SECURITY

Topics in this Chapter

- Servlet Authentication
  - Principals and Roles
  - Declarative Authentication
  - Portability
  - Types of Authentication
- Basic Authentication
- Digest Authentication
- Form-Based Authentication
- SSL and Client Certificate Authentication
- Web Application Security Elements
- Customizing Authentication
  - Resin
  - Tomcat 4.0
- Programmatic Authentication

Chapter 9

Computer security used to be the domain of hackers and their antagonists, but with the advent of the World Wide Web, it's become an issue for the rank and file setting up shop on the net. Because of this growing awareness, software developers today are far more likely to deal with security than were their counterparts of the late 20th century.

Many books have been written about the wide ranging topic of computer security, including Java security, and this chapter is a substitute for none of them. This discussion is restricted to protecting web application resources with the authentication mechanisms described in the servlet specification.[1]

Servlet Authentication

Servlet authentication looks simple:

1. A user tries to access a protected resource, such as a JSP page.
2. If the user has been authenticated, the servlet container makes the resource available; otherwise, the user is asked for a username and password.
3. If the name and password cannot be authenticated, an error is displayed and the user is given the opportunity to enter a new username and password.

1. This chapter is based upon the 2.2 Servlet specification; for specification links, see `http://java.sun.com/products/servlet/download.html`.

The steps outlined above are simple, but vague. It's not apparent who asks for a username and password, who does the authentication, how it's performed, or even how the user is asked for a username and password. Those steps are unspecified because the servlet specification leaves them up to applications and servlet containers. This vagueness in the servlet specification has an effect on portability; see "Portability" on page 254 for more information.

Principals and Roles

In security-speak, the user in the steps listed on page 251 is a *principal*. Principals are named entities that can represent anything; most often, they represent individuals or corporations.

Principals can fill one or more roles; for example, a customer could also be an employee. Security constraints in `WEB-INF/web.xml` associate *roles* with protected *resources*, like this:

```
<web-app>
...
   <security-constraint>
      <!-- web resources that are protected -->
      <web-resource-collection>
         <web-resource-name>Protected Resource</web-resource-name>
         <url-pattern>/page_1.jsp</url-pattern>
      </web-resource-collection>

      <auth-constraint>
         <!-- role-name indicates roles that are allowed
              to access the web resources specified above -->
         <role-name>customer</role-name>
      </auth-constraint>
   </security-constraint>
...
   <security-constraint>
      <!-- web resources that are protected -->
      <web-resource-collection>
        <web-resource-name>Protected Resource2</web-resource-name>
         <url-pattern>/page_2.jsp</url-pattern>
      </web-resource-collection>

      <auth-constraint>
         <!-- role-name indicates roles that are allowed
              to access the web resources specified above -->
         <role-name>employee</role-name>
      </auth-constraint>
   </security-constraint>
<web-app>
```

Two security constraints are specified above that restrict access to `/page_1.jsp` and `/page_2.jsp` to principals that are in roles `customer` or `employee`, respectively.

Security constraints, like those listed above, associate resources with roles. It's up to servlet containers or applications to associate roles with principals; for example, with Tomcat, you edit a `tomcat-users.xml` file that has entries like this:

```
<tomcat-users>
  ...
  <user name="rwhite" password="tomcat" roles="customer", "other"/>
  ...
</tomcat-users>
```

Here, `rwhite` has a password of `tomcat` and can fill roles `customer` or `other`; thus, `rwhite` can access `/page_1.jsp`, but not `/page_2.jsp` according to the security constraints listed above.

Other servlet containers provide different mechanisms for associating principals with roles; for example, "Resin" on page 264 illustrates how it's done with Resin for basic authentication.

Table 9-1 lists `HttpServletRequest` methods that allow you to retrieve information about principals and roles.

Table 9-1 `HttpServletRequest` Methods for Principals and Roles

Method	Description
`Principal getUserPrincipal()`	Returns a reference to a `java.security.Principal`
`boolean isUserInRole(String)`	Determines whether a user is in a role, specified by the string argument
`String getRemoteUser()`	Returns the username that was used for login

The servlet API does not provide corresponding setter methods for the getter methods listed in Table 9-1; therefore, principals and roles can only be set by servlet containers, meaning that applications cannot set them. This can be a consideration if you implement programmatic authentication—see "Programmatic Authentication" on page 271 for more information.

Table 9-2 lists other `ServletRequest` methods that provide security information.

Table 9-2 Other `ServletRequest` Security Methods[1]

Method	Description
`String getAuthType()`	Returns the authentication type: `BASIC`, `SSL`, or `null`
`boolean isSecure()`	Returns `true` if the connection is `HTTPS`
`String getScheme()`	Scheme represents transport mechanism: `http`, `https`...

1. `getAuthType()` is from `HttpServletRequest`.

Like the methods listed in Table 9-1 on page 253, the servlet API does not provide corresponding setter methods for those methods listed in Table 9-2. This means that the authentication type and transport scheme can only be set by servlet containers.

Declarative Authentication

Declarative authentication requires no programming because authentication is *declared* with XML tags in a deployment descriptor and implemented by the servlet container. Declarative authentication is attractive because it's easy, but it's not as flexible as other approaches that require you to write code.

At one end of the spectrum is declarative authentication, with 100% servlet container implemented and 0% application code; at the other end is programmatic authentication, with 0% servlet container and 100% application code.

Most servlet containers provide access to the middle of that spectrum by providing hooks so that you can replace their default authentication mechanism.

"Basic Authentication" on page 256 provides an example of declarative authentication, "Customizing Authentication" on page 263 illustrates customizing authentication, and programmatic authentication is discussed in "Programmatic Authentication" on page 271.

Portability

The servlet specification leaves enough security details unspecified that servlet containers must fill in the gaps with nonportable functionality. For example, the servlet specification does not specify a default authentication mechanism, so

servlet containers implement their own; for example, Tomcat uses an XML file to specify usernames and passwords, whereas Resin requires you to implement an authenticator.

Because of nonportable security aspects of servlet containers and depending upon your choice for authentication, you may need to write some nonportable code, such as a Resin authenticator or a Tomcat realm, both of which are discussed in "Customizing Authentication" on page 263.

On the other hand, you can use declarative authentication to minimize any code you have to write.

Types of Authentication

A servlet-based web application can choose from the following types of authentication, from least secure to most:

- Basic authentication
- Form-based authentication
- Digest authentication
- SSL and client certificate authentication

All of the authentication mechanisms listed above are discussed in this chapter. Basic and digest authentication are discussed in much detail in RFC2617, which can be found at `ftp://ftp.isi.edu/in-notes/rfc2617.txt`.

You select one of the authentication mechanisms listed above in `/WEBINF/web.xml`, like this:

```
<web-app>
...
   <login-config>
      <auth-method>BASIC</auth-method>
      <realm-name>Basic Authentication Example</realm-name>
   </login-config>
...
</web-app>
```

Although basic and form-based authentication are not secure, you can use them in combination with SSL for secure transport.

You can find out the authentication method for a request with `HttpServletRequest.getAuthType`—see Table 9-2 on page 254.

Basic Authentication

Basic authentication is defined by the HTTP/1.1 specification. When a client attempts to access a protected resource, the server prompts for a username and password. If the server can authenticate that username and password, access is granted to the resource; otherwise, the process repeats. The server retries a server-specific number of times, three being typical.

The most notable aspect of basic authentication is its total lack of security. Passwords are transmitted with base64 encoding, which provides no encryption, thus making the passwords vulnerable.

Figure 9-1 illustrates basic authentication with Tomcat 4.0.

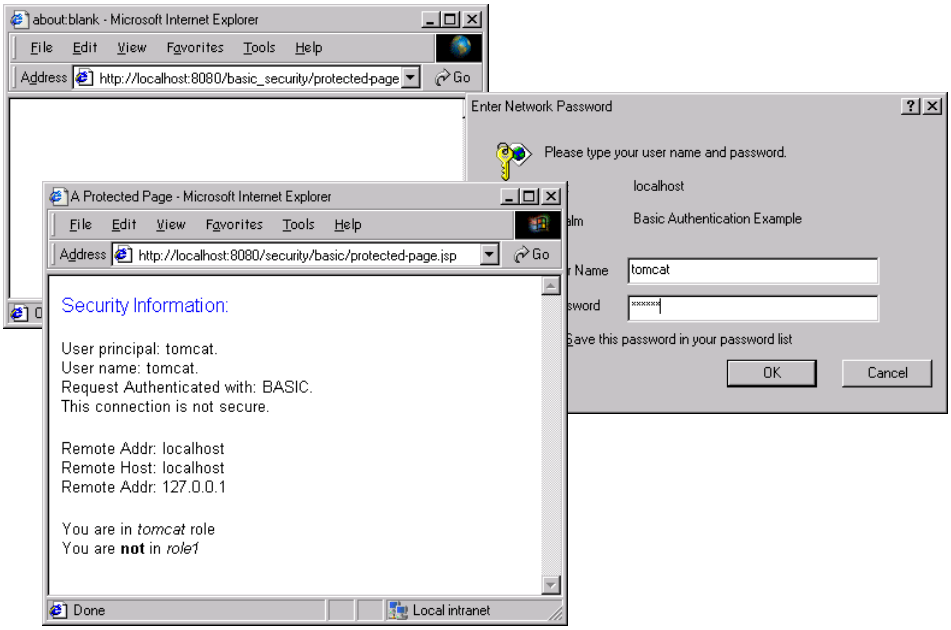

Figure 9-1 Basic Authentication with Tomcat 4.0

From top to bottom in Figure 9-1: An attempt is made to access a protected JSP page—`/protected-page.jsp`—and the user is presented with a dialog. After the dialog is filled out and the username and password are authenticated, the JSP page is displayed. That JSP page is listed in Example 9-1.a.

Example 9-1.a /protected-page.jsp

```
<html><head><title>A Protected Page</title></head>
<body>

<%@ include file='show-security.jsp' %></p>
<p>
<% if(request.isUserInRole("tomcat")) { %>
   You are in <i>tomcat</i> role<br/>
<% } else {%>
   You are <b>not</b> in <i>tomcat</i><br/>
<% } %>

<% if(request.isUserInRole("role1")) { %>
   You are in <i>role1</i><br/>
<% } else {%>
   You are <b>not</b> in <i>role1</i><br/>
<% } %>
</p>

</body>
</html>
```

The JSP page listed in Example 9-1.a prints security information and the principal's role—`tomcat` or `role1`. Security information is printed by a JSP page that's listed in Example 9-1.b.

Example 9-1.b /show-security.jsp

```
<font size='4' color='blue'>
     Security Information:
</font><br>
<p>
User principal: <%= request.getUserPrincipal().getName() %>.<br/>
User name: <%= request.getRemoteUser() %>.<br/>
Request Authenticated with: <%= request.getAuthType() %>.<br/>

<% if(request.isSecure()) { %>
     This connection is secure.<br/>
<% } else { %>
     This connection is not secure.<br/>
<% } %>
</p>
Remote Addr: <%= request.getServerName() %><br/>
Remote Host: <%= request.getRemoteHost() %><br/>
Remote Addr: <%= request.getRemoteAddr() %>
```

The JSP page listed in Example 9-1.b can be handy for debugging authentication.

The protected JSP page listed in Example 9-1.b is specified as a protected resource in the application's deployment descriptor, which is listed in Example 9-1.c.

Example 9-1.c /WEB-INF/web.xml

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

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
   <security-constraint>
      <!-- web resources that are protected -->
      <web-resource-collection>
         <web-resource-name>A Protected Page</web-resource-name>
         <url-pattern>/protected-page.jsp</url-pattern>
      </web-resource-collection>

      <auth-constraint>
         <!-- role-name indicates roles that are allowed
              to access the web resource specified above -->
         <role-name>tomcat</role-name>
         <role-name>role1</role-name>
      </auth-constraint>
   </security-constraint>

   <login-config>
      <auth-method>BASIC</auth-method>
      <realm-name>Basic Authentication Example</realm-name>
   </login-config>
</web-app>
```

The deployment descriptor listed above restricts access to `/protectedpage.jsp` to principals in either `tomcat` or `role1` roles, and `BASIC` is specified as the authentication method.

With Tomcat, usernames and passwords are associated with roles in `$TOMCAT_HOME/conf/tomcat-users.xml`, which is listed in Example 9-1.d.

Example 9-1.d $TOMCAT_HOME/conf/tomcat-users.xml

```
<tomcat-users>
  <user name="tomcat" password="tomcat" roles="tomcat" />
  <user name="role1"  password="tomcat" roles="role1"  />
  <user name="both"   password="tomcat" roles="tomcat,role1" />
</tomcat-users>
```

The configuration file listed in Example 9-1.d binds the username `tomcat` and password `tomcat`, which were used in the application shown in Figure 9-1 on page 256, to the `tomcat` role. That's why the application shows that the principal is in `tomcat` role, but not in `role1`.

The entry for username `both` in Example 9-1.d illustrates how you can associate a single principal with multiple roles using Tomcat. In Figure 9-1 on page 256, if we had logged in as `both`, we would be in both `tomcat` and `role1` roles.

Digest Authentication

Digest authentication is just like basic authentication, except digest authentication uses encryption to protect passwords. In fact, digest authentication transmits a password's hash value, not the password itself.[2]

Figure 9-2 illustrates digest authentication with Tomcat. Notice the differences between the dialog in Figure 9-2, which declares this web site to be secure, and the dialog in Figure 9-1 on page 256, which does not.

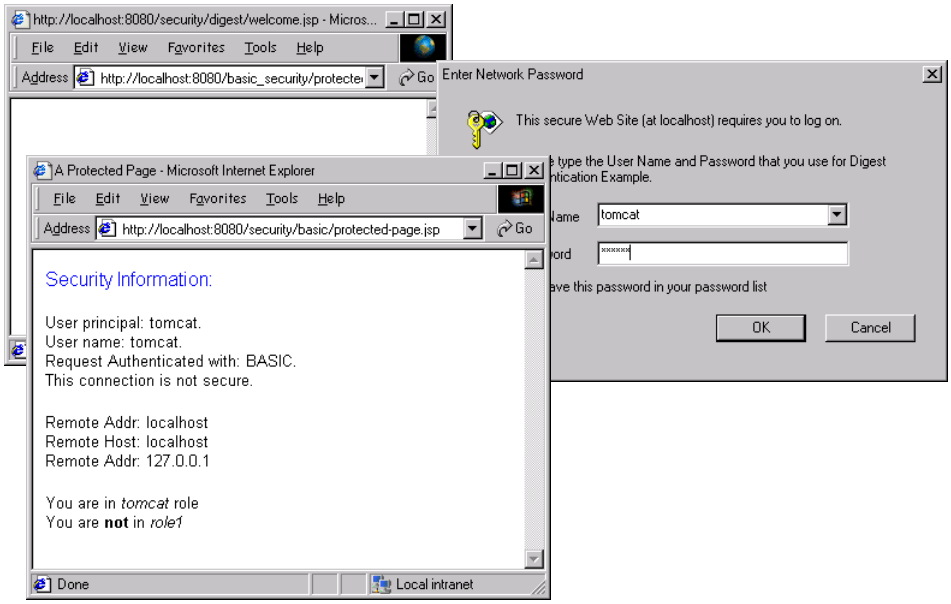

Figure 9-2 Digest Authentication with Tomcat

2. Digest authentication is also specified by the HTTP/1.1 specification; see `ftp://ftp.isi.edu/in-notes/rfc2617.txt`.

Digest authentication is specified in an application's deployment descriptor, like this:

```
<login-config>
      <auth-method>DIGEST</auth-method>
      <realm-name>Digest Authentication Example</realm-name>
   </login-config>
</web-app>
```

The only difference between basic and digest authentication is the specification of the authentication method, as listed above.

Note: The digest authentication example discussed in this section works with Tomcat 4.0, but not with Tomcat 3.2.1.

Form-Based Authentication

Form-based authentication allows you to control the look and feel of the login page. Form-based authentication works like basic authentication, except that you specify a login page that is displayed instead of a dialog and an error page that's displayed if login fails.

Like basic authentication, form-based authentication is not secure because passwords are transmitted as clear text. Unlike basic and digest authentication, form-based authentication is defined in the servlet specification, not the HTTP specification.

Form-based login allows customization of the login page, but not the authentication process itself. If you're interested in customizing the authentication of usernames and passwords, see "Customizing Authentication" on page 263.

Form-based authentication requires the following steps:

1. Implement a login page.
2. Implement an error page that will be displayed if login fails.
3. In the deployment descriptor, specify form-based authentication and the login and error pages from step #2.

Figure 9-3 shows an application that illustrates form-based authentication.

The top pictures in Figure 9-3 show a failed login, and the bottom pictures show subsequent success. Notice that the login form is displayed in the browser, not in a dialog, as is the case for basic and digest authentication.

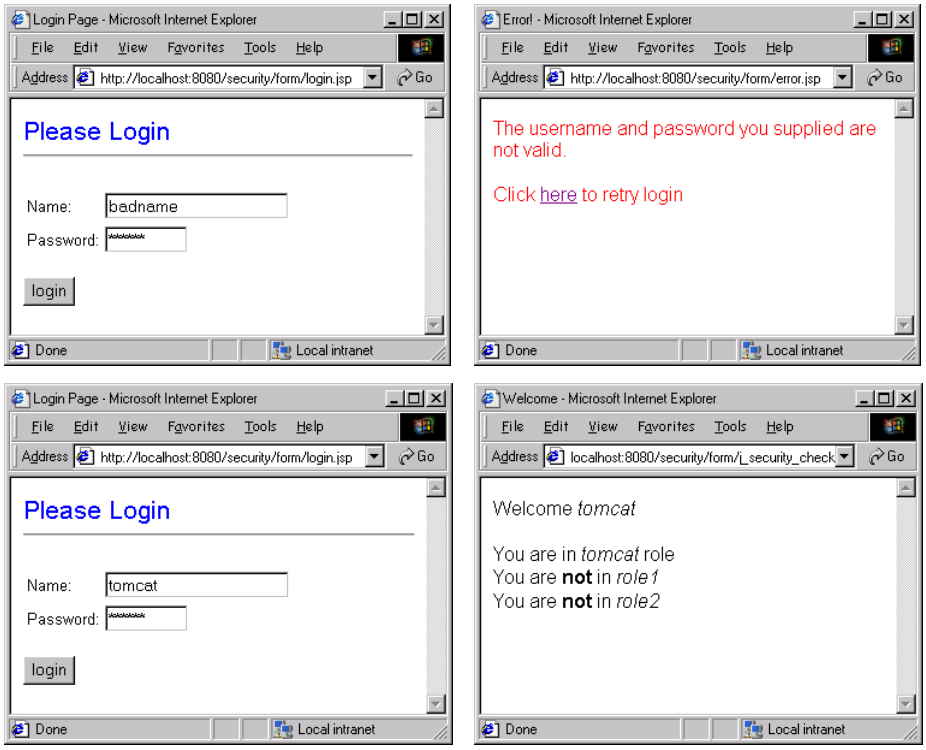

Figure 9-3 Form-Based Authentication with Tomcat

The login form used in Figure 9-3 is listed in Example 9-2.a.

Example 9-2.a /login.jsp

```
<html><head><title>Login Page</title></head>
<body>
<font size='5' color='blue'>Please Login</font><hr>

<form action='j_security_check' method='post'>
<table>
  <tr><td>Name:</td>
      <td><input type='text' name='j_username'></td></tr>
  <tr><td>Password:</td>
      <td><input type='password' name='j_password' size='8'></td>
  </tr>
</table>
<br>
   <input type='submit' value='login'>
</form></body>
</html>
```

The login page listed in Example 9-2.a is unremarkable except for the names of the name and password fields and the form's action. Those names, `j_username`, `j_password`, and `j_security_check`, respectively—which are defined in the Servlet Specification—must be used for form-based login. Table 9-3 summarizes those names.

Table 9-3 Login Form Attributes for Form-Based Login

Attribute	Description
`j_username`	The name of the username field
`j_password`	The name of the password field
`j_security_check`	The login form's action

The error page for the application shown in Figure 9-3 is listed in Example 9-2.b.

Example 9-2.b /error.jsp

```
<html> <head> <title>Error!</title></head>
<body>

<font size='4' color='red'>
   The username and password you supplied are not valid.
</p>
Click <a href='<%= response.encodeURL("login.jsp") %>'>here</a>
to retry login

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

The error page displays an error message and provides a link back to the login page. The deployment descriptor for the application shown in Figure 9-3 is listed in Example 9-2.c.

Example 9-2.c /WEB-INF/web.xml

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

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
   <security-constraint>
      <web-resource-collection>
```

```
            <web-resource-name>A Protected Page</web-resource-name>
            <url-pattern>/protected-page.jsp</url-pattern>
         </web-resource-collection>

         <auth-constraint>
            <role-name>tomcat</role-name>
         </auth-constraint>
      </security-constraint>

      <login-config>
         <auth-method>FORM</auth-method>
         <form-login-config>
            <form-login-page>/login.jsp</form-login-page>
            <form-error-page>/error.jsp</form-error-page>
         </form-login-config>
      </login-config>
   </web-app>
```

The deployment descriptor listed in Example 9-2.c specifies a security constraint that restricts access to `/protected-page.jsp` to principals in the role of `tomcat`. The authentication method is specified as `FORM`, and the login and error pages are identified.

SSL and Client Certificate Authentication

Secure sockets layer (SSL) is a secure transport mechanism that ensures privacy and data integrity through encryption. Additionally, SSL allows verification of client and server identity. For more information on SSL, see `http://home.netscape.com/eng/ssl3/3-SPEC.HTM`.

SSL is designed so that it can be layered on top of existing servers. The details of adding SSL to a web server are server dependent; see your server documentation for details. Resin's technical FAQ provides detailed instructions for layering SSL on stand-alone Resin; it can be found at `http://www.caucho.com/products/resin/ref/faq.xtp`.

Client certificate authentication is implemented with SSL and requires the client to possess a public key certificate. Although Tomcat 4.0 plans to support client certificate authentication, at the time of this writing it did not.

Customizing Authentication

There are two aspects to authentication: *challenging* principals for usernames and passwords and *authenticating* usernames and passwords. The servlet specification requires servlet containers to allow customization of the former with form-based

authentication, as discussed in “Form-Based Authentication” on page 260. The servlet specification does not require servlet containers to allow customization of the latter, but most servlet containers let you do so.

Because the servlet specification does not provide a standard mechanism for customizing authentication of usernames and passwords, that kind of customization is inherently nonportable. This section describes how to customize authentication with Resin and Tomcat and should give you a good idea of what to look for if you are using a different servlet container.

Resin

Resin authenticates usernames and passwords with authenticators, which are classes that implement the Resin `Authenticator` interface.

The default Resin authenticator will authenticate any combination of username and password—a useful feature if you are using Resin in combination with Apache or IIS, because you can rely on the web server’s authentication. If you are using Resin in stand-alone mode, then you need to implement an authenticator for basic authentication.

Figure 9-4 shows a basic authentication example with Resin.

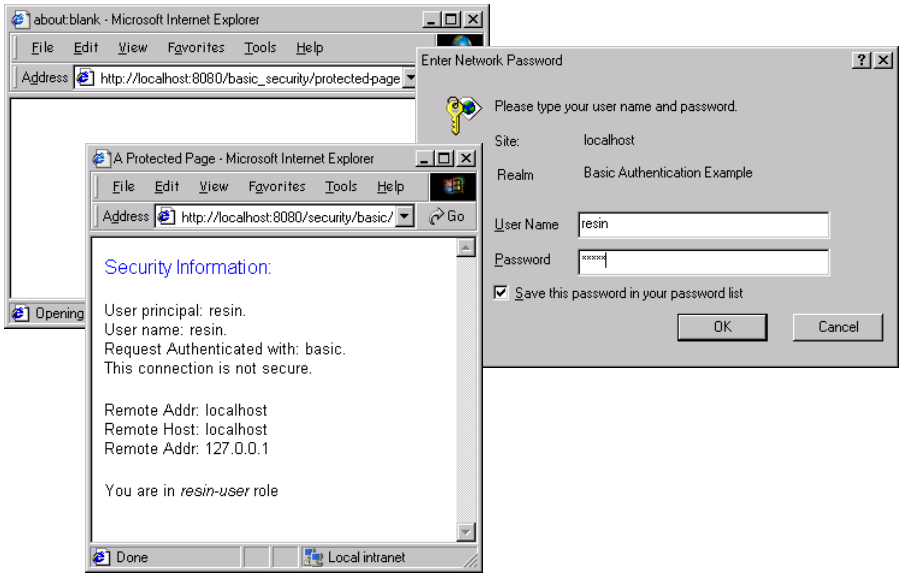

Figure 9-4 Customizing Basic Authentication with Resin

The protected page shown in Figure 9-4 is listed in Example 9-3.a.

Example 9-3.a /protected-page.jsp

```
<html><head><title>A Protected Page</title></head>
<body>

<%@ include file='show-security.jsp' %></p>
<p>
<% if(request.isUserInRole("resin-user")) { %>
   You are in <i>resin-user</i> role<br/>
<% } else {%>
   You are <b>not</b> in <i>resin-user</i> role<br/>
<% } %>
</p>
</body>
</html>
```

The JSP page listed in Example 9-3.a relies on the `show-security` JSP page to print security information; see "Basic Authentication" on page 256 for more information about that page. The JSP page listed in Example 9-3.a also verifies the user's role.

Example 9-3.b lists the deployment descriptor for the application shown in Figure 9-4.

Example 9-3.b /WEB-INF/web.xml

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

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
   <security-constraint>
      <!-- web resources that are protected -->
      <web-resource-collection>
         <web-resource-name>A Protected Page</web-resource-name>
         <url-pattern>/protected-page.jsp</url-pattern>
      </web-resource-collection>

      <auth-constraint>
         <role-name>resin-user</role-name>
      </auth-constraint>
   </security-constraint>
```

```
   <login-config>
      <auth-method>BASIC</auth-method>
      <realm-name>Basic Authentication Example</realm-name>
         <!-- The authenticator tag is Resin-specific -->
         <authenticator id='beans.SimpleAuthenticator'/>
   </login-config>
</web-app>
```

The deployment descriptor listed in Example 9-3.b restricts access to `/protected-page.jsp` to principals in the role of `resin-user` and specifies `BASIC` as the authentication method. That deployment descriptor also contains a Resin-specific authenticator tag that specifies the authenticator to use for this authentication. That authenticator is listed in Example 9-3.c.

Example 9-3.c /WEB-INF/classes/beans/SimpleAuthenticator.java

```
package beans;

import com.caucho.server.http.AbstractAuthenticator;
import com.caucho.server.http.BasicPrincipal;
import java.security.Principal;

public class SimpleAuthenticator extends AbstractAuthenticator {
   public Principal authenticate(String user, String password) {
      boolean valid = password != null &&
                      password.equals("resin") &&
                      user != null && user.equals("resin");

      if(valid) return new BasicPrincipal(user);
      else        return null;
   }
   public boolean isUserInRole(Principal user, String role) {
      return user.getName().equals("resin") &&
             role.equals("resin-user");
   }
}
```

The authenticator listed in Example 9-3.c extends the Resin `AbstractAuthenticator` class and overrides the `authenticate` and `isUserInRole` methods, both of which are defined in the `Authenticator` interface and given default implementations in `AbstractAuthenticator`.

The authenticate method returns an instance of `BasicPrincipal`, which is a Resin-specific class from `com.caucho.server.http`, if the username and password are authentic; otherwise, the method returns `null`.

Tomcat 4.0

Tomcat 4.0 uses realms, which are similar in principle to Resin's authenticators, to authenticate usernames and passwords. Unlike Resin, Tomcat does not require special tags in `/WEB-INF/web.xml`; instead, Tomcat specifies a realm in `$TOMCAT_HOME/conf/server.xml`, like this:

```
...
<!-- From $TOMCAT_HOME/conf/server.xml -->
<!-- Example Server Configuration File -->
<!-- Note that component elements are nested corresponding to their
     parent-child relationships with each other -->

<Server port="8005" shutdown="SHUTDOWN" debug="0">
...
    <!-- Because this Realm is here, an instance will be
         shared globally
    <Realm className="org.apache.catalina.realm.MemoryRealm" />
      -->
    <Realm className="CustomRealm"/>
...
</Server>
```

Just inside the `Server` start tag, Tomcat specifies a default realm—`org.apache.catalina.realm.MemoryRealm`—which is shared by all contexts.[3] To replace the default realm, comment out the default and insert your own, as listed above.

Tomcat custom realms typically extend the Tomcat `RealmBase` abstract class, which implements the `Realm` interface. `RealmBase` defines three abstract methods that extensions must implement. Those methods are listed in Table 9-4.

Table 9-4 Tomcat 4.0 `RealmBase` Abstract Methods

Method	Intent
`boolean hasRole(Principal principal, String role)`	Returns `true` if a role is suitable for a principal
`String getPassword(String user)`	Returns a password associated with a user
`Principal getPrincipal(String user)`	Returns a principal associated with a user

3. Tomcat realms can also be specified for individual web applications.

The `CustomRealm` class referred to in the `server.xml` file listed above is listed in Example 9-4.

Example 9-4 A Tomcat Custom Realm

```
import java.security.Principal;
import org.apache.catalina.realm.RealmBase;

public class CustomRealm extends RealmBase {
   public boolean hasRole(Principal principal, String role) {
      String name = principal.getName();

      if(name.equals("tomcat"))
         return role.equals("tomcat");

      if(name.equals("role1"))
         return role.equals("role1");

      if(name.equals("both"))
         return role.equals("tomcat") || role.equals("role1");

      return false;
   }
   protected String getPassword(String username) {
      return "tomcat";
   }
   protected Principal getPrincipal(String username) {
      return new CustomPrincipal(username);
   }
   class CustomPrincipal implements Principal {
      private final String name;

      public CustomPrincipal(String name) {
         this.name = name;
      }
      public String getName() {
         return name;
      }
      public String toString() {
         return getName();
      }
   }
}
```

The custom realm listed in Example 9-4 is designed to work with the default entries from `$TOMCAT_HOME/conf/tomcat-users.xml`, which is listed in Example 9-1.d on page 258. For example, `hasRole` returns `true` if the principal

and role correspond to those specified in `tomcat-users.xml`. The `getPassword` method returns `tomcat`, which is the password used for all of the users defined in `tomcat-users.xml`. The `getPrincipal` method returns a custom principal, which is a simple implementation of the `java.security.Principal` interface.

Custom realms must be made available to Tomcat at startup, which requires that custom realm classes reside in a JAR file in `$TOMCAT_HOME/server`. So, for the example listed above to work, `CustomRealm.java` is compiled, yielding two class files. Those class files are placed in a JAR file and copied to `$TOMCAT_HOME/server`.

Note: The code in this section is based on a beta version of Tomcat 4.0, so that code may need to be modified by the time you read this.

Web Application Security Elements

This section provides a reference to security elements from the Servlet 2.2 specification. A number of the examples in this chapter have illustrated the use of most of these elements; for example, see Example 9-1.c on page 258.

Table 9-5 lists the elements contained within a `security-constraint` element, which is the outermost security element in a deployment descriptor.

Table 9-5 `<security-constraint>` Elements

Element	Type[1]	Description
`web-resource-collection`	+	A subset of a web application's resources to which security constraints apply
`auth-constraint`	?	Authorization constraints placed on one or more web resource collections
`user-data-constraint`	?	A specification of how data sent between a client and a container should be protected

1. + = one or more? = one, optional

Web resource collections identify one or more protected resources, and authorization constraints specify one or more roles that can access those resources. User data constraints specify how data should be protected while in transit.

Table 9-6 lists web resource collection elements.

Table 9-6 `<web-resource-collection>` Elements

Element	Type[1]	Description
`web-resource-name`	1	The name of a web resource
`description`	?	A description of a web resource
`url-pattern`	*	A url pattern associated with a web resource
`http-method`	?	An HTTP method associated with a web resource

1. 1 = one, required? = zero or more* = one or more

Each web resource collection is associated with the name of a resource and an optional description of that resource. One or more URL patterns are associated with a resource name.

HTTP methods may also be associated with a web resource collection; for example, if `GET` is specified as the HTTP method, the security constraint is only enforced for `GET` requests. If no HTTP methods are specified, the corresponding security constraint applies to all HTTP requests for the specified resources.

Table 9-7 lists authorization constraint elements.

Table 9-7 `<auth-constraint>` Elements

Element	Type[1]	Description
`description`	?	A description of an authorization constraint
`role-name`	*	The role(s) to which a constraint applies

1. ? = zero or more* = one or more

Authorization constraints specify one or more roles that are allowed access to protected resources. Optionally, those roles can be accompanied by a description.

Table 9-8 lists user data constraint elements.

Table 9-8 `<user-data-constraint>` Elements

Element	Type[1]	Description
`description`	?	A description of a user data constraint
`transport-guarantee`	1	`NONE`, `INTEGRAL`, or `CONFIDENTIAL`

1. ? = zero or more 1 = one, required

User data constraints consist of a `transport-guarantee` and an optional description. That guarantee can be either `NONE`, `INTEGRAL`, or `CONFIDENTIAL`. A guarantee of `NONE` means there are no restrictions on the transport of data, and `INTEGRAL` means the servlet container must ensure that data cannot be changed

in transit. A value of `CONFIDENTIAL` means that the data cannot be read while in transit.

The servlet specification does not specify how servlet containers should implement transport guarantees; however, a value of `INTEGRAL` or `CONFIDENTIAL` typically indicates a secure transport layer, such as SSL. Resin, for example, will only provide access to confidential data if `ServletRequest.isSecure` returns `true`.[4]

Programmatic Authentication

The word programmatic here means implemented from scratch, which is a good choice for authentication if you must have portability or if you want total control. Because it's more work than relying on your servlet container, programmatic authentication can be a bad choice if you are not interested in those benefits.

Another drawback to programmatic authentication is that `HttpServletRequest.getUserPrincipal`, `HttpServletRequest.getRemoteUser`, and `HttpServletRequest.isUserInRole` are rendered useless for applications with programmatic authentication. Programmatic authentication requires you to implement, and use, your own API because setting principals and roles is strictly for servlet containers. See "Principals and Roles" on page 252 for more information about setting principals and roles.

The rest of this section discusses an authentication mechanism implemented from scratch; if you're interested in something similar, you can use it for ideas or perhaps as a starting point.

The authentication mechanism discussed in this section entails protecting JSP pages with a custom tag, like this:

```
<!--A protected JSP page-->
...
<%@ taglib='/WEB-INF/tlds/security' prefix='security' %>
...
<!-- errorPage is optional; if unspecified, control goes back to
     loginPage if login fails -->

<security:enforceLogin loginPage='/login.jsp'
                       errorPage='/error.jsp'/>

<!--The rest of the file is accessed only if a user has logged
     into this session -->
...
```

4. See page 465 for more information on the `CONFIDENTIAL` transport guarantee and SSL.

The `enforceLogin` tag looks for a user in session scope. If the user is in the session, the tag does nothing; if not, the tag forwards to the login page. The login page is specified with the `loginPage` attribute.

If login fails, control is forwarded to the error page. The `errorPage` attribute is optional; without it, the login page is redisplayed if login fails.

When login succeeds, a user is created and placed in session scope, and the rest of the page after the `enforceLogin` tag is evaluated.

Figure 9-5 provides a more visual representation of the sequence of events initiated by the `enforceLogin` tag.

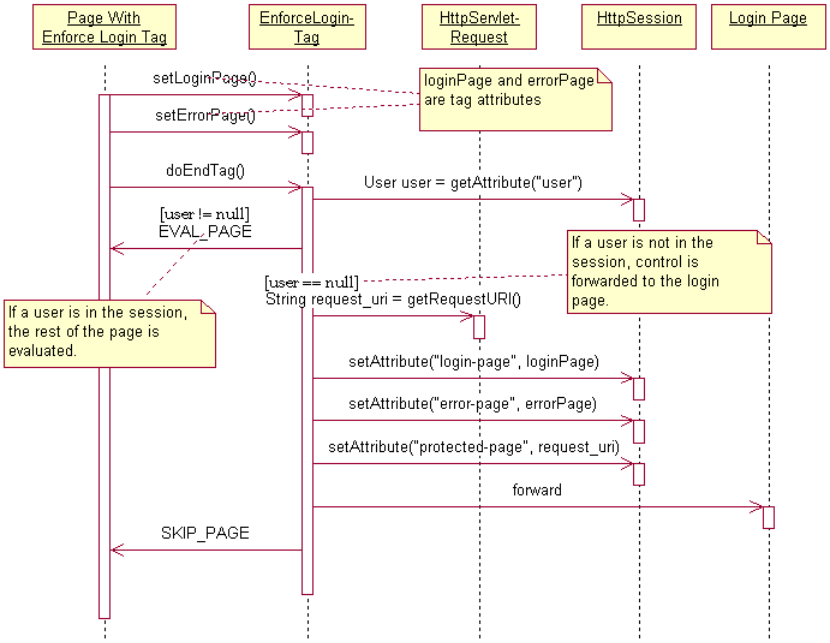

Figure 9-5 Enforce Login Tag Sequence Diagram

If no user is in session scope, three session attributes, listed in Table 9-9, are set by the `enforceLogin` tag.

The attributes listed in Table 9-9 determine how the request is subsequently handled; the first two correspond to the `loginPage` and `errorPage` attributes of the `enforceLogin` tag, respectively. The `protected-page` attribute represents the `URI` of the protected page.

Table 9-9 Session Attributes Set by the `enforceLogin` Tag

Attribute Name	Description
`login-page`	The `enforceLogin` tag forwards to this page if there's no user in the session. If login subsequently fails and no error page is specified, control is returned to this page.
`error-page`	An optional error page that's displayed when login fails
`protected-page`	The page with the `enforceLogin` tag; when login succeeds, the rest of the page after that tag is evaluated.

The login page submits the login form to a servlet. If that servlet authenticates the username and password, it redirects the request to the protected page; otherwise, it forwards to the error page, if specified, or back to the login page, if not.

Figure 9-6 shows an example that uses the programmatic authentication discussed in this section.

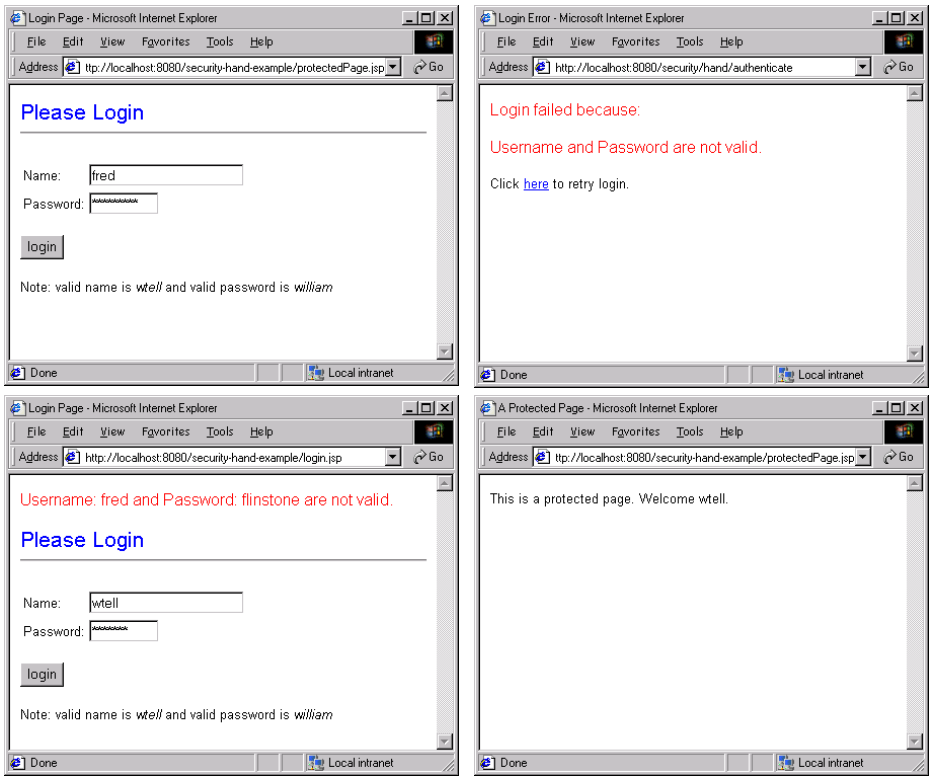

Figure 9-6 Programmatic Authentication

The top two pictures in Figure 9-6 show a failed login, and the bottom two show subsequent success. Figure 9-7 shows the files involved in the application shown in Figure 9-6.

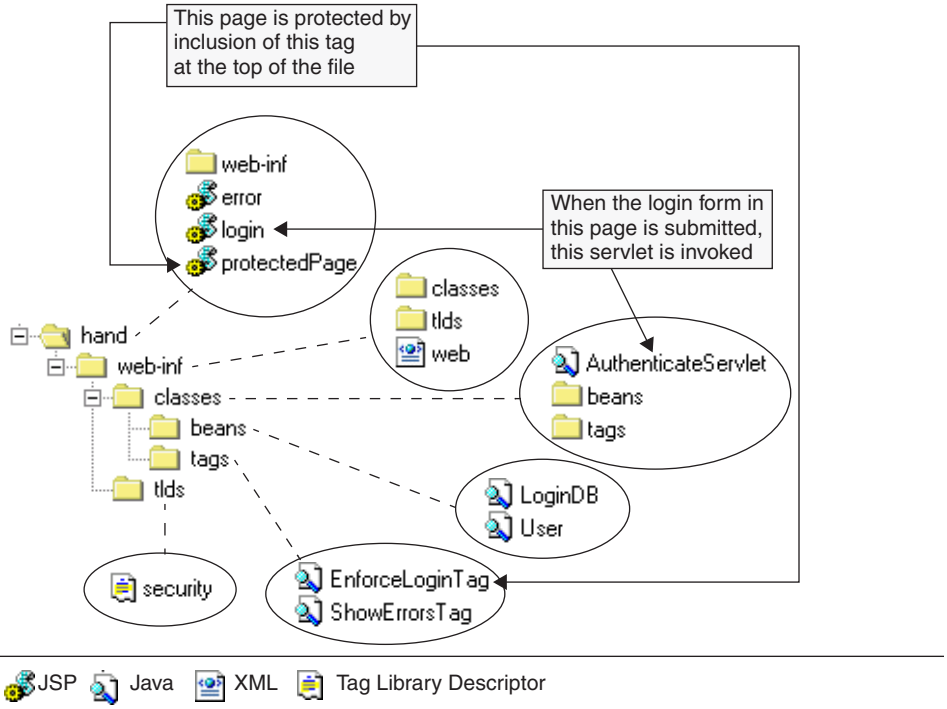

Figure 9-7 Files for the Programmatic Authentication Example

The application maintains a makeshift database of users. That database is an instance of `LoginDB` and users are `User` instances; those classes are listed in Example 5-1.b on page 139 and Example 5-1.a on page 138, respectively. This implementation of `LoginDB` adds a default user, as listed in Example 9-5.a.

Example 9-5.a /WEB-INF/classes/beans/LoginDB.java

```
// The User class is listed in Example 5-1.a on page 138.
...
public class LoginDB implements java.io.Serializable {

   private Vector users = new Vector();
   private User[] defaultUsers = {
      new User("wtell", "william", "my first name"),
   };
```

```
    public LoginDB() {
       for(int i=0; i < defaultUsers.length; ++i)
          users.add(defaultUsers[i]);
    }
    public void addUser(String uname, String pwd, String hint) {
       users.add(new User(uname, pwd, hint));
    }
    // The rest of this class is identical to LoginDB listed in
    // Example 5-1.b on page 139.
    ...
}
```

The application shown in Figure 9-6 has one protected page, listed in Example 9-5.b.

Example 9-5.b /protectedPage.jsp

```
<html><head><title>A Protected Page</title></head>
<%@ taglib uri='security' prefix='security' %>
</body>

<!-- Without the errorPage attribute, control is forwarded back
     to the login page if login fails.  -->

<security:enforceLogin loginPage='/login.jsp'
                       errorPage='/error.jsp'/>

<jsp:useBean id='user' type='beans.User' scope='session'/>

This is a protected page. Welcome <%= user.getUserName() %>.

</body>
</html>
```

The protected page accesses the user in the session to display a welcome message. The `enforceLogin` tag handler is listed in Example 9-5.c.

Example 9-5.c /WEB-INF/classes/tags/EnforceLoginTag.java

```
package tags;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.tagext.TagSupport;

public class EnforceLoginTag extends TagSupport {
   private String loginPage, errorPage;
```

```
    public void setLoginPage(String loginPage) {
       this.loginPage = loginPage;
    }
    public void setErrorPage(String errorPage) {
       this.errorPage = errorPage;
    }
    public int doEndTag() throws JspException {
       HttpSession session = pageContext.getSession();
       HttpServletRequest req = (HttpServletRequest)pageContext.
                                                    getRequest();
       String protectedPage = req.getRequestURI();

       if(session.getAttribute("user") == null) {
          session.setAttribute("login-page",     loginPage);
          session.setAttribute("error-page",     errorPage);
          session.setAttribute("protected-page", protectedPage);

          try {
             pageContext.forward(loginPage);
             return SKIP_PAGE;
          }
          catch(Exception ex) {
             throw new JspException(ex.getMessage());
          }
       }
       return EVAL_PAGE;
    }
    public void release() {
       loginPage = errorPage = null;
    }
}
```

If there's a user in the session, the tag handler listed in Example 9-5.c returns `EVAL_PAGE` and the rest of the page after the tag is evaluated. If the user is not in the session, the attributes listed in Table 9-9 on page 273 are set and control is forwarded to the login page.

The login page is listed in Example 9-5.d.

Example 9-5.d /login.jsp

```
<html><head><title>Login Page</title></head>
<%@ taglib uri='/WEB-INF/tlds/security.tld' prefix='security' %>
<body>

<font size='4' color='red'><security:showErrors/></font>

<p><font size='5' color='blue'>Please Login</font><hr>
<form action='<%= response.encodeURL("authenticate") %>'
```

```
      method='post'>
   <table>
      <tr>
         <td>Name:</td>
         <td><input type='text' name='userName'/>
         </td>
      </tr><tr>
         <td>Password:</td>
         <td><input type='password' name='password' size='8'></td>
      </tr>
   </table>
   <br>
   <input type='submit' value='login'>
</form></p>

Note: valid name is <i>wtell</i> and valid
password is <i>william</i>

</body>
</html>
```

The login form is submitted to the `authenticate` servlet, which generates error messages in session scope if authentication fails. Those messages are displayed by the `security:showErrors` tag at the top of the login page. The mappings between the name `authenticate` and the authenticate servlet are specified in `web.xml`, which is listed in Example 9-5.e.

Example 9-5.e /WEB-INF/web.xml

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

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
   <servlet>
      <servlet-name>authenticate</servlet-name>
      <servlet-class>AuthenticateServlet</servlet-class>
   </servlet>

   <servlet-mapping>
      <servlet-name>authenticate</servlet-name>
      <url-pattern>/authenticate</url-pattern>
   </servlet-mapping>

   <taglib>
    <taglib-uri>/WEB-INF/tlds/security.tld</taglib-uri>
```

```
    <taglib-location>/WEB-INF/tlds/security.tld</taglib-location>
  </taglib>
</web-app>
```

Example 9-5.f lists the authenticate servlet.

Example 9-5.f /WEB-INF/classes/AuthenticateServlet.java

```
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import java.io.IOException;
import beans.LoginDB;
import beans.User;

public class AuthenticateServlet extends HttpServlet {
   private LoginDB loginDB;

   public void init(ServletConfig config) throws ServletException{
      super.init(config);
      loginDB = new LoginDB();
   }
   public void service(HttpServletRequest req,
                       HttpServletResponse res)
                          throws IOException, ServletException {
      HttpSession session = req.getSession();
      String      uname   = req.getParameter("userName");
      String      pwd     = req.getParameter("password");
      User        user    = loginDB.getUser(uname, pwd);

      if(user != null) { // authorized
         String protectedPage = (String)session.
                                   getAttribute("protected-page");
         session.removeAttribute("login-page");
         session.removeAttribute("error-page");
         session.removeAttribute("protected-page");
         session.removeAttribute("login-error");

         session.setAttribute("user", user);
         res.sendRedirect(res.encodeURL(protectedPage));
      }
      else { // not authorized
         String loginPage = (String)session.
                            getAttribute("login-page");
         String errorPage = (String)session.
                            getAttribute("error-page");
```

```
            String forwardTo = errorPage != null ? errorPage :
                                                   loginPage;
            session.setAttribute("login-error",
                     "Username and Password are not valid.");

            getServletContext().getRequestDispatcher(
               res.encodeURL(forwardTo)).forward(req,res);
        }
    }
}
```

The authenticate servlet obtains the username and password from the request and attempts to obtain a reference to a corresponding user in the login database. If the user exists in the database, session attributes generated by the servlet and the `enforceLogin` tag are removed from the session and the request is redirected to the protected page. Figure 9-8 shows the sequence of events for a successful login.

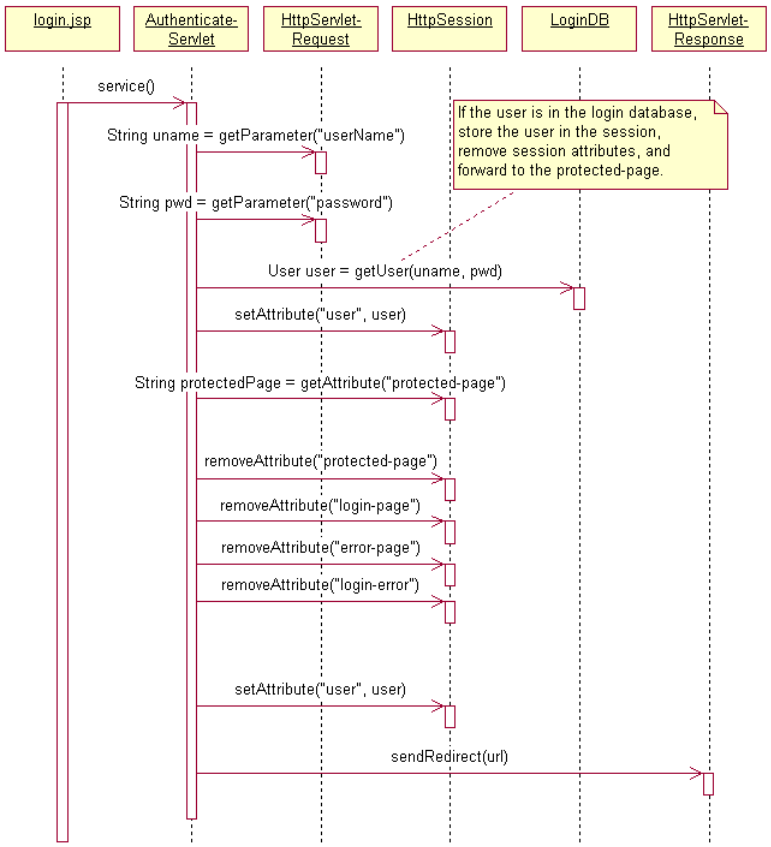

Figure 9-8 Login Succeeds Sequence Diagram

If the user is not in the login database, a `login-error` session attribute is set and the request is forwarded to the error page, if specified, or back to the login page, if not. Figure 9-9 shows the sequence of events for a failed login.

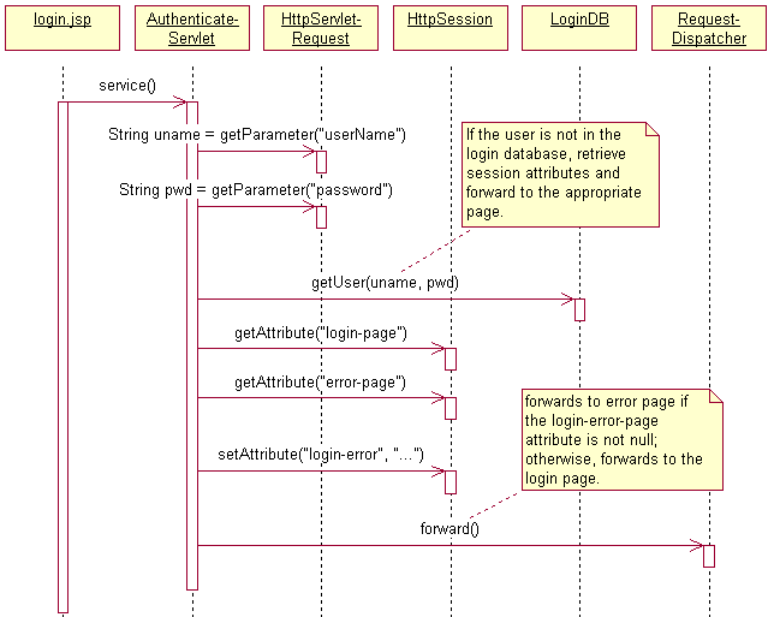

Figure 9-9 Login Fails Sequence Diagram

The error page for the application in Figure 9-6 on page 273 is listed in Example 9-5.g.

Example 9-5.g /error.jsp

```
<html><head><title>Login Error</title></head>
<%@ taglib uri='/WEB-INF/tlds/security.tld' prefix='security' %>
<body>

<font size='4' color='red'>
Login failed because:<p>
<security:showErrors/></font></p>

Click <a href='login.jsp'>here</a> to retry login.

</body>
</html>
```

Like the login page, the error page uses the `security:showErrors` tag, whose handler is listed in Example 9-5.h.

Example 9-5.h /WEB-INF/classes/tags/ShowErrorsTag.java

```
package tags;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.tagext.TagSupport;

public class ShowErrorsTag extends TagSupport {
   public int doStartTag() throws JspException {
      String error = (String)pageContext.getSession().
                              getAttribute("login-error");
      if(error != null) {
         try {
            pageContext.getOut().print(error);
         }
         catch(java.io.IOException ex) {
            throw new JspException(ex.getMessage());
         }
      }
      return SKIP_BODY;
   }
}
```

The `showErrors` tag handler prints the value of the `login-error` session attribute that was set by the authenticate servlet.

Conclusion

Security is an important aspect of applications that transport sensitive data over the Internet. Because of this requirement, the servlet specification requires servlet containers to provide implementations of basic and digest authentication, as defined in the HTTP/1.1 specification. Additionally, servlet containers must provide form-based security that allows developers to control the look and feel of login screens. Finally, servlet containers may provide SSL and client certificate authentication, although containers that are not J2EE compliant are not required to do so.

Unlike other aspects of web applications implemented with JSP and the Java programming language, security typically requires some nonportable code. If portability is a high priority, you can implement security from scratch by using JSP and servlets, as illustrated in "Programmatic Authentication" on page 271.