DMIT2015

Java EE 8 Security API

AN INSTITUTE OF TECHNOLOGY COMMITTED TO STUDENT SUCCESS

Learning Objectives

- Upon completion of this lesson, you will be able to:
  - Secure a Java EE application using the security features provided by the Java EE container
  - Apply container-managed authentication, authorization, and data protection

Securing Java EE applications

- How to control and restrict who is permitted to access your applications and what operations users may perform?
- Java EE defines a role-based access control (RBAC) security model for web components and EJBs.
- Java EE containers provides the authentication, authorization, auditing, and mapping capabilities for Java EE applications

Security Terminology

- Authentication
  - Verify who is currently executing an application
  - Usually performed by means of a Login module
- Authorization
  - Verify if a authenticated user has the right (permission) to access system resources or invoke certain operations
- Data Protection
  - Confidential data are encrypted before sending over the network
  - Store passwords using a message digest algorithm
  - Store confidential data in encrypted format

Java Security API

- A container can enforce security in three ways:
    1. **Declarative security** - security requirements are defined using deployment descriptors
    2. **Metadata Annotations** - security requirements are defined within a class file using Java annotations
    3. **Programmatic security** – the developer use can the Java EE security API to check the roles of a user and to access the user's identity

Declarative Security

- Manages an application component's security requirements by means of deployment descriptors.
- Deployment descriptors are XML files.
  - **`web.xml`** for web module
  - **`ejb-jar.xml`** for EJB module

Metadata Annotations

- Security requirements are defined within a class file
  - `@javax.servlet.annotation.`**`ServletSecurity`**
  - `@org.jboss.ejb.annotation.`**`SecurityDomain`**
    - Security domain that is associated with the class/method
  - `@javax.annotation.security.`**`DeclaredRoles`**
    - All the roles the application will use
  - `@javax.annotation.security.`**`RolesAllowed`**
    - The list of roles permitted to access the EJB method
  - `@javax.annotation.security.`**`PermitAll`**
    - EJB method can be invoked by any client
  - `@javax.annotation.security.`**`DenyAll`**
    - EJB method cannot be invoked by external clients
  - `@javax.annotation.security.`**`RunAs`**

Programmatic Security

- In Java EE 8 programmatic security is embedded in the application and used to make security decisions using the following calls:
  - **`javax.security.enterprise.SecurityContext`** instance methods
    - **`isCallerInRole(), getCallerPrincipal()`**
- In Java EE 7 you can use the following calls:
  - **`javax.servlet.http.HttpServletRequest`** instance methods
    - **`isUserInRole(), getUserPrincipal()`** for web module
  - **`javax.ejb.SessionContenxt`** instance methods
    - **`isCallerInRole(), getCallerPrincipal()`** for EJB module

Securing an Enterprise Bean Using Declarative Security

- Method permissions can be specified on the class, the business methods of the class, or both.
- Method permissions can be specified on a method of the bean class to override the method permissions value specified on the entire bean class.

EJB Security Annotations

Annotation	Description	Example

`@DeclareRoles`	Specifies all the roles that the application will use, including roles not specifically named in a @RolesAllowed annotation	`@DeclareRoles("BusinessAdmin")`
		
		`@DeclareRoles({"Administrator", "Manager", "Employee"})`

`@RolesAllowed("list-of-roles")`	Specifies the security roles permitted to access methods in an application	`@RolesAllowed("RestrictedUsers")`
`@PermitAll`	Specifies that all security roles are permitted to execute the specified method or methods	`@PermitAll`
`@DenyAll`	Specifies that no security roles are permitted to execute the specified method or methods	`@DenyAll`

Protect Access to Resources

- Java EE 8 has three kinds of security constraints:

1. Excluded – no external access (denyAll)
2. Unchecked – public access (permitAll)
3. By role

Excluded

- Exclude the webapp **/resources** folder from external access:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>The /resources folders</web-resource-name>
    <url-pattern>/resources/*</url-pattern>
  </web-resource-collection>
</security-constraint>
```

By Role

- Only authenticated users with role VIEW_CUSTOMER_PAGES can access the url-pattern

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Customer Resources</web-resource-name>
    <url-pattern>/customer/playlists.xhtml</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>VIEW_CUSTOMER_PAGES</role-name>
  </auth-constraint>
</security-constraint>
<security-role>
  <role-name>VIEW_CUSTOMER_PAGES</role-name>
</security-role>
```

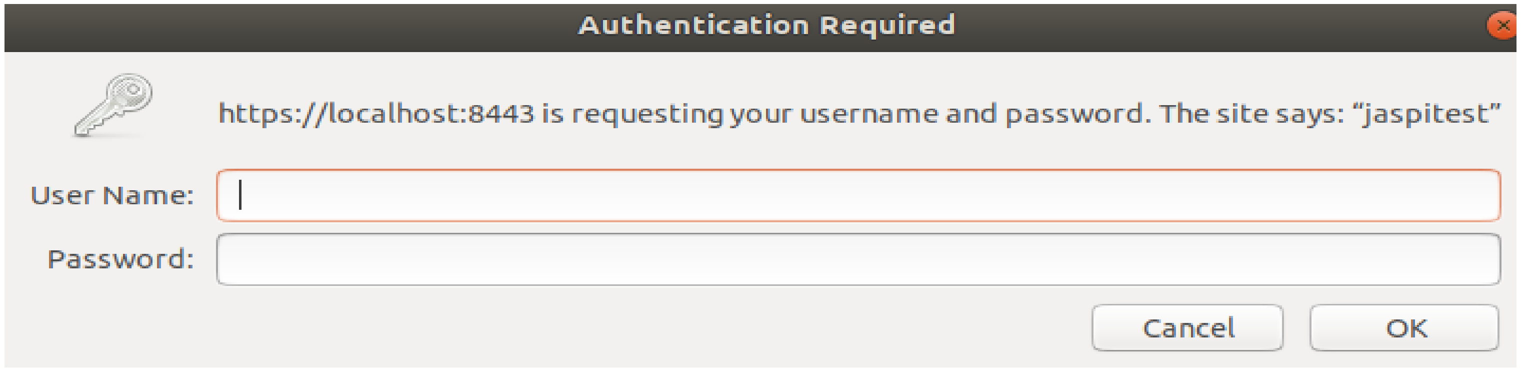
Setting the Authentication Mechanism

- Java EE 8 includes three authentication mechanism:

1. BASIC authentication
   - browser prompts user to login
2. FORM authentication
   - Authentication is done using the URL `/j_security_check` with form fields named `j_username` and `j_password`
3. Custom Form authentication
   - Authentication is done programmatically via a call to the injected SecurityContextHTML form submits to `/j_security_check` with form fields named `j_username` and `j_password` n

Basic Authentication

```
@BasicAuthenticationMechanismDefinition(
  realmName="jaspitest"
)
@ApplicationScoped
public class ApplicationSecurityConfig {
}
```


Authentication Required

https://localhost:8443 is requesting your username and password. The site says: "jaspitest"

User Name:

Password:

Form Authentication (1)

```
@FormAuthenticationMechanismDefinition(
  loginToContinue = @LoginToContinue(
    loginPage="/login.html",
    errorPage="/login.html?error=true",
    useForwardToLogin = false
  )
)
@ApplicationScoped
public class ApplicationSecurityConfig {
}
```

Form Authentication (2)

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

CustomForm Authentication (1)

```
@CustomFormAuthenticationMechanismDefinition(
  loginToContinue = @LoginToContinue(
    loginPage="/security/customLogin.xhtml",
    useForwardToLogin = false,
    errorPage=""
  )
)
@FacesConfig @ApplicationScoped
public class ApplicationSecurityConfig {
}
```

Form Authentication (2)

```
@Named
@ViewScoped
public class Login implements Serializable {
private static final long serialVersionUID = 1L;

@Inject
private SecurityContext securityContext;

@Inject @ManagedProperty("#{param.new}")
private boolean isNew;// added for Caller-Initiated Authentication

@NotBlank(message="Username value is required.")
@Getter @Setter
private String username;

@NotBlank(message="Password value is required.")
@Getter @Setter
private String password;
```

Form Authentication (3)

```
public void submit() {
  switch (continueAuthentication()) {
case SEND_CONTINUE:
Faces.responseComplete();
break;
case SEND_FAILURE:
Messages.addGlobalError("Login failed. Incorrect login credentaials.");
break;
case SUCCESS:
Messages.addFlashGlobalInfo("Login succeed");
Faces.redirect(Faces.getRequestContextPath() + "/index.xhtml");// added for Caller-Initiated
Authentication
break;
case NOT_DONE:
// JSF does not need to take any special action here
break;
  }
}
```

Form Authentication (4)

```
private AuthenticationStatus continueAuthentication() {
  Credential credential = new UsernamePasswordCredential(username, new Password(password) );
  HttpServletRequest request = Faces.getRequest();
  HttpServletResponse response = Faces.getResponse();
  return securityContext.authenticate(request, response,
    AuthenticationParameters.withParams()
    .newAuthentication(isNew)// added for Caller-Initiated Authentication
    .credential(credential));
  }
}
```

Setting the Identity Store

1. Embedded (in-memory)
2. LDAP (Lightweight Directory Access Protocol)
3. Database
4. Custom

Embedded Identity Store

```
@EmbeddedIdentityStoreDefinition({
@Credentials(callerName = "customer",password = "Password2015",groups =
{ "VIEW_CUSTOMER_PAGES" }),
@Credentials(callerName = "employee", password = "Password2015", groups
= { "VIEW_EMPLOYEE_PAGES" }),
@Credentials(callerName = "webadmin",password = "Password2015",groups =
{ "VIEW_EMPLOYEE_PAGES","VIEW_CUSTOMER_PAGES" }),
})
@ApplicationScoped
public class ApplicationSecurityConfig {
}
```

LDAP Identity Store

```
@LdapIdentityStoreDefinition(
  url = "ldap://metro-ds1.nait.ca:389",
  callerSearchBase = "dc=nait,dc=ca",
  callerNameAttribute = "SamAccountName",// SamAccountName or
UserPrincipalName
  groupSearchBase = "dc=nait,dc=ca",
  bindDn = "cn=DMIT
Student1,ou=DMITStudentRestricted,ou=Student,ou=DMIT,ou=SICET,dc=nait,dc
=ca",
  bindDnPassword = "Password2015",
priority = 5
)
@ApplicationScoped
public class ApplicationSecurityConfig {
}
```

Database Identity Store

```
@DatabaseIdentityStoreDefinition(
  dataSourceLookup="java:app/datasources/ChinookSqlServerDS",
  callerQuery="SELECT password FROM LoginCredential WHERE CallerName =
?",
  groupsQuery="SELECT GroupName FROM LoginGroup WHERE CallerName = ? ",
  priority = 10
)
@ApplicationScoped
public class ApplicationSecurityConfig {
}
```

Custom Identity Store

```
@ApplicationScoped
public class CustomIdentityStore implements IdentityStore {
  private LoginService loginService;
  @Inject
  private Pbkdf2PasswordHash passwordHash;
  @Override
  public CredentialValidationResult validate(Credential credential) {
    UsernamePasswordCredential login = (UsernamePasswordCredential) credential;
    String username = login.getCaller();
    LoginUser existingLoginUser = loginService.findOneUserByUsername(username);
    if (existingLoginUser != null &&
passwordHash.verify(login.getPasswordAsString().toCharArray(),
existingLoginUser.getPassword())) {
      return new CredentialValidationResult(username,
existingLoginUser.getGroups().stream().map(item ->
item.getGroupname()).collect(Collectors.toSet()));
    } else {
   return CredentialValidationResult.INVALID_RESULT;
    }
  }
}
```

Logout.java

```
@Named
@RequestScoped
public class Logout {
  @Inject
  private HttpServletRequest request;

  public String submit() throws ServletException {
    request.logout();
    request.getSession().invalidate();
    return "/index?faces-redirect=true";
  }
}
```

LDAP Authentication with Java EE 8 Security API on WildFly

1. Add Java EE 8 Security API implementation library, Soteria, to your project
2. Specify the LDAP identity store
3. Specify the authentication mechanism
4. Declare authentication, authorization, and transport-level encryption on web module
5. Declare security restrictions on EJB module
6. Link the security domain in WildFly to the application

Step 1: Add Java EE 8 Security API reference implementation your project

- Use the following Maven coordinates to specify Soteria in your **pom.xml**:

```
<dependency>
        <groupId>org.glassfish.soteria</groupId>
        <artifactId>jakarta.security.enterprise</artifactId>
        <version>1.0.1</version>
</dependency>
```

Step 2: Specify the LDAP identity store

- To authenticate a secured resource against credentials stored in a LDAP database, we need to annotate an **application-scope CDI bean** with the **@LdapIdentityStoreDefinition** annotation. For example:

```
@LdapIdentityStoreDefinition(
  url = "ldap://192.168.73.137:389",
  callerSearchBase = "ou=JobRole,dc=classicmodelcars,dc=com",
  callerNameAttribute = "SamAccountName",
  groupSearchBase = "ou=JobRole,dc=classicmodelcars,dc=com",
  bindDn = "cn=DMIT2015 Student,ou=Software
Developer,ou=JobRole,dc=classicmodelcars,dc=com",
  bindDnPassword = "Password2015",
  priority = 5
)
@FacesConfig @ApplicationScoped
public class ApplicationSecurityConfig {
}
```

Step 3: Specify the authentication mechanism

- The Java EE 8 Security API supports three authentication mechanisms: basic HTTP authentication, form authentication, and custom form authentication.
- Custom form authentication is specified using the @CustomFormAuthenticationMechanismDefinition annotation. For example:

```
@CustomFormAuthenticationMechanismDefinition(
  loginToContinue = @LoginToContinue(
    loginPage="/security/login.xhtml",
    useForwardToLogin = false,
    errorPage=""
  )
)
@FacesConfig @ApplicationScoped
public class ApplicationSecurityConfig {
}
```

Step 3: (2)

- Create the custom login page

```
<b:form name="loginForm">
   <h2 class="form-signin-heading">Please sign in</h2>
   <b:inputText id="username" label="User Name"
value="#{login.username}">
   </b:inputText>
   <b:inputSecret id="password" label="Password" type="password"
value="#{login.password}">
   </b:inputSecret>
   <b:commandButton value="Login" action="#{login.submit()}" />
</b:form>
```

Step 3: (3)

- Create a CDI named bean for the custom login page

```
@Named
@RequestScoped
public class LoginController {
  @Inject
  private SecurityContext securityContext;
  private String username;
  private String password;
  public void login() {
    Credential credential = new UsernamePasswordCredential(username, new Password(password) );
    HttpServletRequest request = Faces.getRequest();
    HttpServletResponse response = Faces.getResponse();
    AuthenticationStatus status = securityContext.authenticate(request, response,
AuthenticationParameters.withParams().credential(credential));
    if (status.equals(AuthenticationStatus.SEND_CONTINUE)) {
      Faces.responseComplete();
    } else if (status.equals(AuthenticationStatus.SEND_FAILURE)) {
      Messages.addGlobalError("Authentication failed");
    }
  }
}
```

Step 4: Declare security restrictions on web module

- Use the **`<security-constraint>`** element in **`web.xml`** to declare authorization and authentication
- Use the **`<user-data-constraint>`** element of **`<security-constraint>`** in **`web.xml`** to declare transport-level encryption
- Use the **`<security-role>`** element in **`web.xml`** to declare all the roles used in the web module

Step 4: web.xml (transport-level encryption)

- Use the **`<user-data-constraint>`** element of **`<security-constraint>`** in **`web.xml`** to declare transport-level encryption

```
<security-constraint>
  <display-name>Use HTTPS only</display-name>
  <web-resource-collection>
    <web-resource-name>All files and sub-directory</web-resource-name>
    <url-pattern>/*</url-pattern>
    </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

Step 4: web.xml (security roles)

- Use the **`<security-role>`** element in **`web.xml`** to declare security roles used in the application

```
<security-role>
  <role-name>Executive</role-name>
</security-role>
<security-role>
  <role-name>Sales Manager</role-name>
</security-role>
<security-role>
  <role-name>Sales Rep</role-name>
</security-role>
```

Step 4: web.xml (any web.xml authenticated role)

- Use the **`<security-constraint>`** element in **`web.xml`** to declare authorization any role defined in web.xml

```
<security-constraint>
<web-resource-collection>
    <web-resource-name>executive resources</web-resource-name>
    <url-pattern>/report/viewProductLineRevenueReport.xhtml</url-pattern>
</web-resource-collection>
  <auth-constraint>
    <role-name>Executive</role-name>
  </auth-constraint>
</security-constraint>
```

Step 4: web.xml (Multiple Role Resources)

- Use the **`<security-constraint>`** element in **`web.xml`** to declare authorization for one or more role

```
<security-constraint>
<web-resource-collection>
    <web-resource-name>Executive and Sales Manager resources</web-
resource-name>
    <url-pattern>/report/viewTopSellingProductsReport.xhtml</url-pattern>
</web-resource-collection>
  <auth-constraint>
    <role-name>Executive</role-name>
    <role-name>Sales Manager</role-name>
</auth-constraint>
</security-constraint>
```

Step 4: web.xml (Multiple URL Pattern)

- Use the **`<security-constraint>`** element in **`web.xml`** to declare authorization for one or more role

```
<security-constraint>
<web-resource-collection>
    <web-resource-name>Common Resources</web-resource-name>
    <url-pattern>/report/findOrderByOrderNumber.xhtml</url-pattern>
    <url-pattern>/crud/payment/*</url-pattern>
</web-resource-collection>
  <auth-constraint>
    <role-name>Sales Manager</role-name>
    <role-name>Sales Rep</role-name>
</auth-constraint>
</security-constraint>
```

Step 4: web.xml (error pages)

- Use the **`<error-page>`** element in **`web.xml`** to declare location for HTTP status error pages

```
<error-page>
  <error-code>403</error-code>
  <location>/errorpages/403.xhtml</location>
</error-page>
<error-page>
  <error-code>500</error-code>
  <location>/WEB-INF/errorpages/500.xhtml</location>
</error-page>
```

Step 5: Declare security restrictions on EJB module (2)

- Use the metadata annotation **`@DeclareRoles`, `@RolesAllowed`, `@PermitAll`, `@DenyAll`, `@RunAs`** to secure EJB methods

```
@Stateless
@DeclareRoles({"Executive","Sales Manager","Sales Rep"})
@PermitAll
public class ClassicModelsService {
  . . .
  @RolesAllowed({"Sales Manager","Sales Rep"})
  public List<Order> findOrdersByStatus(String status, int maxResults) { }

}
```

Step 6: Link the security domain in WildFly to your application

- Create a WildFly web deployment descriptor file named **`jboss-web.xml`** in **`/webapp/WEB-INF`** directory and set the **security domain** that will be used to authenticate the users.

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-web>

    <security-domain>jaspitest</security-domain>

</jboss-web>
```

Database Authentication with Java EE 8 Security API on WildFly

1. Add Java EE 8 Security API implementation library, Soteria, to your project
2. Code the JPA entity classes to support RBAC
3. Generate the database tables to support RBAC
4. Specify the identity store
5. Specify the authentication mechanism
6. Declare authentication, authorization, and transport-level encryption on web module
7. Declare security restrictions on EJB module
8. Link the security domain in WildFly to the application

Step 1: Add Java EE 8 Security API reference implementation your project

- Use the following Maven coordinates to specify Soteria in your **pom.xml**:

```
<dependency>
        <groupId>org.glassfish.soteria</groupId>
        <artifactId>jakarta.security.enterprise</artifactId>
        <version>1.0.1</version>
</dependency>
```

Step 2: Code the JPA entity classes to support RBAC (1)

```
@Entity
public class LoginUser implements Serializable {
   @Id
   @Column(name="userid",nullable=false)
   private Long id;

   @Column(length=64, unique=true, nullable=false)
   private String username;

   @Column(nullable=false)
   private String password;

   @ManyToMany(fetch=FetchType.EAGER)
   @JoinTable(name="LoginUserGroup", joinColumns={@JoinColumn(name="userid")},
inverseJoinColumns={@JoinColumn(name="groupid")})
   private List<LoginGroup> groups = new ArrayList<>();

   // getters, setters, constructors
}
```

Step 2: (2)

```
@Entity
public class LoginGroup implements Serializable {
   @Id
   @Column(name="groupid",nullable=false)
   private Long id;

   @Column(length=64, unique=true, nullable=false)
   private String groupname;

   @ManyToMany(mappedBy="groups")
   private List<LoginUser> users;@Entity

   // getters, setters, constructors
}
```

Step 2: (3) persistence.xml

```xml
<persistence-unit name="SecurityPU" transaction-type="JTA" >
  <jta-data-source>java:app/datasources/securityapp/SecurityDS</jta-data-source>
  <class>security.entity.LoginUser</class>
  <class>security.entity.LoginGroup</class>
  <properties>
    <property name="javax.persistence.schema-generation.create-source"
value="metadata"/>
    <property name="javax.persistence.schema-generation.drop-source"
value="metadata"/>
<!-- action value: none, create, drop-and-create, drop -->
<!-- <property name="javax.persistence.schema-generation.database.action"
value="drop-and-create"/> -->
    <property name="javax.persistence.schema-generation.scripts.action"
value="create"/>
    <property name="javax.persistence.schema-generation.scripts.create-target"
value="/home/dmit2015/Desktop/create-security-tables.sql"/>
  </properties>
</persistence-unit>
```

Step 3: Create tables on the database (MySQL sample code)

```
CREATE TABLE LoginUser (
   userid BIGINT NOT NULL,
   username VARCHAR(64) NOT NULL,
   password VARCHAR(255) NOT NULL,
   PRIMARY KEY (userid),
   UNIQUE(username)
);

CREATE TABLE LoginGroup(
    groupid BIGINT NOT NULL,
    groupname VARCHAR(64),
    PRIMARY KEY (groupid),
    UNIQUE (groupname)
);
```

Step 3: (2)

```
CREATE TABLE LoginUserGroup(
    userid BIGINT not null,
    groupid BIGINT not null,
    PRIMARY KEY (userid, groupid),
    FOREIGN KEY (userid) references LoginUser(userid),
    FOREIGN KEY (groupid) references LoginGroup(groupid)
);
```

Step 4: Specify the identity store

- To authenticate a secured resource against credentials stored in a relational database, we need to annotate an application-scope CDI bean with the **@DatabaseIdentityStoreDefinition** annotation. For example:

```
@DatabaseIdentityStoreDefinition(
   dataSourceLookup="java:app/datasources/securityapp/SecurityDS",
   callerQuery="SELECT password FROM LoginUser WHERE username = ?",
   groupsQuery="SELECT g.groupname FROM LoginUser u, LoginUserGroup ug, 
LoginGroup g WHERE u.username = ? AND u.id = ug.userid AND ug.groupid = 
g.id"
)
@FacesConfig @ApplicationScoped
public class ApplicationSecurityConfig {
}
```

Step 5: Specify the authentication mechanism

- The Java EE 8 Security API supports three authentication mechanisms: basic HTTP authentication, form authentication, and custom form authentication.
- Custom form authentication is specified using the @CustomFormAuthenticationMechanismDefinition annotation.

```
@CustomFormAuthenticationMechanismDefinition(
   loginToContinue = @LoginToContinue(
      loginPage="/security/login.xhtml",
      useForwardToLogin = false,
      errorPage=""
   )
)
@FacesConfig @ApplicationScoped
public class ApplicationSecurityConfig {
}
```

Step 5: (2)

- Create the custom login page

```
<b:form name="loginForm">
   <h2 class="form-signin-heading">Please sign in</h2>
   <b:inputText id="username" label="User Name"
value="#{login.username}">
   </b:inputText>
   <b:inputSecret id="password" label="Password" type="password"
value="#{login.password}">
   </b:inputSecret>
   <b:commandButton value="Login" action="#{login.submit()}" />
</b:form>
```

Step 5: (3)

- Create a CDI named bean for the custom login page

```
@Named
@RequestScoped
public class LoginController {
  @Inject
  private SecurityContext securityContext;
  private String username;
  private String password;
  public void login() {
    Credential credential = new UsernamePasswordCredential(username, new Password(password) );
    HttpServletRequest request = Faces.getRequest();
    HttpServletResponse response = Faces.getResponse();
    AuthenticationStatus status = securityContext.authenticate(request, response, 
AuthenticationParameters.withParams().credential(credential));
    if (status.equals(AuthenticationStatus.SEND_CONTINUE)) {
      Faces.responseComplete();
    } else if (status.equals(AuthenticationStatus.SEND_FAILURE)) {
      Messages.addGlobalError("Authentication failed");
    }
  }
}
```

Step 6: Declare security restrictions on web module

- Use the **`<security-constraint>`** element in **`web.xml`** to declare authorization and authentication
- Use the **`<user-data-constraint>`** element of **`<security-constraint>`** in **`web.xml`** to declare transport-level encryption
- Use the **`<security-role>`** element in **`web.xml`** to declare all the roles used in the web module

Step 6: web.xml (transport-level encryption)

- Use the **`<user-data-constraint>`** element of **`<security-constraint>`** in **`web.xml`** to declare transport-level encryption

```
<security-constraint>
  <display-name>Use HTTPS only</display-name>
  <web-resource-collection>
    <web-resource-name>sslResources</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

Step 6: web.xml (security roles)

- Use the **`<security-role>`** element in **`web.xml`** to declare security roles used in the application

```
<security-role>
  <role-name>Investor</role-name>
</security-role>
<security-role>
  <role-name>Auditor</role-name>
</security-role>
```

Step 6: web.xml (any web.xml authenticated role)

- Use the **`<security-constraint>`** element in **`web.xml`** to declare authorization any role defined in web.xml

```
<security-constraint>
<web-resource-collection>
    <web-resource-name>anyRoleInWebXml</web-resource-name>
    <url-pattern>/crud/employee/list.xhtml</url-pattern>
</web-resource-collection>
  <auth-constraint>
    <role-name>*</role-name>
  </auth-constraint>
</security-constraint>
```

Step 6: web.xml (any authenticated user)

- Use the **`<security-constraint>`** element in **`web.xml`** to declare authorization any authenticated user

```
<security-constraint>
<web-resource-collection>
  <web-resource-name>anyAuthenticatedUser</web-resource-name>
  <url-pattern>/security/changePassword.xhtml</url-pattern>
</web-resource-collection>
  <auth-constraint>
    <role-name>**</role-name>
  </auth-constraint>
</security-constraint>
```

Step 6: web.xml (Multiple url-pattern)

- Use the **`<security-constraint>`** element in **`web.xml`** to declare authorization for one or more role

```
<security-constraint>
<web-resource-collection>
  <web-resource-name>Investor Resources</web-resource-name>
  <url-pattern>/report/viewProductLineRevenueReport.xhtml</url-pattern>
  <url-pattern>/report/viewTopSellingProductsReport.xhtml</url-pattern>
</web-resource-collection>
  <auth-constraint>
    <role-name>Investor</role-name>
</auth-constraint>
</security-constraint>
```

Step 6: web.xml (Single Role Resource)

- Use the **`<security-constraint>`** element in **`web.xml`** to declare authorization for Employee role

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>auditor resources</web-resource-name>
    <url-pattern>/report/findOrderbyOrderNumber.xhtml</url-pattern>
</web-resource-collection>
  <auth-constraint>
    <role-name>Auditor</role-name>
  </auth-constraint>
</security-constraint>
```

Step 6: web.xml (error pages)

- Use the **`<error-page>`** element in **`web.xml`** to declare location for HTTP status error pages

```
<error-page>
  <error-code>403</error-code>
  <location>/errorpages/403.xhtml</location>
</error-page>
<error-page>
  <error-code>500</error-code>
  <location>/WEB-INF/errorpages/500.xhtml</location>
</error-page>
```

Step 7: Declare security restrictions on EJB module (2)

- Use the metadata annotation **`@DeclareRoles`**, **`@RolesAllowed`**, **`@PermitAll`**, **`@DenyAll`**, **`@RunAs`** to secure EJB methods

```
@Stateless
@DeclareRoles({"Investor","Auditor"})
@PermitAll
public class ClassicModelsService {
  . . .
  @RolesAllowed({"Investor","Auditor"})
  public List<Payment> findAllPayment() { }


}
```

Step 8: Link the security domain in WildFly to your application

- Create a WildFly web deployment descriptor file named `jboss-web.xml` in `/webapp/WEB-INF` directory and set the **security domain** that will be used to authenticate the users.

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-web>

    <security-domain>jaspitest</security-domain>

</jboss-web>
```

JSF-EL expressions for security

- Get the username of the authenticated user

  `#{request.remoteUser}`

- Check if the authenticated user is in the Employee role

  `#{request.isUserInRole('Employee')}`

- Check if user has been authenticated

  `#{request.isUserInRole('**')}`

JSF Security Managed Bean

```
@Named
@SessionScoped
public class SessionController implements Serializable {
  private static final long serialVersionUID = 1L;
  public String logout() throws IOException {
    Faces.invalidateSession();
    return "/index.xhtml?faces-redirect=true";
  }
  public boolean isLoggedIn() {
    return Faces.getRemoteUser() != null;
  }
  public String getRemoteUser() {
    return Faces.getRemoteUser();
  }
  public boolean isUserInRole(String roleName) {
    return Faces.isUserInRole(roleName);
  }
}
```

Using JSF Security Managed Bean

- To logout an authenticated user

```
<b:form>
  <b:navCommandLink value="Logout #{sessionController.remoteUser}"
     action="#{sessionController.logout()}"
     rendered="#{sessionController.loggedIn}">
  </b:navCommandLink>
</b:form>
```

- To login, create a link to a protected page to go to after login

```
<b:navLink value="Sign In"
   outcome="/public/viewJobs"
   rendered="#{not sessionController.loggedIn}">
</b:navLink>
```

Resources

- Java EE 8 Tutorial on Security
- Get started with the Java EE 8 Security API, Part 1
- Get started with the Java EE 8 Security API, Part 2