cellpadding="0" cellspacing="0" border="0">
  <tr>
    <td align="right">Username: </td>
    <td>
      <input type="text" name="j_username">
    </td>
  </tr>
  <tr>
    <td align="right">Password: </td>
    <td>
      <input type="password" name="j_password">
    </td>
  </tr>
  <tr>
    <td></td>
    <td><input type="submit" value="Login"></td>
  </tr>
</table>
</form>
</body>
</html>
```

The login page for an application using form-based authentication must contain a form whose method is `"POST"` and whose action is `"j_security_check"`. We don't need to implement a servlet or anything else to process this form. The code to process it is supplied by the application server.

The form in the login page must contain a text field named `j_username`; this text field is meant to hold the user's user name. Additionally, the form must contain a password field named `j_password`, meant for the user's password. Of course, the form must contain a submit button to submit the data to the server.

The only requirement for a login page is for it to have a form whose attributes match those in the preceding example, and the `j_username` and `j_password` input fields as described in the above paragraph.

There are no special requirements for the error page. Of course, it should show an error message telling the user that login was unsuccessful; however, it can contain anything we wish. The error page for our application simply tells the user that there was an error logging in, and links back to the login page to give the user a chance to try again.

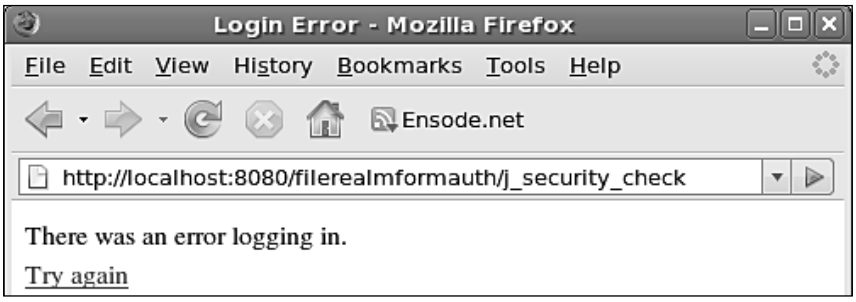
In addition to a login page and a login error page, we added a servlet to our application. This servlet allows us to implement logout functionality, something that wasn't possible when we were using basic authentication.

```
package net.ensode.glassfishbook;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class LogoutServlet extends HttpServlet
{
  @Override
  protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException
  {
    request.getSession().invalidate();
    response.sendRedirect("index.jsp");
  }
}
```

As you can see, all we need to do to log out the user is invalidate the session. In our servlet, we redirect the response to the `index.jsp` page; as the session is invalid at this point, the security mechanism will "kick in" and automatically direct the user to the login page.

We are now ready to test form-based authentication; after building our application, deploying it, and pointing the browser to any of its pages, we should see our login page rendered in the browser.

If we submit invalid credentials, we are automatically forwarded to the login error page.



We can click on the **Try again** link to try again. After entering valid credentials, we are allowed into the application.

As you can see, we added a logout link to the page; this page directs the user to the logout servlet, which as we mentioned before simply invalidates the session. From the user's point of view, this link will simply log them out and direct them to the login screen.

Where to buy this book

You can buy Java EE 5 Development using GlassFish Application Server from the Packt Publishing website:

`http://www.packtpub.com/Java-EE-5-GlassFish-Application-Servers/book`

Free shipping to the US, UK, Europe, Australia, New Zealand and India.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:

www.packtpub.com/Java-EE-5-GlassFish-Application-Servers/book