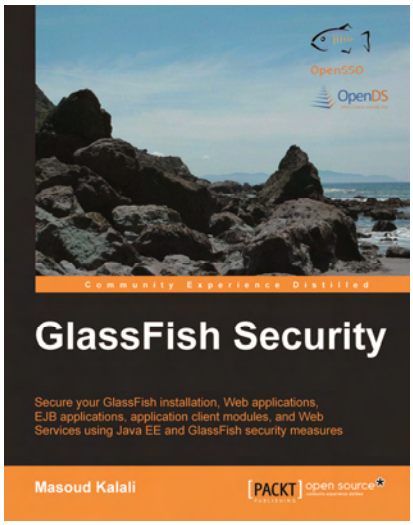
GlassFish Security

Masoud Kalali



Chapter No.3

"Designing and Developing Secure Java EE Applications"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.3 "Designing and Developing Secure Java EE Applications"

A synopsis of the book's content

Information on where to buy this book

About the Author

Masoud Kalali has a software engineering degree and has been working on software development projects since 1998. He has experience with a variety of technologies (.NET, J2EE, CORBA, and COM+) on diverse platforms (Solaris, Linux, and Windows). His experience is in software architecture, design, and server-side development.

Masoud has several articles published in Java.net and DZone, and has authored multiple refcards published by DZone, including Java EE security and GlassFish v3 refcards. He is one of founder members of the NetBeans Dream Team and a GlassFish community spotlighted developer.

Masoud's main areas of research and interest include Service Oriented Architecture and large-scale systems' development and deployment. In his leisure time he enjoys photography, mountaineering and camping.

Masoud blogs on Java EE, Software Architecture and Security at `http://weblogs.java.net/blog/kalali/` and you can follow him at his Twitter account at `http://twitter.com/MasoudKalali`.

Masoud can be reached via `Kalali@gmail.com` in case you had some queries about the book or if you just felt like talking to him about software engineering.

GlassFish Security

We are living in a world full of dazzling wonders, and I for one always enjoy encountering them. Software development is one of the wonders that dazzles me because of its enormously vast domain, including many concerns and subjects of interest. Looking at this domain from any distance, we will see one big and sometimes blurry-edged spot named security.

Security, an orthogonal and inseparable part of software systems, is not for preventing others from accessing some information and system resources but for allowing them access in an appropriate way, by implementing necessary means to precisely check any attempt to access a resource and either allow it to go further or not and record all information related to examining this attempt for further review.

Java EE is the platform of choice for developing enormously large-scale applications, and provides plethora of features for implementing security plans for applications, starting from dealing with identity storages and identity solutions up to providing GUI-level support for security concerns and integration with other security providers.

Nowadays, integration is something that we hear in every software development meeting and session independent from what the session is about. Security integration, however, is a delicate matter compared to all other issues as it deals directly with the organization's assets. Java EE design allows it to delegate its security requirements to another entity in the enterprise, like a single sign-on solution, which on the other hand can integrate with other products and platforms in use in the organization.

The GlassFish Security book is an attempt to explain this domain considering Java EE, GlassFish, and OpenSSO capabilities and features.

What This Book Covers

Chapter 1, *Java EE Security Model*, discusses how we can secure different Java applications by using the declarative security model or by using the API exposed by Java EE containers to access the security enforcement layers programmatically. It also briefl y introduces Web modules, EJB modules, and application client module's security in different levels, including authentication, authorization, and transport security.

Chapter 2, *GlassFish Security Realms*, discusses JAAS and GlassFish security realm, including File realm, JDBC realm, LDAP realm, and Certificate realm in detail as that will be required to develop a secure enterprise application. It also discusses GlassFish application server interaction with identity storages such as relational databases**,** Lightweight Directory Access Protocol (LDAP) servers, fl at fi le storage, and so on.

Chapter 3, *Designing and Developing Secure Java EE Applications*, covers developing and deploying a secure Java EE application with all standard modules including Web,

EJB, and application client modules. It also teaches us how we can secure EJBs using annotation and then use a web frontend to use the secured EJBs after a user provides correct identification information.

Chapter 4, *Securing GlassFish Environment*, helps you secure your operating system and environment from unprivileged access by applications deployed in GlassFish using the OS features and Java policy management. It also covers network communication security, GlassFish password security, and finally security auditing, which is a complementary function in software security.

Chapter 5, *Securing GlassFish*, covers GlassFish administration security tasks such as password security and listener security. This chapter will teach you to secure GlassFish by examining the administration security, password protection, and network listener security. It also discusses the benefits of virtual servers for isolating different applications deployed in a single machine with a single IP address.

Chapter 6, *Introducing OpenDS: Open Source Directory Service,* teaches you about directory service and the set of features OpenDS provides—installing, administrating, and monitoring OpenDS and using OpenDS in embedded mode. This chapter teaches you to set up a replication topology to ensure service and data availability in case of unpredicted disasters.

Chapter 7, *OpenSSO, the Single sign-on Solution,* covers projects security from an integration point of view. In this chapter you will install and configure OpenSSO and understand different methods of using OpenSSO. It also teaches you how to use OpenSSO RESTful Web Services for authentication, authorization, and acquiring SSO tokens.

Chapter 8, *Securing Java EE Applications using OpenSSO,* covers OpenSSO Policy Agents that let us as architects, system designers, and developers secure a Java EE application using OpenSSO without changing the application source code. It also discusses about Policy Agents, Policy Agent's installation, and administration, along with changing our sample application to place it under agent protection instead of using plain Java EE protection.

Chapter 9, *Securing Web Services by OpenSSO,* covers Web Services security and how we can use OpenSSO and OpenSSO agents to secure our Web Services deployed in GlassFish. It also teaches you to install OpenSSO Web Services Security Provider Agent and develop a simple, secure pair of WSP and WSC.

3

Designing and Developing Secure Java EE Applications

In previous chapters we discussed how we can utilize Java EE capabilities to secure our Java EE applications. In this chapter, we are going to put what we learned into practice and develop a secure Java EE application with all standard modules including Web, EJB, and application client modules.

Security is an orthogonal concern for an application and we should assess it right from the start by reviewing the analysis we receive from business and functional analysts. Assessing the security requirements results in understanding the functionalities we need to include in our architecture to deliver a secure application covering the necessary requirements.

Security necessities can include a wide area of requirements, which may vary from a simple authentication to several sub-systems. A list of these sub-systems includes identity and access management system and transport security, which can include encrypting data as well.

In this chapter we will develop a secure Java EE application based on Java EE and GlassFish capabilities. In course of the chapter we will cover following topics:

- Analyzing Java EE application security requirements
- Including security requirements in Java EE application design
- Developing secure Business layer using EJBs
- Developing secure Presentation layer using JSP and Servlets
- Configuring deployment descriptors of Java EE applications
- Specifying security realm for enterprise applications
- Developing secure application client module
- Configuring Application Client Container

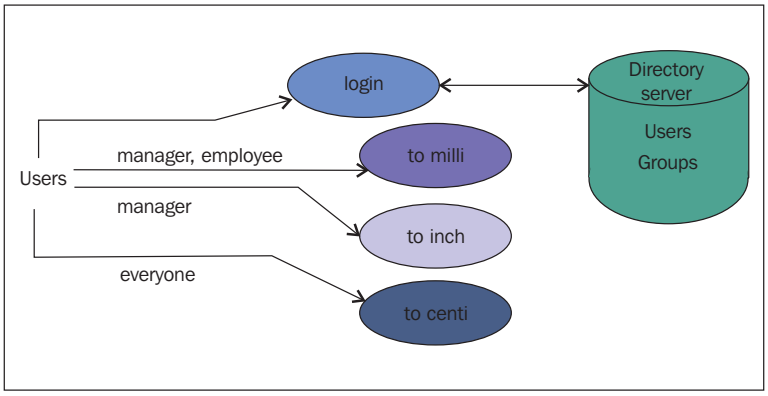
Understanding the sample application

The sample application that we are going to develop, converts different length measurement units into each other. Our application converts meter to centimeter, millimeter, and inch. The application also stores usage statistics for later use cases.

Guest users who prefer not to log in can only use meter to centimeter conversion, while any company employee can use meter to centimeter and meter to millimeter conversion, and finally any of company's managers can access meter to inch in addition to two other conversion functionalities. We should show a custom login page to comply with site-wide look and feel.

No encryption is required for communication between clients and our application but we need to make sure that no one can intercept and steal the username and passwords provided by members. All members' identification information is stored in the company's wide directory server.

The following diagram shows the high-level functionality of the sample application:



We have login action and three conversion actions. Users can access some of them after logging in and some of them can be accessed without logging in.

Analyzing sample application business logic

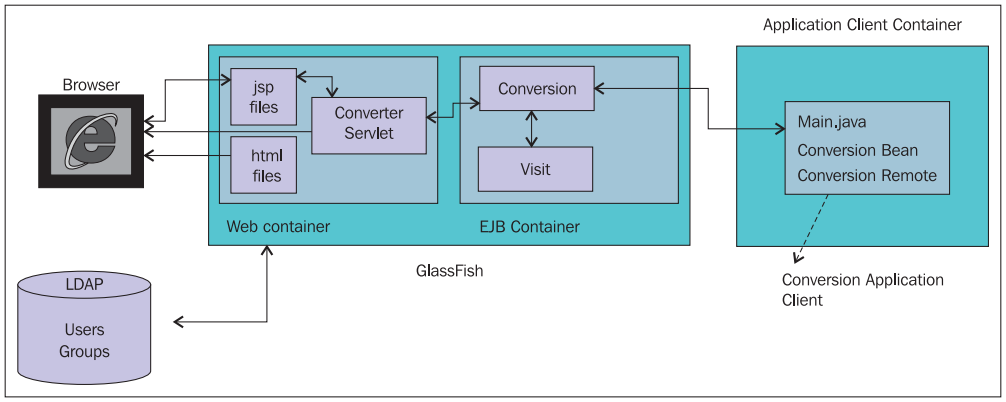
Before looking at security requirements and factors affecting the software security let's see what we need to provide in our business layer. Our business logic consists of conversion operations and persistence of the conversion operations usage statistics. We can use a stateless Session Bean with three methods, one for each type of conversion. And for statistics persistence we can use EJB 3 entity beans.

After studying the application description we can extract the following security-related requirements which we need to address to comply with the application description:

- Authentication is required
- Authentication should happen over a secure channel
- Authorization is required
- We need to use LDAP security realm

So far we translated the business analysis to technical requirements and now we are going to check each requirement in further detail to extract the implementation details. For implementing the sample application we can use a simple **bottom-up** procedure.

The following diagram shows the application blocks down to JSP files, Servlet, and EJBs.



As you can see we have Web module, EJB module, and an application client module. The Web module and the application client module presents a frontend for the EJB layer that performs both business logic, which is the conversion operations, and storing the conversion operation invocation statistics using Entity Beans. GlassFish uses the LDAP realm to authenticate the users against the specified directory server.

Implementing the Business and Persistence layers

The Persistence layer consists of an Entity Bean named `Visit`; we use this entity bean to store information about each visit. We will use a session bean with three business methods to convert a given length in meter to centimeter, millimeter, and inch.

Implementing the Persistence layer

We are using EJB 3 to develop the Persistence layer so we will only need to implement the entity bean and define the persistence unit. The following listing shows the `Visit` class.

Complete code for this class is available in the book's source code: `https://www.packtpub.com//sites/default/files/downloads/9386_Code.zip`.

```
@Entity
public class Visit implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Temporal(javax.persistence.TemporalType.DATE)
    private Date visitDate;
    private String username;
    private String operationName;
    private int conversionValue;
        public Visit() {
    }

    public Visit(Date visitDate, String username, String Operation,
      int conversionValue) {
        this.visitDate = visitDate;
        this.username = username;
        this.operationName = Operation;
        this.conversionValue = conversionValue;
    }
}
```

Now that our entity bean is ready we can start looking at our session bean that drives the application business logic and also stores information about each invocation using the `Visit` entity bean. The following listing shows Conversion session bean local interface.

```
@Local
public interface ConversionLocal {
    float toInch(int meter);
    int toCentimeter(int meter);
    int toMillimeter(int meter);
}
```

All of these methods are implemented in Conversion bean implementation which is as follows:

```
@Stateless
public class ConversionBean implements ConversionLocal {
    @PersistenceContext(unitName = "chapter3")
    private EntityManager em;
    @Resource
    private SessionContext ctx;

    @RolesAllowed({"manager_role"})
    public float toInch(int meter) {
        persist(meter, "toInch");
        return Math.round(meter * 39.37);
    }

    @PermitAll
    public int toCentimeter(int meter) {
        persist(meter, "toCentimeter");
        return meter * 100;
    }

    @RolesAllowed("employee_role")
    public int toMillimeter(int meter) {
        persist(meter, "toInch");
        return meter * 1000;
    }

    private void persist(int value, String operationName) {
        String userName = ctx.getCallerPrincipal().getName();
        Visit v = new Visit(new Date(), userName, operationName,
          value);
        em.persist(v);
    }
}
```

Starting from the first line we are using `@Stateless` to mark this class as a stateless Session Bean. Later on we are using `@PersistenceContext` to inject an entity manager into the instance. We will use this entity manager to store `Visit` entities. Then we are using `@Resource` to inject the current `SessionContext` into the session bean. Later on we will use it to extract the current principal and username of the invoker. The first security-related annotation is `@RolesAllowed({"manager"})`, which instructs the application server to only permit an authenticated user with `manager` role to invoke this method. After this we have `@PermitAll` which instructs the application server to allow anyone, either authenticated or not, to invoke this method. And finally we are using `@RolesAllowed("employee")` to instruct the application server that any authenticated user with `employee` role can invoke this method.

The `persist` method stores the invocation information. This information includes the current invoker username, which we extract from `SessionContext` using the `getCallerPrincipal().getName()` method.

Finally we have a persistence unit that uses `sample` data source and sample database which is bundled with GlassFish. The listing shown below contains a snippet of `persistence.xml` file, which configures a persistence unit for `chapter3`.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
    <persistence-unit name="chapter3" transaction-type="JTA">
        <provider>oracle.toplink.essentials.PersistenceProvider
          </provider>
        <jta-data-source>jdbc/sample</jta-data-source>
        <class>book.glassfish.security.chapter3.Visit</class>
        <exclude-unlisted-classes>true</exclude-unlisted-classes>
        <properties>
            <property name="toplink.ddl-generation"
              value="drop-and-create-tables"/>
        </properties>
    </persistence-unit>
</persistence>
```

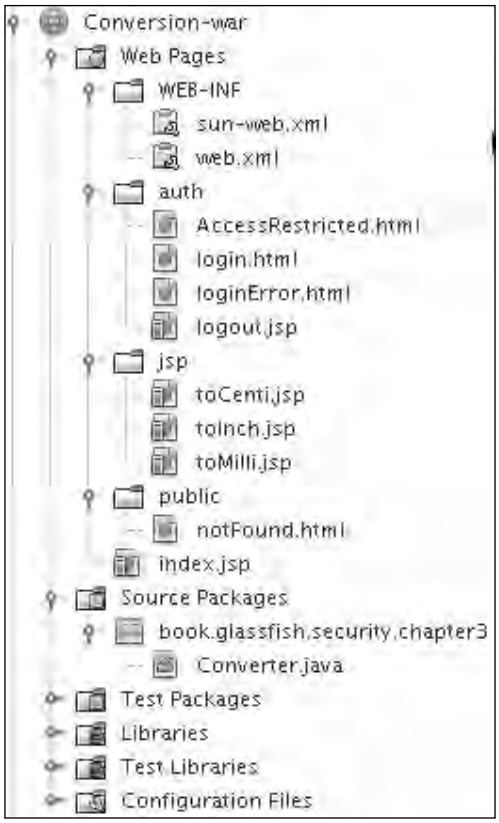
Now that we have our Persistence and Business layers ready we can start looking at the Web layer and how the Web layer can complement the inner layer in securing the system.

Developing the Presentation layer

The Presentation layer is the closest layer to end users when we are developing applications that are meant to be used by humans instead of other applications. In our application, the Presentation layer is a Java EE web application consisting of the elements listed in the following table. In the table you can see that different JSP files are categorized into different directories to make the security description easier.

Element Name	Element Description
`Index.jsp`	Application entry point. It has some links to functional JSP pages like `toMilli.jsp` and so on.
`auth/login.html`	This file presents a custom login page to a user when they try to access a restricted resource. This file is placed inside `auth` directory of the Web application.
`auth/logout.jsp`	Logs users out of the system after their work is finished.
`auth/loginError.html`	Unsuccessful login attempt redirect users to this page. This file is placed inside the `auth` directory of the Web application.
`jsp/toInch.jsp`	Converts given length to inch, it is only available for managers.
`jsp/toMilli.jsp`	Converts given length to millimeter, this page is available to any employee.
`jsp/toCenti.jsp`	Converts given length to centimeter, this functionality is available for everyone.
`Converter Servlet`	Receives the request and invokes the session bean to perform the conversion and returns back the value to the user.
`auth/accessRestricted.html`	An error page for error 401 which happens when authorization fails.
`Deployment Descriptors`	The deployment descriptors which we describe the security constraints over resources we want to protect.

Now that our application building blocks are identified we can start implementing them to complete the application. Before anything else let's implement JSP files that provides the conversion GUI. The directory layout and content of the Web module is shown in the following figure:



Implementing the Conversion GUI

In our application we have an `index.jsp` file that acts as a gateway to the entire system and is shown in the following listing:

```
<html>
    <head><title>Select A conversion</title></head>
    <body><h1>Select A conversion</h1>
        <a href="auth/login.html">Login</a>
        <br/>
        <a href="jsp/toCenti.jsp">Convert Meter to Centimeter</a>
        <br/>
        <a href="jsp/toInch.jsp">Convert Meter to Inch</a>
        <br/>
```

```
            <a href="jsp/toMilli.jsp">Convert to Millimeter</a><br/>
            <a href="auth/logout.jsp">Logout</a>
        </body>
    </html>
```

Implementing the Converter servlet

The `Converter` servlet receives the conversion value and method from JSP files and calls the corresponding method of a session bean to perform the actual conversion. The following listing shows the Converter servlet content:

```
@WebServlet(name="Converter", urlPatterns={"/Converter"})
public class Converter extends HttpServlet {
    @EJB
    private ConversionLocal conversionBean;

    protected void processRequest(HttpServletRequest request,
      HttpServletResponse response)
            throws ServletException, IOException {
    }
    @Override
    protected void doPost(HttpServletRequest request,
      HttpServletResponse response)
          throws ServletException, IOException {
            System.out.println("POST");
            response.setContentType("text/html;charset=UTF-8");
            PrintWriter out = response.getWriter();
            try{
              int valueToconvert =
                Integer.parseInt(request.getParameter("meterValue"));
              String method = request.getParameter("method");
              out.print("<hr/> <center><h2>Conversion Result is: ");
              if (method.equalsIgnoreCase("toMilli")) {

              out.print(conversionBean.toMillimeter(valueToconvert));
              } else if (method.equalsIgnoreCase("toCenti")) {

              out.print(conversionBean.toCentimeter(valueToconvert));
            } else if (method.equalsIgnoreCase("toInch")) {
                out.print(conversionBean.toInch(valueToconvert));
            }
            out.print("</h2></center>");

          }catch (AccessLocalException ale) {
              response.sendError(401);
          }finally {
              out.close();
          }
    }
}
```

Starting from the beginning we are using annotation to configure the servlet mapping and servlet name instead of using the deployment descriptor for it. Then we use dependency injection to inject an instance of `Conversion` session bean into the servlet and decide which one of its methods we should invoke based on the conversion type that the caller JSP sends as a parameter. Finally, we catch `javax.ejb.AccessLocalException` and send an **HTTP 401 error** back to inform the client that it does not have the required privileges to perform the requested action. The following figure shows what the result of invocation could look like:

Each servlet needs some description elements in the deployment descriptor or included as deployment descriptor elements.

Implementing the conversion JSP files is the last step in implementing the functional pieces. In the following listing you can see content of the `toMilli.jsp` file.

```
<html>
    <head><title>Convert To Millimeter</title></head>
    <body><h1>Convert To Millimeter</h1>
        <form method=POST action="../Converter">Enter Value to
          Convert: <input name=meterValue>
            <input type="hidden" name="method" value="toMilli">
              <input type="submit" value="Submit" />
        </form>
    </body>
</html>
```

The `toCenti.jsp` and `toInch.jsp` files look the same except for the descriptive content and the value of the hidden parameter which will be `toCenti` and `toInch` respectively for `toCenti.jsp` and `toInch.jsp`.

Now we are finished with the functional parts of the Web layer; we just need to implement the required GUI for security measures.

Implementing the authentication frontend

For the authentication, we should use a custom login page to have a unified look and feel in the entire web frontend of our application. We can use a custom login page with the `FORM` authentication method. To implement the `FORM` authentication method we need to implement a login page and an error page to redirect the users to that page in case authentication fails. Implementing authentication requires us to go through the following steps:

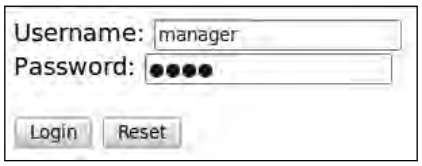
- Implementing `login.html` and `loginError.html`
- Including security description in the `web.xml` and `sun-web.xml` or `sun-application.xml`

Implementing a login page

In `FORM` authentication we implement our own login form to collect username and password and we then pass them to the container for authentication. We should let the container know which field is username and which field is password by using standard names for these fields. The username field is `j_username` and the password field is `j_password`. To pass these fields to container for authentication we should use `j_security_check` as the form action. When we are posting to `j_security_check` the servlet container takes action and authenticates the included `j_username` and `j_password` against the configured realm. The listing below shows `login.html` content.

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

The following figure shows the login page which is shown when an unauthenticated user tries to access a restricted resource:



Implementing a logout page

A user may need to log out of our system after they're finished using it. So we need to implement a logout page. The following listing shows the `logout.jsp` file:

```
<%
session.invalidate();
%>
<body>
    <center>
    <h1>Logout</h1>
    You have successfully logged out.
    </center>
</body>
```

Implementing a login error page

And now we should implement `LoginError.html`, an authentication error page to inform user about its authentication failure.

```
<html>
    <body>
        <h2>A Login Error Occurred</h2>
        Please click <a href="login.html">here</a> for another try.
    </body>
</html>
```

Implementing an access restricted page

When an authenticated user with no required privileges tries to invoke a session bean method, the EJB container throws a `javax.ejb.AccessLocalException`. To show a meaningful error page to our users we should either map this exception to an error page or we should catch the exception, log the event for audition purposes, and then use the `sendError()` method of the `HttpServletResponse` object to send out an error code. We will map the HTTP error code to our custom web pages with meaningful descriptions using the `web.xml` deployment descriptor. You will see which configuration elements we will use to do the mapping. The following snippet shows `AccessRestricted.html` file:

```
<body>
    <center>  <p>You need to login to access the requested
      resource. To login go to <a href="auth/login.html">Login
      Page</a></p></center>
</body>
```

Configuring deployment descriptors

So far we have implemented required files for the FORM-based authentication and we only need to include required descriptions in the `web.xml` file. Looking back at the application requirement definitions, we see that anyone can use meter to centimeter conversion functionality and any other functionality that requires the user to login. We use three different HTML pages for different types of conversion. We do not need any constraint on `toCentimeter.html` therefore we do not need to include any definition for it. Per application description, any employee can access the `toMilli.jsp` page. Defining security constraint for this page is shown in the following listing:

```
<security-constraint>
    <display-name>You should be an employee</display-name>
    <web-resource-collection>
        <web-resource-name>all</web-resource-name>
        <description/>
        <url-pattern>/jsp/toMillimeter.html</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
        <http-method>DELETE</http-method>
    </web-resource-collection>
    <auth-constraint>
        <description/>
        <role-name>employee_role</role-name>
    </auth-constraint>
</security-constraint>
```

We should put enough constraints on the `toInch.jsp` page so that only managers can access the page. The listing included below shows the security constraint definition for this page.

```
<security-constraint>
    <display-name>You should be a manager</display-name>
    <web-resource-collection>
        <web-resource-name>Inch</web-resource-name>
        <description/>
        <url-pattern>/jsp/toInch.html</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <description/>
        <role-name>manager_role</role-name>
    </auth-constraint>
</security-constraint>
```

Finally we need to define any role we used in the deployment descriptor. The following snippet shows how we define these roles in the `web.xml` page.

```
<security-role>
    <description/>
    <role-name>manager_role</role-name>
</security-role>
<security-role>
    <description/>
    <role-name>employee_role</role-name>
</security-role>
```

Looking back at the application requirements, we need to define data constraint and ensure that username and passwords provided by our users are safe during transmission. The following listing shows how we can define the data constraint on the `login.html` page.

```
<security-constraint>
    <display-name>Login page Protection</display-name>
    <web-resource-collection>
        <web-resource-name>Authentication</web-resource-name>
        <description/>
        <url-pattern>/auth/login.html</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <user-data-constraint>
        <description/>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

One more step and our `web.xml` file will be complete. In this step we define an error page for HTML 401 error code. This error code means that application server is unable to perform the requested action due to negative authorization result. The following snippet shows the required elements to define this error page.

```
<error-page>
    <error-code>401</error-code>
    <location>AccessRestricted.html</location>
</error-page>
```

Now that we are finished with declaring the security we can create the conversion pages and after creating these pages we can start with Business layer and its security requirements.

Specifying the security realm

Up to this point we have defined all the constraints that our application requires but we still need to follow one more step to complete the application's security configuration. The last step is specifying the security realm and authentication. We should specify the `FORM` authentication and per-application description; authentication must happen against the company-wide LDAP server.

Here we are going to use the LDAP security realm `LDAPRealm` which we created in *Chapter 2*. We need to import a new LDIF file into our LDAP server, which contains groups and users definition required for this chapter. To import the file we can use the following command, assuming that you downloaded the source code bundle from `https://www.packtpub.com//sites/default/files/downloads/9386_Code.zip` and you have it extracted.

```
import-ldif --ldifFile path/to/chapter03/users.ldif
 --backendID userRoot --clearBackend --hostname 127.0.0.1 --port 4444 --
bindDN cn=gf\ cn=admin --bindPassword admin --trustAll --noPropertiesFile
```

The following table show users and groups that are defined inside the `users.ldif` file.

Username and password	Group membership
`james/james`	`manager, employee`
`meera/meera`	`employee`

We used OpenDS for the realm data storage and it had two users, one in the `employee` group and the other one in the `manager` group. To configure the authentication realm we need to include the following snippet in the `web.xml` file.

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

If we look at our Web and EJB modules as separate modules we must specify the role mappings for each module separately using the GlassFish deployment descriptors, which are `sun-web.xml` and `sun-ejb.xml`. But we are going to bundle our modules as an **Enterprise Application Archive** (**EAR)** file so we can use the GlassFish deployment descriptor for enterprise applications to define the role mapping in one place and let all modules use that definitions. The following listing shows roles and groups mapping in the `sun-application.xml` file.

```
<sun-application>
    <security-role-mapping>
        <role-name>manager_role</role-name>
        <group-name>manager</group-name>
    </security-role-mapping>
    <security-role-mapping>
        <role-name>employee_role</role-name>
        <group-name>employee</group-name>
    </security-role-mapping>
    <realm>LDAPRealm</realm>
</sun-application>
```

The `security-role-mapping` element we used in `sun-application.xml` has the same schema as the `security-role-mapping` element of the `sun-web.xml` and `sun-ejb-jar.xml` files.

You should have noticed that we have a `realm` element in addition to role mapping elements. We can use the `realm` element of the `sun-application.xml` to specify the default authentication realm for the entire application instead of specifying it for each module separately.

Deploying the application client module in the Application Client Container

The application client module can be a first layer Java SE application which directly communicates with the EJB container and uses services like transaction and security management of EJB container through the **Application Client Container**.

When it comes to software structure an application client is not different from a simple Java SE application. It has a `main` method, which is the software entry point and we can access different Java EE services simply with annotation or using deployment descriptors.

The following listing shows the `main` method for our application client, which invokes the `Conversion` Session Bean and prints the result.

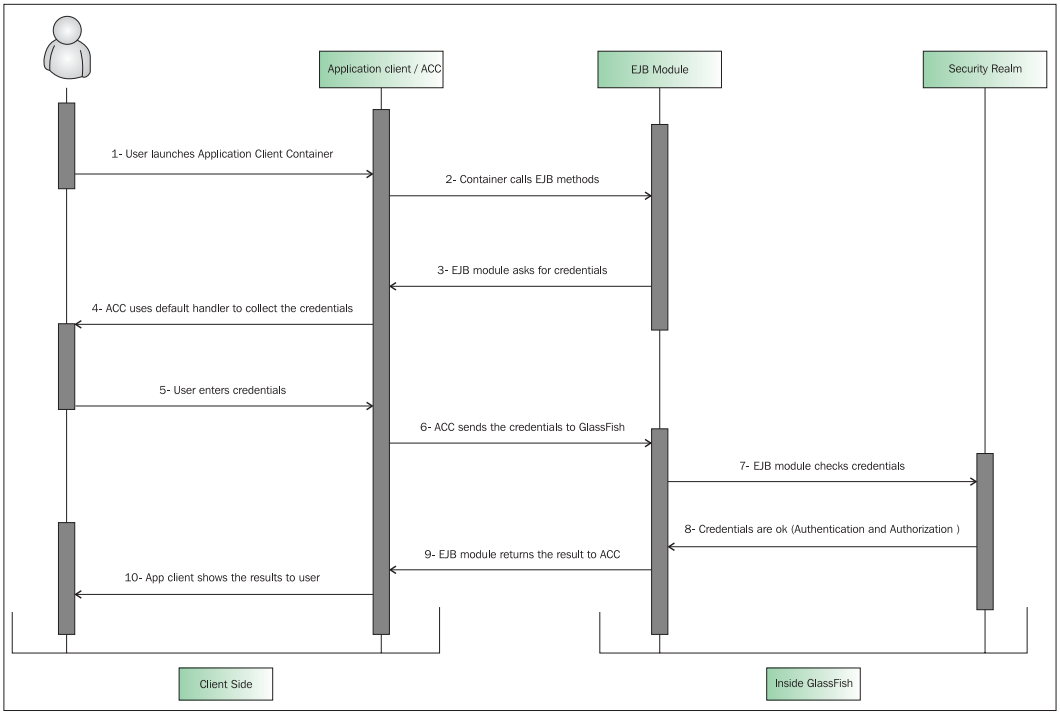
```
public class Main {

    @EJB
    private static ConversionRemote conversionBean;
    public static void main(String[] args) {
        System.out.println(conversionBean.toInch(10));
    }
}
```

You may ask how this application can use injection and access an EJB instance. The secret is, as we saw in *Chapter 1*, hiding in another type of container called the Application Client Container. We deploy an application client module in the ACC and later execute it in the machine either as Java Web Start application or simply using GlassFish-provided scripts. When we run this application the following procedure takes place:

1. Application client (launched using Web Start or directly) results in the ACC trying to inject the secured EJB.
2. The EJB method requires authentication, so GlassFish calls the default `CallbackHandler` to get user login.
3. The default `CallbackHandler`, which is a simple username and password collecting dialog, appears on the client's screen.
4. The collected username and password are sent back to application server for authentication and authorization.
5. After a successful authentication, the method invocation goes through.

This procedure happens even if we do not add any single line of configuration to our EJB module deployment descriptor or Application Client deployment descriptor. The following figure shows more detail about the interaction between different modules when a secure EJB is called from an application client.



The default configuration for application client authentication is summarized in the following table:

Security Measure	Description
Security Realm	If no realm specified in `sun-application.xml` EJB container will use GlassFish default security realm. Default realm is `file` realm if not configured otherwise.
Authentication `CallbackHandler`	Default `CallbackHandler` is a simple swing dialog, which collects username and password.
Transport security	No encryption is applied on data transportation.

All of these measures are configurable either through the Application Client deployment descriptor or the EJB deployment descriptor or the ACC deployment descriptor. The following table shows which attributes are configurable through each one of these deployment descriptors.

Attribute	Deployment descriptor
Authentication mechanism	`sun-ejb-jar.xml`
Security Realm	`sun-acc.xml` and `sun-ejb-jar.xml`
SSL and transport security	`sun-acc.xml` and `sun-ejb-jar.xml`
`CallbackHandler` to collect username and password	`application-client.xml`

Two of the deployment descriptors included in the above table are specific to each vendor and may differ between different application servers. The only standard descriptor is `application-client.xml`, which is a part of the application client standard. This descriptor is placed inside the `META-INF` directory of the client application and contains information like which resources our application is using, how the application is accessing these resources, and finally definitions of the callback handler we want to use to collect user credentials.

The following figure shows default the `CallbackHandler`, which is fired to collect username and password before the container lets the application invoke a method with security constraint.

We can change the default `CallbackHandler` in `application-client.xml` by specifying a new `Callbackhandler`. The new callback should implement the `javax.security.auth.callback.CallbackHandler`. The following snippet shows the `callback-handler` element in `application-client.xml`.

```
<callback-handler>
  book.glassfish.security.chapter3.SwingCallbackHandler
</callback-handler>
```

We can use a programmatic way to provide the ACC with username and password instead of using the callback mechanism to have more control over the authentication procedures. To conduct programmatic login we can use `com.sun.appserv.security.ProgrammaticLogin` class to login before we access any EJB method which has security constraints, defining security measures for communication over IIOP.

We can use the GlassFish-specific deployment descriptor for EJB modules to define several types of configuration elements. We can use one set of these elements to define security measures for communication between the EJB container and the clients over **IIOP** (**Internet Inter-Orb Protocol**).

The super element for the IOR security is `ior-security-config`, which includes the following sub elements:

- The `transport-config` for specifying transport security
- The `sas-context` for specifying the caller propagation options
- The `as-context` for specifying the authentication method, the security realm we want to use for authentication.

Following snippet shows what we should include in the EJB deployment descriptor to get SSL transport security along with username and password-based authentication using the `LDAPRealm` we defined in *Chapter 2*.

```
<ior-security-config>
    <transport-config>
        <integrity>required</integrity>
        <confidentiality>required</confidentiality>
        <establish-trust-in-target>Required
          </establish-trust-in-target>
        <establish-trust-in-client>none</establish-trust-in-client>
    </transport-config>
    <as-context>
        <auth-method>username_password</auth-method>
        <realm>LDAPRealm</realm>
        <required>true</required>
    </as-context>
    <sas-context>
        <caller-propagation>supported</caller-propagation>
    </sas-context>
</ior-security-config>
```

Starting from the top, this snippet instructs the EJB container's IIOP listener to use SSL for data transmission to ensure the integrity and confidentiality of data which is transferred between client and server. Other possible values for `integrity` and `confidentiality` elements are `Supported` and `None`, which means server supports SSL if requested by clients or it does not provide them even if the client asks for data integrity and confidentiality.

We can have SSL mutual authentication by changing the value of `establish-trust-in-target` and `establish-trust-in-client` to required. This way the client will authenticate itself to the server using its digital certificate and in the same way the server will authenticate itself to the client using the digital certificate we specified for IIOP listeners.

When using mutual authentication, we should ensure that the trust store of the client trusts the certificate of the server and the trust store of the server trusts the certificate of the client. To achieve this we should:

1. Add the digital certificate of the client's certificate issuer to the server trust store.
2. Include the digital certificate of the server's certificate issuer to the client's trust store.

Later in the code snippet we have the `as-context` element that we can use to specify which authentication method and security realm we want to use for authenticating clients that need to invoke a secure method of an EJB. The only supported authentication method is `USERNAME_PASSWORD`.

The last element is `sas-context`. We can use it to specify whether EJB container accepts propagated caller identities or not. Possible values are `Supported`, `Required`, and `None`.

Configuring Application Client Container security

The Application Client Container hosts a Java SE layer application that interacts with the EJB container of the application server using IIOP. Each instance of the container can only host one instance of the client application and can be configured for that client application instance.

When we want to run a client application deployed in GlassFish we can either use Java Web Start or the script file provided in the GlassFish `bin` directory. Command format for using the script file is as follow:

```
./appclient -client /opt/dev/Conversion-app-client.
jar -xml /opt/dev/sun-acc.xml
```

It means that we want to launch the `Conversion-app-client.jar` using a configuration file named `sun-acc.xml`.

The `sun-acc.xml` structure follows the schema defined in the `http://www.sun.com/software/appserver/dtds/sun-application-client-container_1_2.dtd` and allows us to configure every aspect of the ACC. The following shows the content of `sun-acc.xml`, which has both authentication and transport security configured.

```
<client-container>
    <target-server name="localhost" address="127.0.0.1" port="3700">
        <security>
            <ssl cert-nickname="s1as"
              ssl2-enabled="false"
              ssl2-ciphers="-rc4,-rc4export,-rc2,-rc2export,-des,
              -desede3"
              ssl3-enabled="true"
              ssl3-tls-ciphers="+rsa_rc4_128_md5,
              -rsa_rc4_40_md5,+rsa3_des_sha,+rsa_des_sha,
              -rsa_rc2_40 _md5,-rsa_null_md5,-rsa_des_56_sha,
              -rsa_rc4_56_sha"
              tls-enabled="true"
              tls-rollback-enabled="true"/>
            <cert-db path="ignored" password="ignored"/>
            <!-- not used -->
        </security>
        <auth-realm name="LDAPRealm"
   classname="com.sun.enterprise.security.auth.realm.ldap.LDAPRealm">
            <property name="directory"
              value="ldap://127.0.0.1:1389"/>
            <property name="base-dn" value=" dc=example,dc=com "/>
            <property name="search-bind-password" value="123456"/>
            <property name="jaas-context" value="ldapRealm"/>
        </auth-realm>
    </target-server>
    <client-credential user-name="james" password="james"/>
</client-container>
```

Starting from the top, we are instructing the container to use a certificate identified by `client` nickname. Later on we will see how we can specify which keystore and trust store we want our client container to use when we launch our application.

All other properties of the `ssl` element specify which SSL version and cipher suites are available to the ACC to choose from. During the negotiation between server and client to establish an SSL session, the strongest cipher suite supported by both server and client is selected.

In addition to configuring the transport security we can configure the authentication mechanism for ACC in order to let ACC collect the identification information and send them back to server when required. Following the security element we have the `auth-realm` element which specifies the authentication realm that ACC must use to conduct the authentication.

You should know all of these properties as we discussed them in great detail in *Chapter 2*. The only thing that you should remember is the fact that this configuration has nothing to do with the LDAP realm we configured in the server. This configuration affects only the client container instance running in the client machine and using this particular `sun-acc.xml` file.

The Application Client Container process exists in the clients machine and anything we configure using the `sun-acc.xml` affects the client machines and has nothing to do with the server or other clients, which run another instance of the application client.

Next we have the `client-credential` element which we can use to specify the default client credential that ACC sends to server instead of collecting the username and password. This element ensures that a single principal is used for all invocation without end users knowing about it.

Using SSL always bring out the issue of keystore and trust store which the application requires using during the SSL handshake and SSL session. There is no vendor-specific way to pass the trust and key store information to Java runtime and rather we can use the JVM environment variables to set these values.

When JVM starts and needs to use SSL, it looks for some environment variables to initiate the SSL session. These variables are included in the following table.

Variable	Description
`javax.net.ssl.keyStore`	Path to keystore containing the client certificate.
`javax.net.ssl.trustStore`	Path to trust store containing certificate issuer's certificates.
`javax.net.ssl.keyStorePassword`	The keystore password.
`javax.net.ssl.trustStorePassword`	The trust store password.

In Linux, we can use the following command to export these variables before launching the application client using the `appclient` script.

```
export VMARGS="-Djavax.net.ssl.keyStore=key-store-path -Djavax.net.ssl.
trustStore= trust-store-path -Djavax.net.ssl.keyStorePassword=key-store-
password -Djavax.net.ssl.trustStorePassword=trust-store-password"
```

For Microsoft Windows we can use the `set` command to set `VMARGS` value as follows:

```
set VMARGS="-Djavax.net.ssl.keyStore=key-store-path -Djavax.net.ssl.
trustStore=trust-store-path -Djavax.net.ssl.keyStorePassword=key-store-
password -Djavax.net.ssl.trustStorePassword=trust-store-password"
```

To create a working pair of certification stores we can follow the same steps we followed to create keystore and trust store for GlassFish application server. Using the same certificate issuer will guarantee that GlassFish will accept the certificate provided by the client and the client will accept the certificate provided by GlassFish.

Now that we have set the required runtime arguments for JVM we can run the client application and be assured about data confidentiality and integrity. The sample application for this chapter is included in the source code archive of the book.

Summary

In this chapter we studied Java EE security in action and developed a secure Java EE application with all of standard modules including EJB, Web, and application clients.

We studied how we can secure EJBs using annotation and then use a web frontend to use the secure EJBs after the user provides correct identification information. We developed a client application to access the secure EJB and later on we studied how we can use SSL and mutual authentication between the application client module and EJB container.

In the next two chapters we will look at GlassFish security independent of the Java EE security and what measures we should consider to have a safe GlassFish installation.