Getting Started Securing Web Applications

A web application is accessed using a web browser over a network, such as the Internet or a company's intranet.

As discussed in Distributed Multitiered Applications, the Java EE platform uses a distributed multitiered application model, and web applications run in the web tier.

Web applications contain resources that can be accessed by many users.

These resources often traverse unprotected, open networks, such as the Internet.

In such an environment, a substantial number of web applications will require some type of security.

The ways to implement security for Java EE web applications are discussed in a general way in Securing Containers.

This chapter provides more detail and a few examples that explore these security services as they relate to web components.

Securing applications and their clients in the business tier and the EIS tier is discussed in Chapter 41, Getting Started Securing Enterprise Applications.

The following topics are addressed here:

- Overview of Web Application Security
- Securing Web Applications
- Using Programmatic Security with Web Applications
- Examples: Securing Web Applications

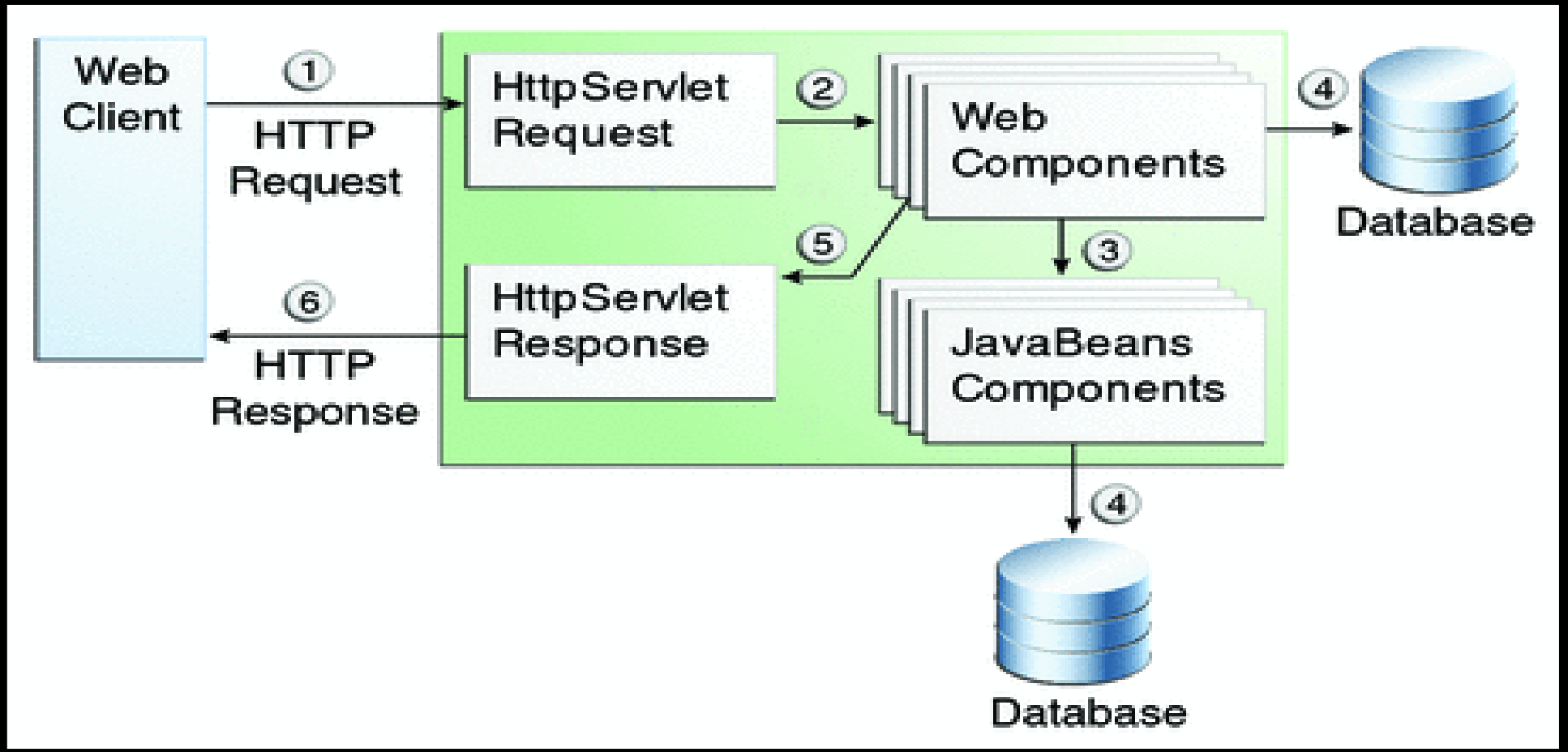
Overview of Web Application Security

In the Java EE platform, web components provide the dynamic extension capabilities for a web server.

Web components can be Java servlets or JavaServer Faces pages.

The interaction between a web client and a web application is illustrated in Figure 40-1.

Figure 40-1 Java Web Application Request Handling



Certain aspects of web application security can be configured when the application is installed, or deployed, to the web container.

Annotations and/or deployment descriptors are used to relay information to the deployer about security and other aspects of the application.

Specifying this information in annotations or in the deployment descriptor helps the deployer set up the appropriate security policy for the web application.

Any values explicitly specified in the deployment descriptor override any values specified in annotations.

Security for Java EE web applications can be implemented in the following ways.

- **Declarative security:** Can be implemented using either metadata annotations or an application's deployment descriptor.

See Overview of Java EE Security for more information.

Declarative security for web applications is described in Securing Web Applications.

- Programmatic security: Is embedded in an application and can be used to make security decisions when declarative security alone is not sufficient to express the security model of an application.

Declarative security alone may not be sufficient when conditional login in a particular work flow, instead of for all cases, is required in the middle of an application.

See Overview of Java EE Security for more information.

Servlet 3.0 provides the `authenticate`, `login`, and `logout` methods of the `HttpServletRequest` interface.

With the addition of the `authenticate`, `login`, and `logout` methods to the Servlet specification, an application deployment descriptor is no longer required for web applications but may still be used to further specify security requirements beyond the basic default values.

Programmatic security is discussed in Using Programmatic Security with Web Applications

- **Message Security:** Works with web services and incorporates security features, such as digital signatures and encryption, into the header of a SOAP message, working in the application layer, ensuring end-to-end security.

  Message security is not a component of Java EE 6 and is mentioned here for informational purposes only.

Some of the material in this chapter builds on material presented earlier in this tutorial.
In particular, this chapter assumes that you are familiar with the information in the following chapters:

- Chapter 3, Getting Started with Web Applications
- Chapter 4, JavaServer Faces Technology
- Chapter 15, Java Servlet Technology
- Chapter 39, Introduction to Security in the Java EE Platform

Securing Web Applications

Web applications are created by application developers who give, sell, or otherwise transfer the application to an application deployer for installation into a runtime environment.

Application developers communicate how to set up security for the deployed application by using annotations or deployment descriptors.

This information is passed on to the deployer, who uses it to define method permissions for security roles, set up user authentication, and set up the appropriate transport mechanism.

If the application developer doesn't define security requirements, the deployer will have to determine the security requirements independently.

Some elements necessary for security in a web application cannot be specified as annotations for all types of web applications.

This chapter explains how to secure web applications using annotations wherever possible.

It explains how to use deployment descriptors where annotations cannot be used.

Specifying Security Constraints

A security constraint is used to define the access privileges to a collection of resources using their URL mapping.

If your web application uses a servlet, you can express the security constraint information by using annotations.

Specifically, you use the `@HttpConstraint` and, optionally, the `@HttpMethodConstraint` annotations within the `@ServletSecurity` annotation to specify a security constraint.

If your web application does not use a servlet, however, you must specify a `security-constraint` element in the deployment descriptor file.

The authentication mechanism cannot be expressed using annotations, so if you use any authentication method other than `BASIC` (the default), a deployment descriptor is required.

The following subelements can be part of a `security-constraint`:

- Web resource collection (`web-resource-collection`):

A list of URL patterns (the part of a URL after the host name and port you want to constrain) and HTTP operations (the methods within the files that match the URL pattern you want to constrain) that describe a set of resources to be protected.

Web resource collections are discussed in Specifying a Web Resource Collection.

- Authorization constraint (`auth-constraint`):

Specifies whether authentication is to be used and names the roles authorized to perform the constrained requests.

For more information about authorization constraints, see Specifying an Authorization Constraint.

- User data constraint (`user-data-constraint`):

Specifies how data is protected when transported between a client and a server.

User data constraints are discussed in Specifying a Secure Connection.

Specifying a Web Resource Collection

A web resource collection consists of the following subelements:

- `web-resource-name` is the name you use for this resource.

  Its use is optional.

- `url-pattern` is used to list the request URI to be protected.

  Many applications have both unprotected and protected resources.

  To provide unrestricted access to a resource, do not configure a security constraint for that particular request URI.

  The request URI is the part of a URL after the host name and port.

For example, let's say that you have an e-commerce site with a catalog that you would want anyone to be able to access and browse, and a shopping cart area for customers only.

You could set up the paths for your web application so that the pattern `/cart/*` is protected but nothing else is protected.

Assuming that the application is installed at context path `/myapp`, the following are true:

- `http://localhost:8080/myapp/index.xhtml` is not protected.

- `http://localhost:8080/myapp/cart/index.xhtml` is protected.

A user will be prompted to log in the first time he or she accesses a resource in the `cart/` subdirectory.

- `http-method` or `http-method-omission` is used to specify which methods should be protected or which methods should be omitted from protection.

  An HTTP method is protected by a `web-resource-collection` under any of the following circumstances:

- If no HTTP methods are named in the collection (which means that all are protected)

- If the collection specifically names the HTTP method in an `http-method` subelement

- If the collection contains one or more `http-method-omission` elements, none of which names the HTTP method

Specifying an Authorization Constraint

An authorization constraint (`auth-constraint`) contains the `role-name` element.

You can use as many `role-name` elements as needed here.

An authorization constraint establishes a requirement for authentication and names the roles authorized to access the URL patterns and HTTP methods declared by this security constraint.

If there is no authorization constraint, the container must accept the request without requiring user authentication.

If there is an authorization constraint but no roles are specified within it, the container will not allow access to constrained requests under any circumstances.

Each role name specified here must either correspond to the role name of one of the `security-role` elements defined for this web application or be the specially reserved role name `*`, which indicates all roles in the web application.

Role names are case sensitive.

The roles defined for the application must be mapped to users and groups defined on the server, except when default principal-to-role mapping is used.

For more information about security roles, see Declaring Security Roles.

For information on mapping security roles, see Mapping Roles to Users and Groups.

For a servlet, the `@HttpConstraint` and `@HttpMethodConstraint` annotations accept a `rolesAllowed` element that specifies the authorized roles.

Specifying a Secure Connection

A user data constraint

(`user-data-constraint` in the deployment descriptor) contains the `transport-guarantee` subelement.

A user data constraint can be used to require that a protected transport-layer connection, such as HTTPS, be used for all constrained URL patterns and HTTP methods specified in the security constraint.

The choices for transport guarantee are `CONFIDENTIAL`, `INTEGRAL`, or `NONE`.

If you specify `CONFIDENTIAL` or `INTEGRAL` as a security constraint, it generally means that the use of SSL is required and applies to all requests that match the URL patterns in the web resource collection, not just to the login dialog box.

The strength of the required protection is defined by the value of the transport guarantee.

- Specify `CONFIDENTIAL` when the application requires that data be transmitted so as to prevent other entities from observing the contents of the transmission.

- Specify `INTEGRAL` when the application requires that the data be sent between client and server in such a way that it cannot be changed in transit.

- Specify `NONE` to indicate that the container must accept the constrained requests on any connection, including an unprotected one.

Note - In practice, Java EE servers treat the `CONFIDENTIAL` and `INTEGRAL` transport guarantee values identically.

The user data constraint is handy to use in conjunction with basic and form-based user authentication.

When the login authentication method is set to `BASIC` or `FORM`, passwords are not protected, meaning that passwords sent between a client and a server on an unprotected session can be viewed and intercepted by third parties.

Using a user data constraint with the user authentication mechanism can alleviate this concern.

Configuring a user authentication mechanism is described in Specifying an Authentication Mechanism in the Deployment Descriptor.

To guarantee that data is transported over a secure connection, ensure that SSL support is configured for your server.

SSL support is already configured for the GlassFish Server.

Note - After you switch to SSL for a session, you should never accept any non-SSL requests for the rest of that session.

For example, a shopping site might not use SSL until the checkout page, and then it might switch to using SSL to accept your card number.

After switching to SSL, you should stop listening to non-SSL requests for this session.

The reason for this practice is that the session ID itself was not encrypted on the earlier communications.

This is not so bad when you're only doing your shopping, but after the credit card information is stored in the session, you don't want anyone to use that information to fake the purchase transaction against your credit card.

This practice could be easily implemented by using a filter.

Specifying Separate Security Constraints for Various Resources

You can create a separate security constraint for various resources within your application.

For example, you could allow users with the role of `PARTNER` access to the `GET` and `POST` methods of all resources with the URL pattern `/acme/wholesale/*` and allow users with the role of `CLIENT` access to the `GET` and `POST` methods of all resources with the URL pattern `/acme/retail/*`.

An example of a deployment descriptor that would demonstrate this functionality is the following:

```
<!-- SECURITY CONSTRAINT #1 -->
<security-constraint>
<web-resource-collection>
<web-resource-name>
wholesale
</web-resource-name>
<url-pattern>
/acme/wholesale/*
</url-pattern>
<http-method>GET</http-method>
<http-method>POST</http-method>
```

```
</web-resource-collection>
<auth-constraint>
<role-name>PARTNER</role-name>
</auth-constraint>
<user-data-constraint>
<transport-guarantee>
CONFIDENTIAL
</transport-guarantee>
</user-data-constraint>
</security-constraint>
```

```
<!-- SECURITY CONSTRAINT #2 -->
<security-constraint>
<web-resource-collection>
<web-resource-name>
retail
</web-resource-name>
<url-pattern>
/acme/retail/*
</url-pattern>
<http-method>GET</http-method>
<http-method>POST</http-method>
```

```
</web-resource-collection>
<auth-constraint>
<role-name>CLIENT</role-name>
</auth-constraint>
<user-data-constraint>
<transport-guarantee>
CONFIDENTIAL
</transport-guarantee>
</user-data-constraint>
</security-constraint>
```

When the same `url-pattern` and `http-method` occur in multiple security constraints, the constraints on the pattern and method are defined by combining the individual constraints, which could result in unintentional denial of access.

Specifying Authentication Mechanisms

A user authentication mechanism specifies

- The way a user gains access to web content
- With basic authentication, the realm in which the user will be authenticated
- With form-based authentication, additional attributes

When an authentication mechanism is specified, the user must be authenticated before access is granted to any resource that is constrained by a security constraint.

There can be multiple security constraints applying to multiple resources, but the same authentication method will apply to all constrained resources in an application.

Before you can authenticate a user, you must have a database of user names, passwords, and roles configured on your web or application server.

For information on setting up the user database, see Managing Users and Groups on the GlassFish Server.

HTTP basic authentication and form-based authentication are not very secure authentication mechanisms.

Basic authentication sends user names and passwords over the Internet as Base64-encoded text; form-based authentication sends this data as plain text.

In both cases, the target server is not authenticated.

Therefore, these forms of authentication leave user data exposed and vulnerable.

If someone can intercept the transmission, the user name and password information can easily be decoded.

However, when a secure transport mechanism, such as SSL, or security at the network level, such as the Internet Protocol Security (IPsec) protocol or virtual private network (VPN) strategies, is used in conjunction with basic or form-based authentication, some of these concerns can be alleviated.

To specify a secure transport mechanism, use the elements described in Specifying a Secure Connection.

HTTP Basic Authentication

Specifying HTTP basic authentication requires that the server request a user name and password from the web client and verify that the user name and password are valid by comparing them against a database of authorized users in the specified or default realm.

Basic authentication is the default when you do not specify an authentication mechanism.

When basic authentication is used, the following actions occur:

1. A client requests access to a protected resource.

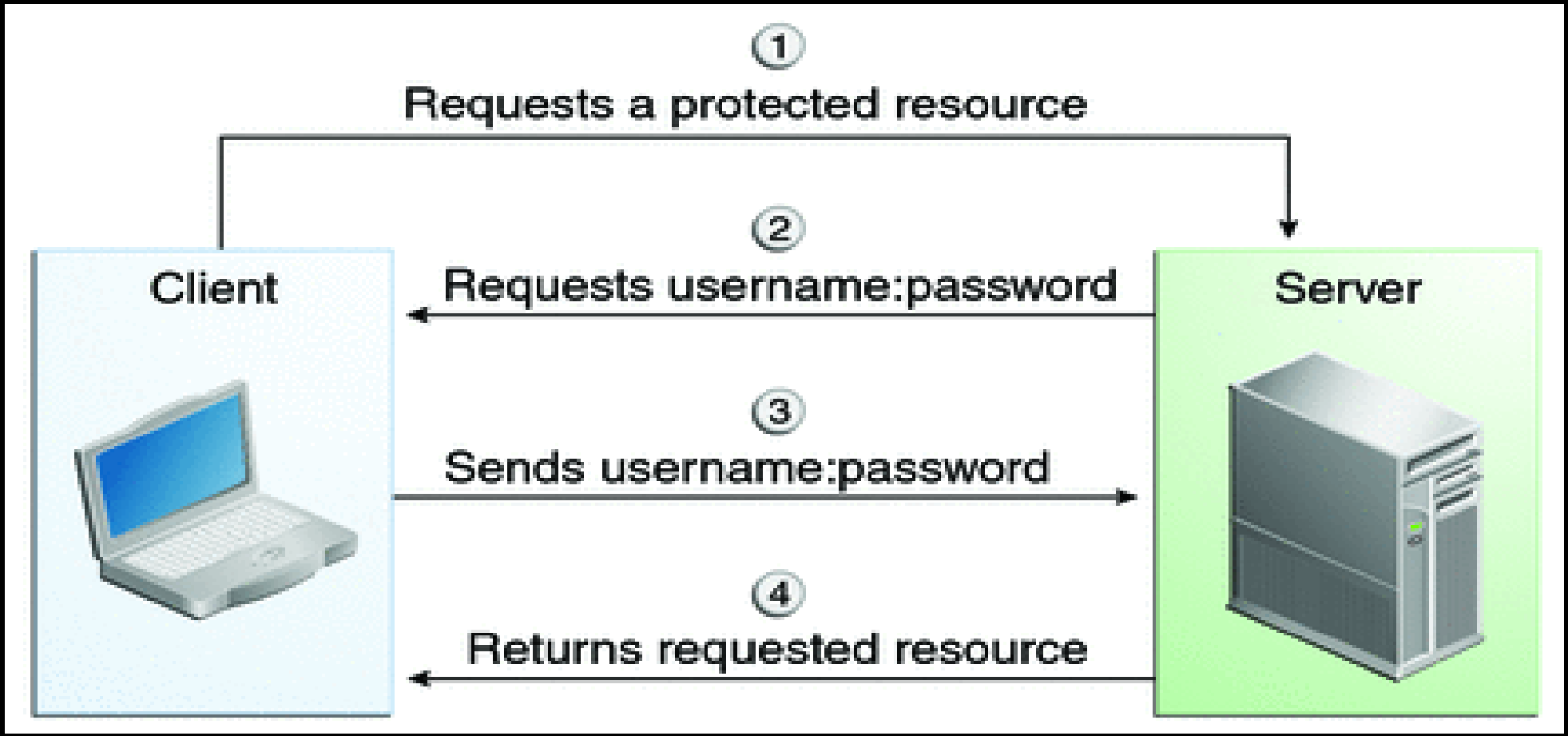
2. The web server returns a dialog box that requests the user name and password.

3. The client submits the user name and password to the server.

4. The server authenticates the user in the specified realm and, if successful, returns the requested resource.

Figure 40-2 shows what happens when you specify HTTP basic authentication.

Figure 40-2 HTTP Basic Authentication



Form-Based Authentication

Form-based authentication allows the developer to control the look and feel of the login authentication screens by customizing the login screen and error pages that an HTTP browser presents to the end user.

When form-based authentication is declared, the following actions occur.

1. A client requests access to a protected resource.

2. If the client is unauthenticated, the server redirects the client to a login page.

3. The client submits the login form to the server.

4. The server attempts to authenticate the user.

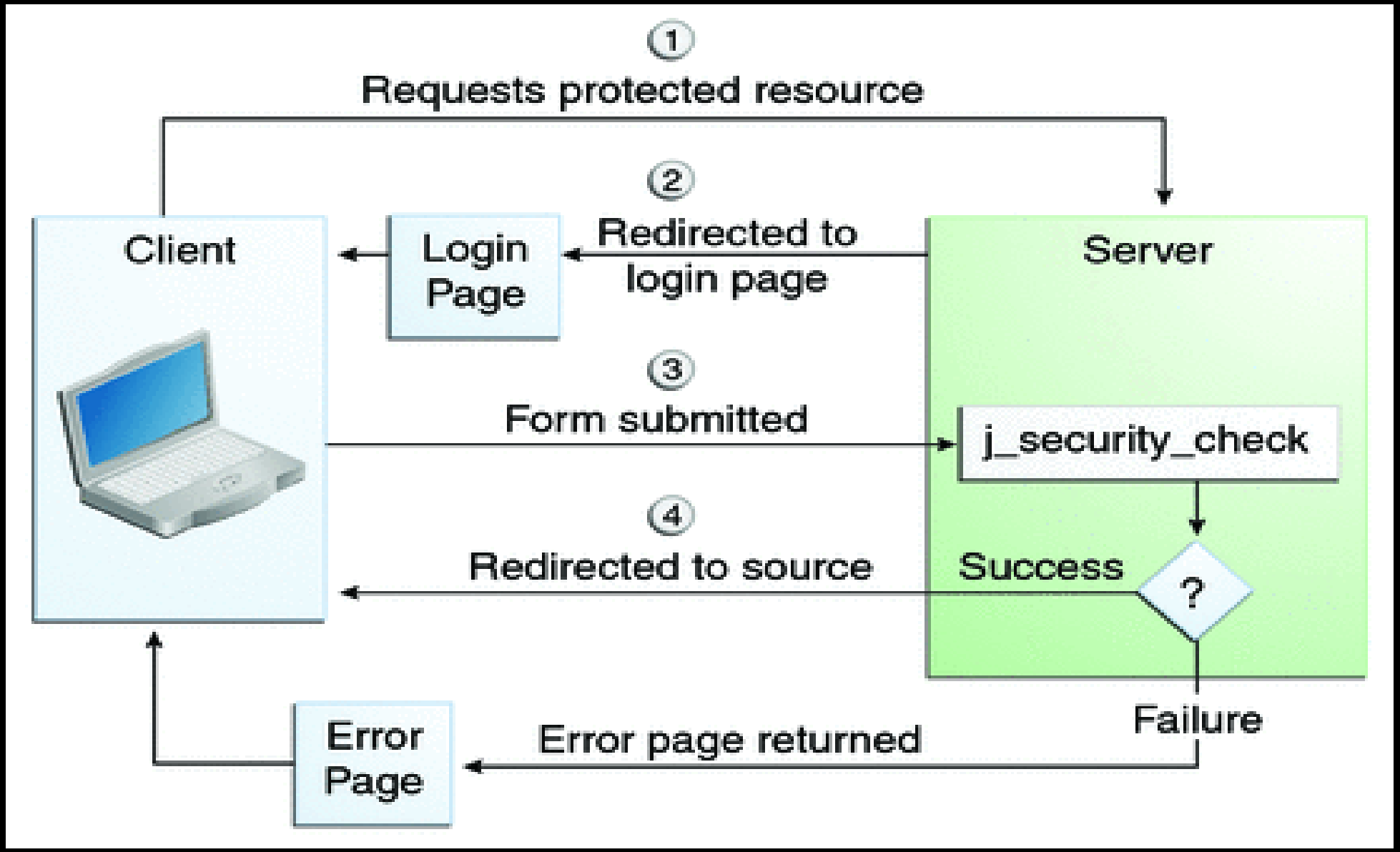
   a. If authentication succeeds, the authenticated user's principal is checked to ensure that it is in a role that is authorized to access the resource.

   If the user is authorized, the server redirects the client to the resource by using the stored URL path.

b. If authentication fails, the client is forwarded or redirected to an error page.

Figure 40-3 shows what happens when you specify form-based authentication.

Figure 40-3 Form-Based Authentication



The section Example: Form-Based Authentication with a JavaServer Faces Application is an example application that uses form-based authentication.

When you create a form-based login, be sure to maintain sessions using cookies or SSL session information.

For authentication to proceed appropriately, the action of the login form must always be `j_security_check`.

This restriction is made so that the login form will work no matter which resource it is for and to avoid requiring the server to specify the action field of the outbound form.

The following code snippet shows how the form should be coded into the HTML page:

```
<form method="POST"
action="j_security_check">
<input type="text"
name="j_username">
<input type="password"
name="j_password">
</form>
```

Digest Authentication

Like basic authentication, digest authentication authenticates a user based on a user name and a password.

However, unlike basic authentication, digest authentication does not send user passwords over the network.

Instead, the client sends a one-way cryptographic hash of the password and additional data.

Although passwords are not sent on the wire, digest authentication requires that clear-text password equivalents be available to the authenticating container so that it can validate received authenticators by calculating the expected digest.

Client Authentication

With client authentication, the web server authenticates the client by using the client's public key certificate.

Client authentication is a more secure method of authentication than either basic or form-based authentication.

It uses HTTP over SSL (HTTPS), in which the server authenticates the client using the client's public key certificate.

SSL technology provides data encryption, server authentication, message integrity, and optional client authentication for a TCP/IP connection.

You can think of a public key certificate as the digital equivalent of a passport.

The certificate is issued by a trusted organization, a certificate authority (CA), and provides identification for the bearer.

Before using client authentication, make sure the client has a valid public key certificate.

For more information on creating and using public key certificates, read Working with Digital Certificates.

Mutual Authentication

With mutual authentication, the server and the client authenticate each other.

Mutual authentication is of two types:

- **Certificate-based (see Figure 40-4)**
- **User name/password-based (see Figure 40-5)**

When using certificate-based mutual authentication, the following actions occur.

1. A client requests access to a protected resource.

2. The web server presents its certificate to the client.

3. The client verifies the server's certificate.

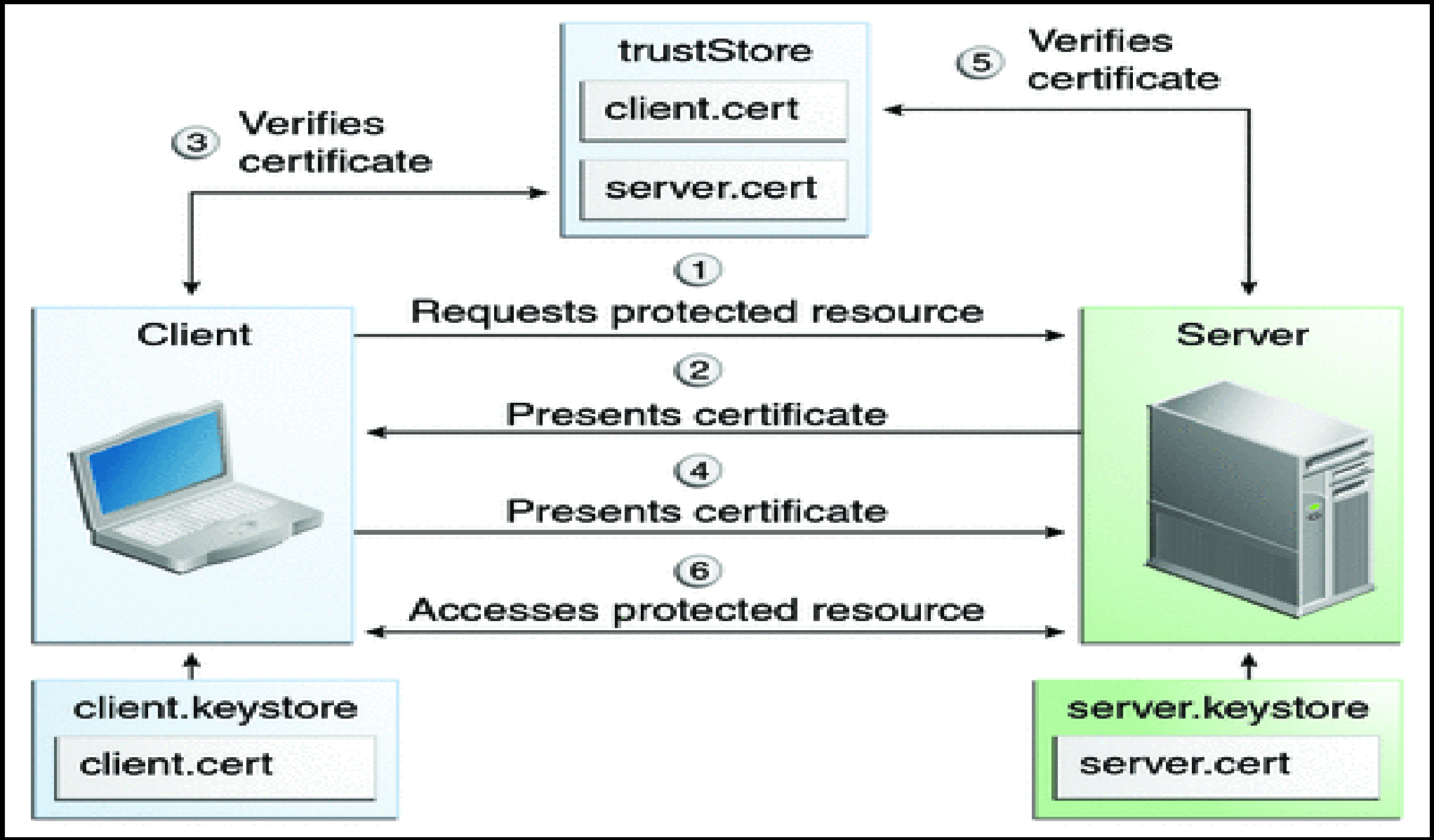
4. If successful, the client sends its certificate to the server.

5. The server verifies the client's credentials.

6. If successful, the server grants access to the protected resource requested by the client.

Figure 40-4 shows what occurs during certificate-based mutual authentication.

Figure 40-4 Certificate-Based Mutual Authentication



In user name/password-based mutual authentication, the following actions occur.

1. A client requests access to a protected resource.

2. The web server presents its certificate to the client.

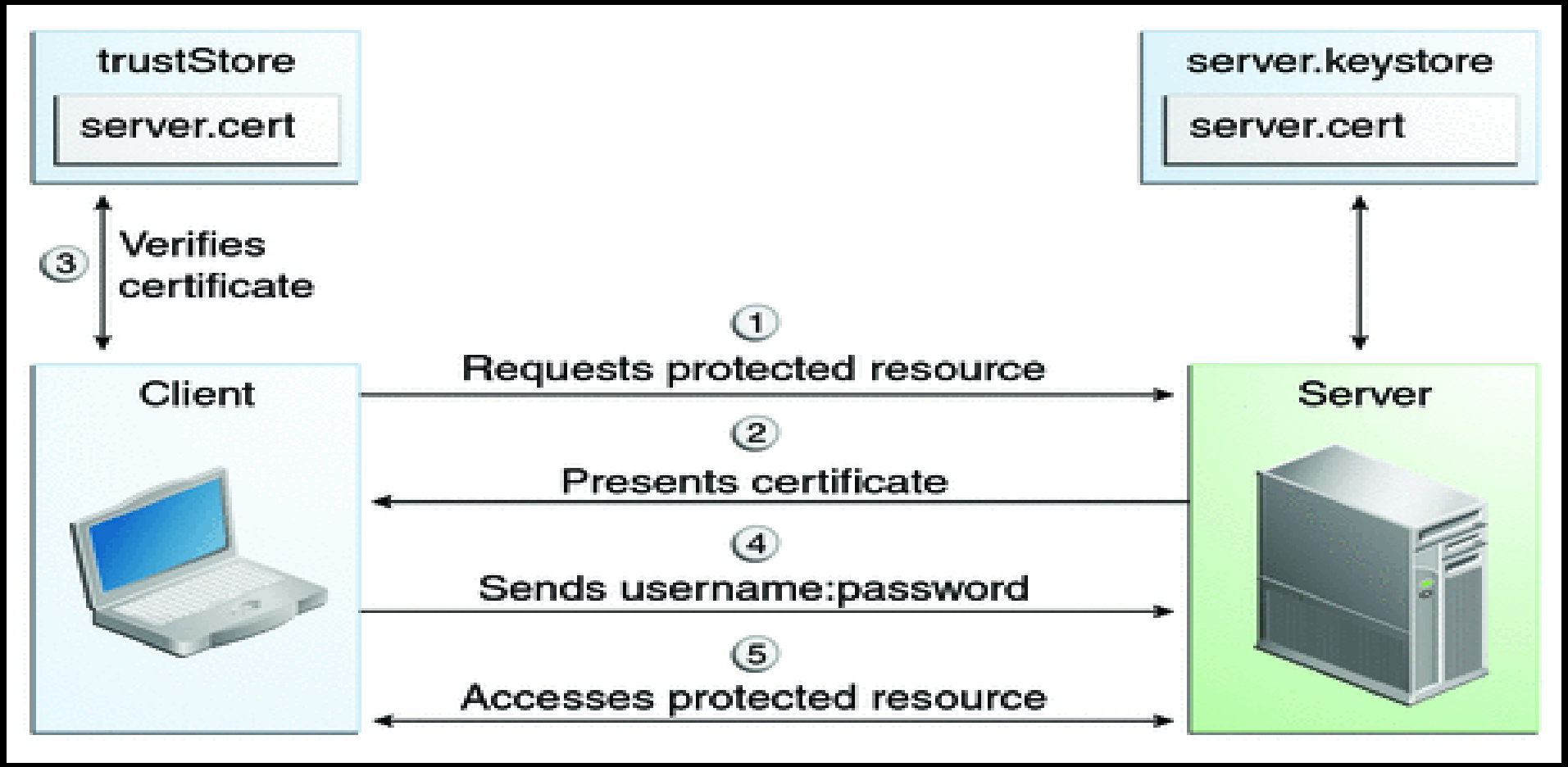
3. The client verifies the server's certificate.

4. If successful, the client sends its user name and password to the server, which verifies the client's credentials.

5. If the verification is successful, the server grants access to the protected resource requested by the client.

Figure 40-5 shows what occurs during user name/password-based mutual authentication.

Figure 40-5 User Name/Password-Based Mutual Authentication



Specifying an Authentication Mechanism in the Deployment Descriptor

To specify an authentication mechanism, use the `login-config` element.

It can contain the following subelements.

- The `auth-method` subelement configures the authentication mechanism for the web application.

  The element content must be either `NONE`, `BASIC`, `DIGEST`, `FORM`, or `CLIENT-CERT`.

- The `realm-name` subelement indicates the realm name to use when the basic authentication scheme is chosen for the web application.

- The `form-login-config` subelement specifies the login and error pages that should be used when form-based login is specified.

Note - Another way to specify form-based authentication is to use the `authenticate`, `login`, and `logout` methods of `HttpServletRequest`, as discussed in Authenticating Users Programmatically.

When you try to access a web resource that is constrained by a `security-constraint` element, the web container activates the authentication mechanism that has been configured for that resource.

The authentication mechanism specifies how the user will be prompted to log in.

If the `login-config` element is present and the `auth-method` element contains a value other than `NONE`, the user must be authenticated to access the resource.

If you do not specify an authentication mechanism, authentication of the user is not required.

The following example shows how to declare form-based authentication in your deployment descriptor:

```
<login-config>
<auth-method>FORM</auth-method>
<realm-name>file</realm-name>
<form-login-config>
<form-login-page>
/login.xhtml
</form-login-page>
```

```
<form-error-page>
/error.xhtml
</form-error-page>
</form-login-config>
</login-config>
```

The login and error page locations are specified relative to the location of the deployment descriptor.

Examples of login and error pages are shown in Creating the Login Form and the Error Page.

The following example shows how to declare digest authentication in your deployment descriptor:

```
<login-config>
<auth-method>DIGEST</auth-method>
</login-config>
```

The following example shows how to declare client authentication in your deployment descriptor:

```
<login-config>
<auth-method>
CLIENT-CERT
</auth-method>
</login-config>
```

Declaring Security Roles

You can declare security role names used in web applications by using the `security-role` element of the deployment descriptor.

Use this element to list all the security roles that you have referenced in your application.

The following snippet of a deployment descriptor declares the roles that will be used in an application using the `security-role` element and specifies which of these roles is authorized to access protected resources using

the `auth-constraint` element:

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

```
<web-resource-name>
Protected Area
</web-resource-name>
<url-pattern>
/security/protected/*
</url-pattern>
<http-method>PUT</http-method>
<http-method>DELETE</http-method>
<http-method>GET</http-method>
<http-method>POST</http-method>
</web-resource-collection>
```

```
<auth-constraint>
<role-name>manager</role-name>
</auth-constraint>
</security-constraint>
<!-- Security roles used by this
web application -->
<security-role>
<role-name>manager</role-name>
</security-role>
<security-role>
<role-name>employee</role-name>
```

```
</security-role>
```

In this example, the `security-role` element lists all the security roles used in the application: `manager` and `employee`.

This enables the deployer to map all the roles defined in the application to users and groups defined on the GlassFish Server.

The `auth-constraint` element specifies the role, `manager`, that can access the HTTP methods `PUT`, `DELETE`, `GET`, `POST` located in the directory specified by the `url-pattern` element (`/jsp/security/protected/*`).

The `@ServletSecurity` annotation cannot be used in this situation because its constraints apply to all URL patterns specified by the `@WebServlet` annotation.

Using Programmatic Security with Web Applications

Programmatic security is used by security-aware applications when declarative security alone is not sufficient to express the security model of the application.

Authenticating Users Programmatically

Servlet 3.0 specifies the following methods of the `HttpServletRequest` interface that enable you to authenticate users for a web application programmatically:

- `authenticate`, which allows an application to instigate authentication of the request caller by the container from within an unconstrained request context.

  A login dialog box displays and collects the user name and password for authentication purposes.

- `login`, which allows an application to collect username and password information as an alternative to specifying form-based authentication in an application deployment descriptor.

- `logout`, which allows an application to reset the caller identity of a request.

The following example code shows how to use the `login` and `logout` methods:

```
package test;
import java.io.IOException;
import java.io.PrintWriter;
import java.math.BigDecimal;
import javax.ejb.EJB;
import
javax.servlet.ServletException;
```

```
import javax.servlet.annotation.
WebServlet;
import
javax.servlet.http.HttpServlet;
import javax.servlet.http.
HttpServletRequest;
import javax.servlet.http.
HttpServletResponse;
@WebServlet(name="TutorialServlet",
urlPatterns={"/TutorialServlet"})
```

```
public class TutorialServlet
extends HttpServlet {
@EJB private
ConverterBean converterBean;
/**
 * Processes requests for both
 * HTTP <code>GET</code>
 * and <code>POST</code> methods.
 * @param request servlet request
 * @param response servlet response
 * @throws ServletException if
```

```
 * a servlet-specific error occurs
 * @throws IOException if
 * an I/O error occurs
 */
protected void processRequest
(HttpServletRequest request,
HttpServletResponse response)
throws ServletException,
IOException{
response.setContentType
("text/html;charset=UTF-8");
```

```
PrintWriter out =
response.getWriter();
try {
out.println("<html>");
out.println("<head>");
out.println("<title>Servlet
TutorialServlet</title>");
out.println("</head>");
out.println("<body>");
request.login
("TutorialUser", "TutorialUser");
```

```
BigDecimal result =
converterBean.dollarToYen
(new BigDecimal("1.0"));
out.println("<h1>Servlet
TutorialServlet result of
dollarToYen= "
+ result + "</h1>");
out.println("</body>");
out.println("</html>");
} catch (Exception e)
{ throw new ServletException(e); }
```

```
finally
{  request.logout();  out.close();  }
} }
```

The following example code shows how to use the `authenticate` method:

```
package com.sam.test;
import java.io.*;
import javax.servlet.*;
```

```
import javax.servlet.http.*;
public class TestServlet extends
HttpServlet{
protected void processRequest
(HttpServletRequest request,
HttpServletResponse response)
throws ServletException,
IOException {
response.setContentType
("text/html;charset=UTF-8");
```

```
PrintWriter out =
response.getWriter();
try {
request.authenticate(response);
out.println
("Authenticate Successful");
} finally { out.close(); }
}
```

Checking Caller Identity Programmatically

In general, security management should be enforced by the container in a manner that is transparent to the web component.

The security API described in this section should be used only in the less frequent situations in which the web component methods need to access the security context information.

Servlet 3.0 specifies the following methods that enable you to access security information about the component's caller:

- `getRemoteUser`, which determines the user name with which the client authenticated.

  The `getRemoteUser` method returns the name of the remote user (the caller) associated by the container with the request.

If no user has been authenticated, this method returns `null`.

- `isUserInRole`, which determines whether a remote user is in a specific security role.

  If no user has been authenticated, this method returns `false`.

This method expects a `String` user `role-name` parameter.

The `security-role-ref` element should be declared in the deployment descriptor with a `role-name` subelement containing the role name to be passed to the method.

Using security role references is discussed in Declaring and Linking Role References.

- `getUserPrincipal`, which determines the principal name of the current user and returns a `java.security.Principal` object.

If no user has been authenticated, this method returns `null`.

Calling the `getName` method on the `Principal` returned by `getUserPrincipal` returns the name of the remote user.

Your application can make business-logic decisions based on the information obtained using these APIs.

Example Code for Programmatic Security

The following code demonstrates the use of programmatic security for the purposes of programmatic login.

This servlet does the following:

1. It displays information about the current user.

2. It prompts the user to log in.

3. It prints out the information again to demonstrate the effect of the `login` method.

4. It logs the user out.

5. It prints out the information again to demonstrate the effect of the `logout` method.

```
package
enterprise.programmatic_login;
import java.io.*;
import java.net.*;
import javax.annotation.security.
DeclareRoles;
import javax.servlet.*;
import javax.servlet.http.*;
```

```
@DeclareRoles("javaee6user")
public class LoginServlet extends
HttpServlet {
/**
 * Processes requests for both
 * HTTP GET and POST methods.
 * @param request servlet request
 * @param response servlet response
 */
```

```
protected void processRequest
(HttpServletRequest request,
HttpServletResponse response)
throws ServletException,
IOException {
response.setContentType
("text/html;charset=UTF-8");
PrintWriter out =
response.getWriter();
```

```
try{
String userName =
request.getParameter("txtUserName");
String password =
request.getParameter("txtPassword");
out.println
("Before Login" + "<br><br>");
out.println("IsUserInRole?.."
+ request.isUserInRole
("javaee6user")+"<br>");
```

```
out.println("getRemoteUser?.." +
request.getRemoteUser()+"<br>");
out.println("getUserPrincipal?.."
+
request.getUserPrincipal()+"<br>");
out.println("getAuthType?.." +
request.getAuthType()+"<br><br>");
try {
request.login(userName, password);
}catch(ServletException ex){
```

```
out.println(
"Login Failed with a ServletException.."
+ ex.getMessage());
return;
}
out.println
("After Login..."+"<br><br>");
out.println("IsUserInRole?.."  +
request.isUserInRole("javaee6user")
+"<br>");
```

```
out.println("getRemoteUser?.." +
request.getRemoteUser()+"<br>");
out.println("getUserPrincipal?.." +
request.getUserPrincipal()+"<br>");
out.println("getAuthType?.." +
request.getAuthType()+"<br><br>");
request.logout();
out.println("After
Logout..."+"<br><br>");
```

```
out.println("IsUserInRole?.." +
request.isUserInRole("javaee6user")
+"<br>");
out.println("getRemoteUser?.." +
request.getRemoteUser()+"<br>");
out.println("getUserPrincipal?.."+
request.getUserPrincipal()+"<br>");
out.println("getAuthType?.." +
request.getAuthType()+"<br>");
} finally { out.close(); }
}  ... }
```

Declaring and Linking Role References

A **security role reference** defines a mapping between the name of a role that is called **from** a web component using `isUserInRole(String role)` and the name of a security role that has been defined for the application.

If no `security-role-ref` element is declared in a deployment descriptor and the `isUserInRole` method is called, the container defaults to checking the provided role name against the list of all security roles defined for the web application.

Using the default method instead of using the `security-role-ref` element limits your flexibility to change role names in an application without also recompiling the servlet making the call.

The `security-role-ref` element is used when an application uses the `HttpServletRequest`.

`isUserInRole(String role).`

The value passed to the `isUserInRole` method is a `String` representing the role name of the user.

The value of the `role-name` element must be the `String` used as the parameter to the `HttpServletRequest`.

```
isUserInRole(String role).
```

The `role-link` must contain the name of one of the security roles defined in the `security-role` elements.

The container uses the mapping of `security-role-ref` to `security-role` when determining the return value of the call.

For example, to map the security role reference `cust` to the security role with role name `bankCustomer`, the syntax would be:

```
<servlet>...
<security-role-ref>
<role-name>cust</role-name>
<role-link>bankCustomer</role-link>
</security-role-ref>...
</servlet>
```

If the servlet method is called by a user in the `bankCustomer` security role, `isUserInRole("cust")` returns `true`.

The `role-link` element in

the `security-role-ref` element must match a `role-name` defined in the `security-role` element of the same `web.xml` deployment descriptor, as shown here:

```
<security-role>
<role-name>bankCustomer</role-name>
</security-role>
```

A security role reference, including the name defined by the reference, is scoped to the component whose deployment descriptor contains the `security-role-ref` deployment descriptor element.

Examples: Securing Web Applications

Some basic setup is required before any of the example applications will run correctly.

The examples use annotations, programmatic security, and/or declarative security to demonstrate adding security to existing web applications.

Here are some other locations where you will find examples of securing various types of applications:

- Example: Securing an Enterprise Bean with Declarative Security
- Example: Securing an Enterprise Bean with Programmatic Security
- GlassFish samples: http://glassfish-samples.java.net/

To Set Up Your System for Running the Security Examples

To set up your system for running the security examples, you need to configure a user database that the application can use for authenticating users.

Before continuing, follow these steps.

1. Add an authorized user to the GlassFish Server.

   For the examples in this chapter and in Chapter 41, Getting Started Securing Enterprise Applications, add a user to the `file` realm of the GlassFish Server, and assign the user to the group `TutorialUser`:

a. From the Administration Console, expand the Configurations node, then expand the server-config node.

b. Expand the Security node.

c. Expand the Realms node.

d. Select the File node.

e. On the Edit Realm page, click Manage Users.

f. On the File Users page, click New.

g. In the User ID field, type a User ID.

h. In the Group List field, type `TutorialUser`.

i. In the New Password and Confirm New Password fields, type a password.

j. Click OK.

Be sure to write down the user name and password for the user you create so that you can use it for testing the example applications.

Authentication is case sensitive for both the user name and password, so write down the user name and password exactly.

This topic is discussed more in Managing Users and Groups on the GlassFish Server.

2. Set up Default Principal to Role Mapping on the GlassFish Server:

a. From the Administration Console, expand the Configurations node, then expand the server-config node.

b. Select the Security node.

c. Select the Default Principal to Role Mapping Enabled check box.

d. Click Save.

Example: Basic Authentication with a Servlet

This example explains how to use basic authentication with a servlet.

With basic authentication of a servlet, the web browser presents a standard login dialog that is not customizable.

When a user submits his or her name and password, the server determines whether the user name and password are those of an authorized user and sends the requested web resource if the user is authorized to view it.

In general, the following steps are necessary for adding basic authentication to an unsecured servlet, such as the ones described in Chapter 3, Getting Started with Web Applications.

In the example application included with this tutorial, many of these steps have been completed for you and are listed here simply to show what needs to be done should you wish to create a similar application.

The completed version of this example application can be found in the directory

tut-install`/examples/security/hello2_basicauth/`.

1. Follow the steps in To Set Up Your System for Running the Security Examples.

2. Create a web module as described in Chapter 3, Getting Started with Web Applications for the servlet example, `hello2`.

3. Add the appropriate security annotations to the servlet.

The security annotations are described in Specifying Security for Basic Authentication Using Annotations.

4. Build, package, and deploy the web application by following the steps in To Build, Package, and Deploy the Servlet Basic Authentication Example Using NetBeans IDE or To Build, Package, and Deploy the Servlet Basic Authentication Example Using Ant.

5. Run the web application by following the steps described in To Run the Basic Authentication Servlet.

Specifying Security for Basic Authentication Using Annotations

The default authentication mechanism used by the GlassFish Server is basic authentication.

With basic authentication, the GlassFish Server spawns a standard login dialog to collect user name and password data for a protected resource.

Once the user is authenticated, access to the protected resource is permitted. To specify security for a servlet, use the `@ServletSecurity` annotation.

This annotation allows you to specify both specific constraints on HTTP methods and more general constraints that apply to all HTTP methods for which no specific constraint is specified.

Within the `@ServletSecurity` annotation, you can specify the following annotations:

- The `@HttpMethodConstraint` annotation, which applies to a specific HTTP method
- The more general `@HttpConstraint` annotation, which applies to all HTTP methods for which there is no corresponding `@HttpMethodConstraint` annotation

Both the `@HttpMethodConstraint` and `@HttpConstraint` annotations within the `@ServletSecurity` annotation can specify the following:

- A `transportGuarantee` element that specifies the data protection requirements (that is, whether or not SSL/TLS is required) that must be satisfied by the connections on which requests arrive.

Valid values for this element are `NONE` and `CONFIDENTIAL`.

- A `rolesAllowed` element that specifies the names of the authorized roles.

For the `hello2_basicauth` application, the `GreetingServlet` has the following annotations:

```
@WebServlet(
name = "GreetingServlet",
urlPatterns = {"/greeting"})
@ServletSecurity(
@HttpConstraint(transportGuarantee
= TransportGuarantee.CONFIDENTIAL,
rolesAllowed = {"TutorialUser"}))
```

These annotations specify that the request URI `/greeting` can be accessed only by users who have been authorized to access this URL because they have been verified to be in the role `TutorialUser`.

The data will be sent over a protected transport in order to keep the user name and password data from being read in transit.

If you use the `@ServletSecurity` annotation, you do not need to specify security settings in the deployment descriptor.

Use the deployment descriptor to specify settings for nondefault authentication mechanisms, for which you cannot use the `@ServletSecurity` annotation.

To Build, Package, and Deploy the Servlet Basic Authentication Example Using NetBeans IDE

1. Follow the steps in To Set Up Your System for Running the Security Examples.

2. In NetBeans IDE, from the File menu, choose Open Project.

3. In the Open Project dialog, navigate to:

   `tut-install/examples/security`

4. Select the `hello2_basicauth` folder.

5. Select the Open as Main Project check box.

6. Click Open Project.

7. Right-click `hello2_basicauth` in the Projects pane and select Deploy.

This option builds and deploys the example application to your GlassFish Server instance.

To Build, Package, and Deploy the Servlet Basic Authentication Example Using Ant

1. **Follow the steps in To Set Up Your System for Running the Security Examples.**

2. In a terminal window, go to:

```
tut-install/examples/security/
hello2_basicauth/
```

3. Type the following command:

```
ant
```

This command calls the `default` target, which builds and packages the application into a WAR file, `hello2_basicauth.war`, that is located in the `dist` directory.

4. Make sure that the GlassFish Server is started.

5. To deploy the application, type the following command:

```
ant deploy
```

To Run the Basic Authentication Servlet

1. In a web browser, navigate to the following URL:

```
https://localhost:8181/
hello2_basicauth/greeting
```

You may be prompted to accept the security certificate for the server.

If so, accept the security certificate.

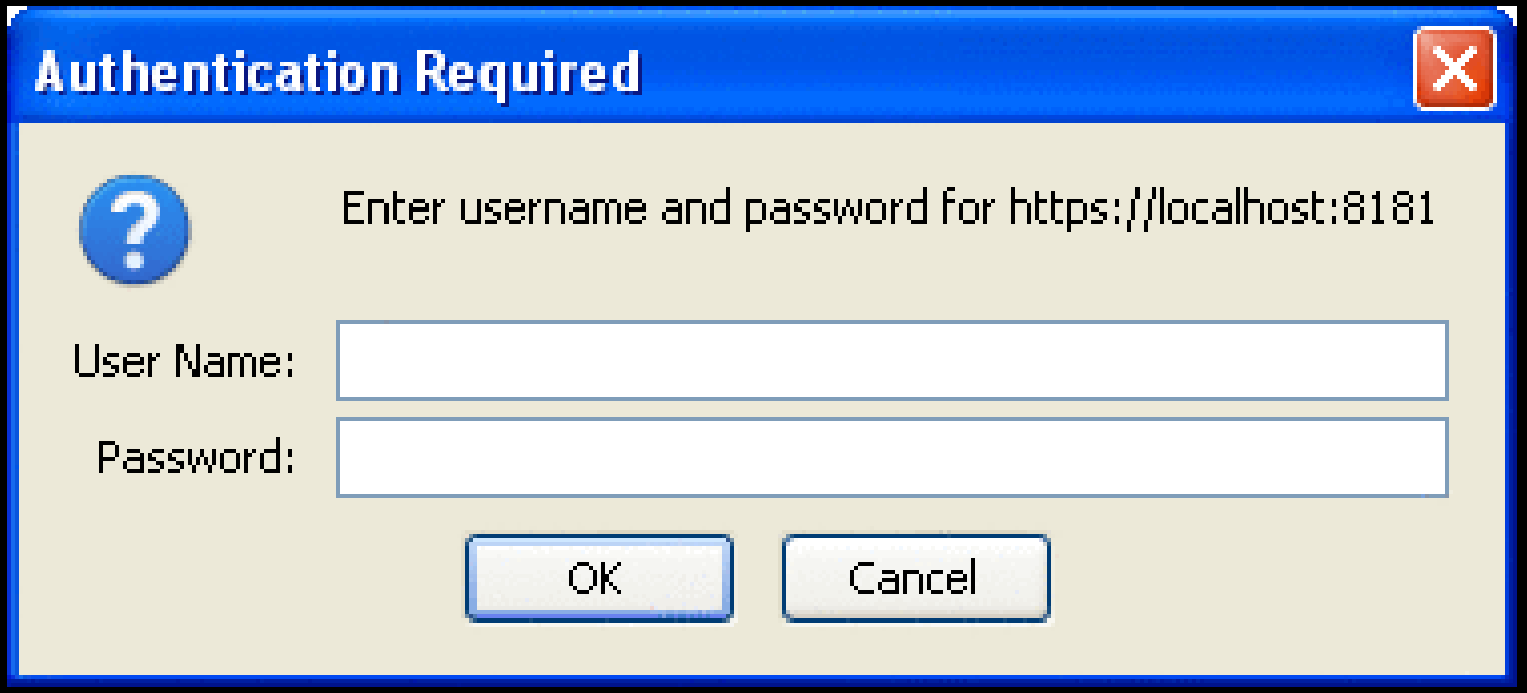
If the browser warns that the certificate is invalid because it is self-signed, add a security exception for the application.

An Authentication Required dialog box appears.

Its appearance varies, depending on the browser you use.

Figure 40-6 shows an example.

Figure 40-6 Sample Basic Authentication Dialog Box



2. Type a user name and password combination that corresponds to a user who has already been created in the `file` realm of the GlassFish Server and has been assigned to the group of `TutorialUser`; then click OK.

Basic authentication is case sensitive for both the user name and password, so type the user name and password exactly as defined for the GlassFish Server.

The server returns the requested resource if all the following conditions are met.

- A user with the user name you entered is defined for the GlassFish Server.
- The user with the user name you entered has the password you entered.

- The user name and password combination you entered is assigned to the group `TutorialUser` on the GlassFish Server.

- The role of `TutorialUser`, as defined for the application, is mapped to the group `TutorialUser`, as defined for the GlassFish Server.

When these conditions are met and the server has authenticated the user, the application will appear as shown in Figure 3-2 but with a different URL.

3. Type a name in the text field and click the Submit button.

Because you have already been authorized, the name you enter in this step does not have any limitations.

You have unlimited access to the application now.

The application responds by saying “Hello” to you, as shown in Figure 3-3 but with a different URL.

Next Steps

For repetitive testing of this example, you may need to close and reopen your browser.

You should also run the `ant undeploy` and `ant clean` targets or the NetBeans IDE Clean and Build option to get a fresh start.

Example: Form-Based Authentication with a JavaServer Faces Application

This example explains how to use form-based authentication with a JavaServer Faces application.

With form-based authentication, you can customize the login screen and error pages that are presented to the web client for authentication of the user name and password.

When a user submits his or her name and password, the server determines whether the user name and password are those of an authorized user and, if authorized, sends the requested web resource.

This example, `hello1_formauth`, adds security to the basic JavaServer Faces application shown in Web Modules: The `hello1` Example.

In general, the steps necessary for adding form-based authentication to an unsecured JavaServer Faces application are similar to those described in Example: Basic Authentication with a Servlet.

The major difference is that you must use a deployment descriptor to specify the use of form-based authentication, as described in Specifying Security for the Form-Based Authentication Example.

In addition, you must create a login form page and a login error page, as described in Creating the Login Form and the Error Page.

The completed version of this example application can be found in the directory

tut-install`/examples/security/hello1_formauth/`.

Creating the Login Form and the Error Page

When using form-based login mechanisms, you must specify a page that contains the form you want to use to obtain the user name and password, as well as a page to display if login authentication fails.

This section discusses the login form and the error page used in this example.

Specifying Security for the Form-Based Authentication Example shows how you specify these pages in the deployment descriptor.

The login page can be an HTML page or a servlet, and it must return an HTML page containing a form that conforms to specific naming conventions (see the Java Servlet 3.0 specification for more information on these requirements).

To do this, include the elements that accept user name and password information between <form></form> tags in your login page.

The content of an HTML page or servlet for a login page should be coded as follows:

```
<form method="post"
action="j_security_check">
<input type="text"
name="j_username">
```

```
<input type="password"
name= "j_password">
</form>
```

The full code for the login page used in this example can be found at *tut-install*`/examples/security/hello1_formauth/web/login.xhtml`.

An example of the running login form page is shown later, in Figure 40-7.

Here is the code for this page:

```
<html
xmlns=
"http://www.w3.org/1999/xhtml"
>
<head>
<title>Login Form</title>
```

```
</head>
<body>
<h2>Hello, please log in:</h2>
<form name="loginForm"
method="POST"
action="j_security_check">
<p>
<strong>
Please type your user name:
</strong>
```

```
<input type="text"
name="j_username" size="25">
</p>
<p>
<strong>
Please type your password:
</strong>
<input type="password" size="15"
name="j_password">
</p>
<p>
```

```
<input type="submit"
value="Submit"/>
<input type="reset" value="Reset"/>
</p>
</form>
</body>
</html>
```

The login error page is displayed if the user enters a user name and password combination that is not authorized to access the protected URI.

For this example, the login error page can be found at *tut-install*`/examples/security/hello1_formauth/web/error.xhtml`.

For this example, the login error page explains the reason for receiving the error page and provides a link that will allow the user to try again.

Here is the code for this page:

```
<html xmlns=
"http://www.w3.org/1999/xhtml">
<head>
<title>Login Error</title>
</head>
<body>
<h2>
Invalid user name or password.
</h2>
```

```
<p>Please enter a user name or
password that is authorized to
access this application.

For this application, this means a
user that has been created in the
<code>file</code> realm and has
been assigned to the
  <em>group</em> of
<code>TutorialUser</code>.</p>
```

```
<p>
<a href="login.xhtml">
Return to login page
</a>
</p>
</body>
</html>
```

Specifying Security for

the Form-Based Authentication Example

This example takes a very simple servlet-based web application and adds form-based security.

To specify form-based instead of basic authentication for a JavaServer Faces example, you must use the deployment descriptor.

The following sample code shows the security elements added to the deployment descriptor for this example, which can be found in

tut-install`/examples/security/`

`hello1_formauth/web/`

`WEB-INF/web.xml`.

```
<security-constraint>
<display-name>
Constraint1
</display-name>
<web-resource-collection>
<web-resource-name>
wrcoll
</web-resource-name>
<description/>
<url-pattern>/*</url-pattern>
</web-resource-collection>
```

```
<auth-constraint>
<description/>
<role-name>TutorialUser</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.xhtml
```

```
</form-login-page>
<form-error-page>
/error.xhtml
</form-error-page>
</form-login-config>
</login-config>
<security-role>
<description/>
<role-name>TutorialUser</role-name>
</security-role>
```

To Build, Package, and Deploy the Form-Based Authentication Example Using NetBeans IDE

1. Follow the steps in To Set Up Your System for Running the Security Examples.

2. In NetBeans IDE, from the File menu, choose Open Project.

3. In the Open Project dialog, navigate to:

   `tut-install/examples/security`

4. Select the `hello1_formauth` folder.

5. Select the Open as Main Project check box.

6. Click Open Project.

7. Right-click `hello1_formauth` in the Projects pane and select Deploy.

To Build, Package, and Deploy the Form-Based Authentication Example Using Ant

1. **Follow the steps in To Set Up Your System for Running the Security Examples.**

2. In a terminal window, go to:

```
tut-install/examples/security/
hello2_formauth/
```

3. Type the following command at the terminal window or command prompt:

```
ant
```

This target will spawn any necessary compilations, copy files to the *tut-install*`/examples/security/hello2_formauth/build/` directory, create the WAR file, and copy it to the *tut-install*`/examples/security/hello2_formauth/dist/` directory.

4. To deploy `hello2_formauth.war` to the GlassFish Server, type the following command:

```
ant deploy
```

To Run the Form-Based Authentication Example

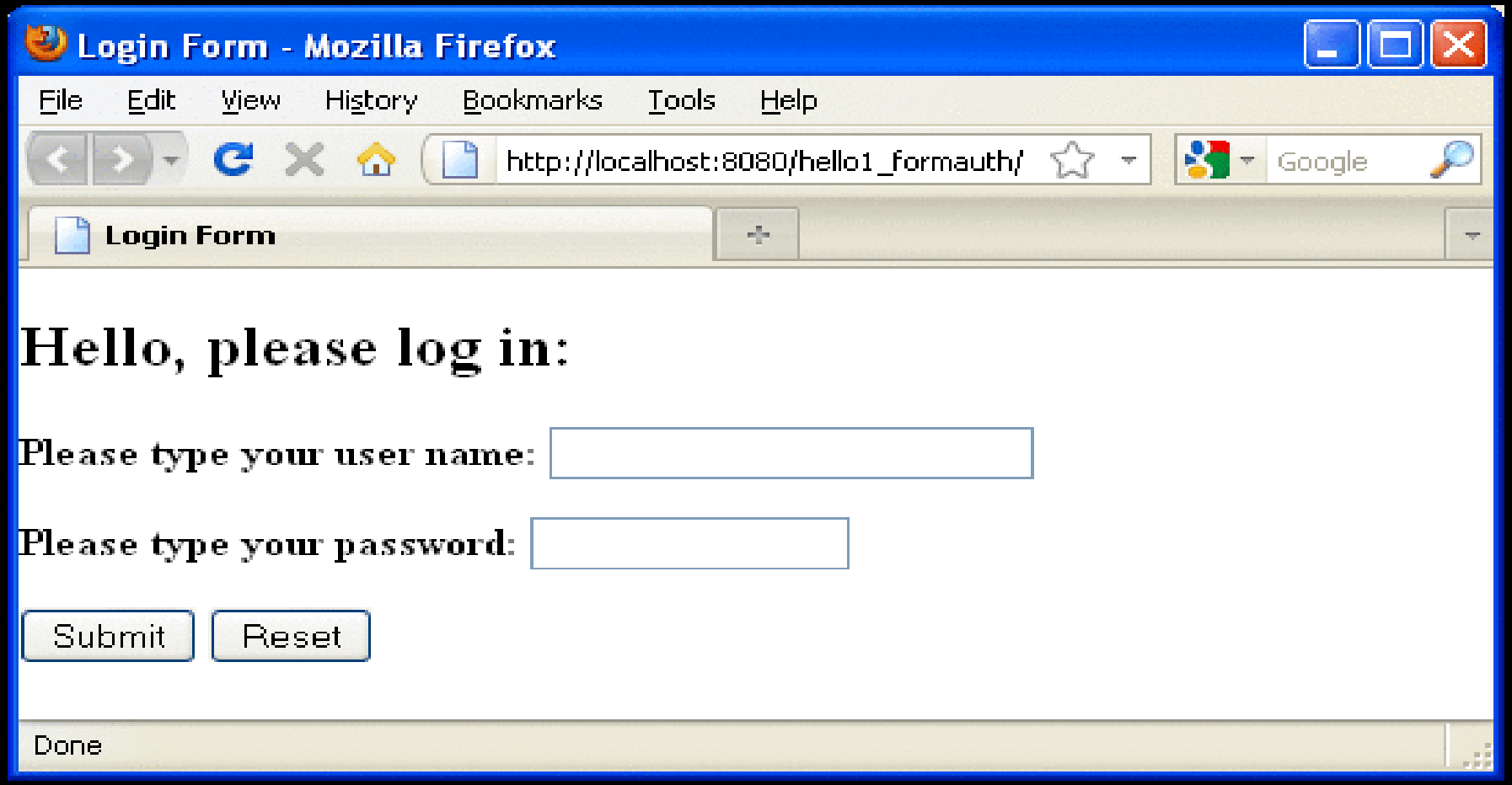
To run the web client for `hello1_formauth`, follow these steps.

1. **Open a web browser to the following URL:**

   `https://localhost:8181/hello1_formauth/`

The login form displays in the browser, as shown in Figure 40-7.

Figure 40-7 Form-Based Login Page



2. Type a user name and password combination that corresponds to a user who has already been created in the `file` realm of the GlassFish Server and has been assigned to the group of `TutorialUser`.

Form-based authentication is case sensitive for both the user name and password, so type the user name and password exactly as defined for the GlassFish Server.

3. Click the Submit button.

If you entered `My_Name` as the name and `My_Pwd` for the password, the server returns the requested resource if all the following conditions are met.

- A user with the user name `My_Name` is defined for the GlassFish Server.

- The user with the user name `My_Name` has a password `My_Pwd` defined for the GlassFish Server.

- The user `My_Name` with the password `My_Pwd` is assigned to the group `TutorialUser` on the GlassFish Server.

- The role `TutorialUser`, as defined for the application, is mapped to the group `TutorialUser`, as defined for the GlassFish Server.

When these conditions are met and the server has authenticated the user, the application appears.

4. Type your name and click the Submit button.

Because you have already been authorized, the name you enter in this step does not have any limitations.

You have unlimited access to the application now.

The application responds by saying "Hello" to you.

Next Steps

For additional testing and to see the login error page generated, close and reopen your browser, type the application URL, and type a user name and password that are not authorized.

Note - For repetitive testing of this example, you may need to close and reopen your browser.

You should also run the `ant clean` and `ant undeploy` commands to ensure a fresh build if using the Ant tool, or select Clean and Build then Deploy if using NetBeans IDE.