CONTENTS

Java EE **Security Essentials**

UPDATED BY ARJAN TIJMS, MICHAEL CHENG, ALAINE DEMYERS, CHUNLONG LIANG, AJAY KARKALA, AND TEDDY TORRES

SECURITY IN JAVA EE APPLICATIONS

The Java EE security specification supports a set of required security functionalities, including authentication, authorization, data integrity, and transport security. Before going deep into Java EE security, everyone should know the following terms:

A **Caller** or **User** is an individual named identity defined in an identity store.

A **Credential** contains or references information used to authenticate a caller. Examples of credentials are usernames/passwords, (API) tokens, or JWT tokens.

A **Group** is an abstract concept defined by the Java EE specification and is a name or label that classifies users with a set of common characteristics that usually leads to a set of common permissions. Groups can be broadly defined and mirror real-life groups such as “admin” or “manager”, but can also be more fine-grained and more technically oriented, such as “view_user_pages”. A group can be defined by the application (typically when using an embedded authentication mechanism via e.g. JASPIC) or external to the application (typically when using a vendor-specific authentication mechanism and/or identity store).

A **Role** is another abstract concept defined by the Java EE specification and like a group is also a name or label that classifies users with a set of common characteristics. The difference is in hierarchy; a group can be mapped to one or more roles, and a role, not a group, is used primarily to specify the permissions used to protect application resources. Java EE declarative security, such as constraints in web.xml, only works with roles, but using the JACC specification application resources can be protected directly via groups if so desired. For example, a URL in a web application may be protected by a “Reader” role, or a particular EJB method may be protected by an “Approver” role. Roles are defined by the application.

The **Operational environment** is the environment in which the Java EE application is deployed and that, among others, contains data and systems that are external to the Java EE application. For Java EE security there are two main variations of this environment.

- In the **traditional environment** security users and their groups are defined and managed externally to the Java EE application. This variation comes into play when, for example, externally obtained applications like a bug tracker and a code quality analyzer are integrated into a suite of applications for an office worker. The external user here is the office worker whose identity (say “Pete”) and groups (say “programmer”) are defined office-wide and are independent from any of the applications.

- In the **standalone environment** security users and their groups are defined and managed internally by the Java EE application itself. This variation comes into play when, for example, applications such as an online pet shop or forum are (cloud) hosted and made available to public users that register themselves with the application. In this case there's typically no concept of groups being externally defined for these users.

Principal to role mapping is the process of mapping one or more principals to one or more roles. A principal here can be either a group or a user/caller, which gives us two semantically distinct mappings:

- **Group to role mapping** is the process of mapping a group to a role. In a *traditional operational environment* with externally defined users and groups, this process is primarily used to map external groups to the local roles defined by the application. For example, map office group “programmer” to application role “coder”. In a *standalone environment* this process can be used to create a hierarchy. For example, map application group “admin” to application roles “can_update_wage” and “can_update_interest”.

- **User/caller to role mapping** is the processing of mapping a user/caller to a role. Effectively this means a known user/caller is directly granted an application role. This type of mapping is mostly used in a *traditional operational environment* as the user/caller name has to be known in advance. It can be used in a *standalone operational environment* to grant roles directly to special users such as “root”.

The Java EE spec does not enforce or describe how either of these mappings are defined or how the mapping should be done. Traditionally this mapping is defined using vendor specific deployment descriptors, for example using “*ibm-application-bnd.xml*” in Liberty or glassfish-web.xml in GlassFish and Payara, but using a combination of JASPIC and JACC, this

Webcast

Build Cognitive IoT Robotics with Java and IBM Bluemix

Learn to build a Java application using IBM® MobileFirst™ and IBM Watson® cognitive services to trigger refined messaging for IoT robotics and mobile communications with Twilio and Sendgrid.

Watch here

mapping can be done programmatically in an application-specific way as well.

Different application server vendors may support different ways of mapping. Some may support implicit mappings, where a role in the application is 1:1 mapped to a group, e.g. role “admin” mapped to group “admin”. Others may support more complex mappings, such as mapping a role to combinations of users and groups, e.g. “admin” role mapped to users “John” and “Mary”, and groups “can_update_wage” and “can_update_interest”.

A **Principal** is used to represent any kind of caller/user identity. The Java EE API defines a Principal for the caller; the caller or user principal. Principals are often, but not necessarily, also used by some vendors to internally to represent groups.

An **Identity Store** contains identity data such as credentials and groups, maybe even additional user profile information. Identity stores are typically unaware of the environment in which they are used (such as web/http or RMI) and perform a purely functional {credential in, caller data out} operation. Concrete implementations of a store can be, among others, an RDBMS, LDAP server, or flat file.

An **Authentication mechanism** is a controller that takes care of the interaction of the security system with the caller and the environment to verify the identity of the caller. For example, an authentication mechanism can grab a credential such as a token from an HTTP header, or can redirect to a view that renders a login dialog. Authentication mechanisms typically delegate to an identity store for the credential validation and group retrieval, but are not obliged to do this.

AUTHENTICATION AND AUTHORIZATION IN JAVA EE

A Java EE application server may consist of multiple containers, including the Web/Servlet container, an EJB container, and an Application Client Container (ACC).

The Web container is one of the doors to the EJB container. It performs authentication and propagates the authenticated identity to the EJB container. The EJB container can then use this authenticated identity to make authorization choices, if necessary.

When an EJB container is accessed by an application client or other remote application (collectively called a remote EJB invocation), the EJB container itself performs authentication on the credentials provided by the remote call.

Web and EJB containers host different sets of resources and therefore each one has its separate authorization mechanism suitable for its deployed components.

Each Java EE application can consist of multiple modules. Each one of these modules can have zero or more deployment descriptors which can contain different types of configuration for the application components (JSPs, Servlets, EJBs, etc.) including but not limited to their security descriptions.

Vendors may offer additional functionality through their own specific deployment descriptors. They are in XML format, though that is not a requirement.

Since Java EE 6, more annotations were introduced, which we can use to plan the application deployment. We can override any standard Java EE annotation used in the source code using the corresponding Java EE deployment descriptor elements.

WEB MODULE SECURITY

In the Web module we can declare authentication, authorization, and transport-level encryption using the provided annotations and deployment descriptors.

AUTHENTICATION AND AUTHORIZATION IN WEB MODULE

The following snippet instructs the application server to only let the “manager” role access a resource with a URL matching /mgr/* in our Web application.

```
<security-constraint>
  <display-name>mgr resources</display-name>
  <web-resource-collection>
    <web-resource-name>managers</web-resource-name>
    <description/>
    <url-pattern>/mgr/*</url-pattern>
    <http-method>GET</http-method>
    <http-method>PUT</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <description/>
    <role-name>manager</role-name>
  </auth-constraint>
</security-constraint>
<security-role>
  <description>All Manager </description>
  <role-name>manager</role-name>
</security-role>
```

We defined a security constraint, defined a collection of resources matching the /mgr/* URL, and defined a constraint over the GET, PUT, and POST methods. Then we permitted the manager role to access the constrained resources. We can include as many roles as we need in the auth-constraint element. In case multiple roles are specified the caller will be granted the permission to access the constrained resources if the user has *any* of those roles; in other words, “OR” semantics apply.

Any security role referenced in the auth-constraint elements should be defined using a security-role element as we did for the manager role.

So far we defined which role has access to the secured resource but we still need to let the application server know how to authenticate the users and later on how to determine which roles the user has.

Java EE containers provide via the Servlet spec four standard authentication mechanisms for usage in the web module. Servlet containers should also provide public interfaces for additional (custom) authentication mechanisms. These mechanisms with their specification names are as follows:

1. HTTP Basic Authentication: BASIC
2. Digest Authentication: DIGEST
3. HTTPS Client Authentication: CLIENT-CERT
4. Form-Based Authentication: FORM
5. Custom/Additional: JASPIC

In the first two mechanisms, **BASIC** and **DIGEST**, the mechanism initiates an HTTP basic authentication process and usually the user agent (the browser) then shows a standard dialog to collect the username and the password. The only difference is that when using DIGEST, the client sends a digest of the password instead of sending it in clear text. To use any of these methods we only need to include the following snippet in web.xml.

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>file-realm</realm-name>
</login-config>
```

In the **CLIENT-CERT** mechanism, clients authenticate the server by asking the server for its digital certificate and the server also asks the client to provide its digital certificate to authenticate its identity. In this mode nothing is required to be done except that the client and the server must have a certificate issued by a certificate authority trusted by the other side.

Example: Configure Liberty profile to accept a certificate login

To enable Liberty profile to authenticate with a digital certificate the request must:

1. The request must come in on an HTTPS connection.
2. The SSL/TLS feature ssl-1.0 must be enabled, and Liberty SSL must be configured to have clientAuthentication or clientAuthenticationSupported.
3. The certificate must map to a user in the user registry.

When SSL/TLS client authentication is configured the server will need to trust the client's certificate. The client's trusted certificate will need to be added to the server's truststore.

```
<server>
  <featureManager>
    <feature>ssl-1.0</feature>
  </featureManager>

  <!-- Server SSL configuration -->
  <ssl id="defaultSSLConfig" keyStoreRef="defaultKeyStore"
       trustStoreRef="defaultTrustStore"
       clientAuthenticationSupported="true" />

  <!-- keystore configuration -->
  <keyStore id="defaultKeyStore" location="key.jks"
            type="JKS" password=<file password> />
  <keyStore id="defaultTrustStore" location="trust.jks"
            type="JKS" password=<file password> />
</server>
```

The digital certificate used on the SSL connection must map to a user in the registry. For a file-based user registry the CN value of the certificate must be the username of a user in the registry. The LDAP user registry by default must contain a user entry that matches the complete DN of the certificate. LDAP can be configured to look at specific attributes on the certificate's DN. Custom user registries will have to be written to take into account a certificate must map to a user.

The **FORM** mechanism lets the developer have more control over authentication by letting them provide their own credentials collecting pages. For that we basically create a login and an error page and let the authentication mechanism know about our pages. The authentication mechanism will then use these pages to collect the user credentials. To use this method we should include the following snippet into **web.xml**.

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

The simplest content for the login.jsp page is as follows:

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

The login-error.jsp page can contain any sort of information you feel necessary for the users to understand they provided the wrong credentials.

The **CUSTOM** mechanism option lets application developers provide their own implementation of an authentication mechanism, e.g. to transparently integrate OAuth or OpenID. The interfaces for this and their semantics are described in the Servlet Container Profile of the **JASPIC** (JSR 196) spec. Introduced in Java EE 6, the custom mechanism is now supported by all full Java EE servers and additionally TomEE, Tomcat, and Jetty.

Custom authentication mechanisms can be bundled inside the web application archive and registered via a programmatic API. There is currently (in Java EE 7) no declarative (web.xml or annotation) option available to do this. If a custom authentication mechanism is registered in this way, any login-config present in web.xml is overridden. To avoid confusion it's therefore best to remove such login-config.

The following shows an example of a very basic Java EE custom authentication mechanism that unconditionally authenticates the user without delegating to an identity store:

```
public class TestServerAuthModule implements ServerAuthModule {
  private CallbackHandler handler;
  private Class<?>[] supportedMessageTypes = new Class[] {
    HttpServletRequest.class, HttpServletResponse.class };

  @Override
  public void initialize(MessagePolicy requestPolicy,
    MessagePolicy responsePolicy, CallbackHandler handler,
    @SuppressWarnings("rawtypes") Map options) throws
    AuthException { this.handler = handler; }

  @Override
  public AuthStatus validateRequest(MessageInfo messageInfo,
    Subject clientSubject, Subject serviceSubject) throws
    AuthException {
    CallerPrincipalCallback callerPrincipalCallback =
      new CallerPrincipalCallback(clientSubject, "Bob");
     GroupPrincipalCallback groupPrincipalCallback =
      new GroupPrincipalCallback(
        clientSubject, new String[] { "users" }
      );
     try {
      handler.handle(new Callback[] { callerPrincipalCallback,
          groupPrincipalCallback });
    } catch (IOException | UnsupportedCallbackException e) {
      e.printStackTrace();
    }
    return SUCCESS;
  }

  @Override
  public Class<?>[] getSupportedMessageTypes() {
    return supportedMessageTypes;
  }

  @Override
  public AuthStatus secureResponse(MessageInfo messageInfo,
      Subject serviceSubject) throws AuthException {
    return SEND_SUCCESS;
  }

  @Override
  public void cleanSubject(MessageInfo messageInfo, Subject subject)
throws AuthException {}
}
```

After the authentication mechanism shown above is installed, the `validateRequest()` will be called before every Filter or Servlet in the web application is called and will set the user/caller Principal to "bob" with group "users".

For more info about custom authentication mechanisms see:

- arjan-tijms.omnifaces.org/2012/11/implementing-container-authentication.html
- javamagazine.mozaicreader.com/JulyAug2016#&pageSet=25&page=0&contentItem=0

For the authentication mechanisms that delegate the credential validation check, it is now time to let the application server know where the user's credentials are stored so it can authenticate the received credentials with them and decide whether the user is authentic or not.

This is done by the identity store, but in Java EE, identity stores are not standardized. Individual servers use different terminology for this concept. The list below gives some examples:

- Tomcat – Realm
- Liberty – UserRegistry
- JBoss/WildFly – LoginModule
- Resin – Authenticator
- Jetty- LoginService
- GlassFish/Payara – LoginModule + Realm

Identity stores are set in a vendor-specific way, which is typically a proprietary deployment descriptor, admin console, and/or CLI. The following table shows several examples for a simular type of identity store: a filebased one containing hardcoded users, credentials, and groups. These stores are typically not used for production, but merely for testing and examples.

SERVER	EXAMPLE

Tomcat	**server.xml**
	`<Realm className="org.apache.catalina.realm.MemoryRealm" />`
	
	tomcat-users.xml
	`<tomcat-users>`
	`<role rolename="admins"/>`
	`<role rolename="users"/>`
	`<user username="Bob" password="secret" roles="users"/>`
	`<user username="Paula" password="secret1"`
	`roles="admins,users"/>`
	`</tomcat-users>`

Liberty	**server.xml**
	`<basicRegistry id="basic" realm="basicRealm">`
	`<user name="Bob" password="secret" />`
	`<user name="Paula" password="secret1" />`
	`<group name="admins">`
	`<member name="Paula" />`
	`</group>`
	`<group name="users">`
	`<member name="Bob" />`
	`<member name="Paula" />`
	`</group>`
	`</basicRegistry>`

Jboss/ WildFly	**standalone.xml**
	`<login-module code="org.jboss.security.auth.spi.`
	`UsersRolesLoginModule" flag="required">`
	`<module-option name="usersProperties">users.properties</`
	`module-option>`
	`<module-option name="rolesProperties">roles.properties</`
	`module-option>`
	`</login-module>`
	
	users.properties
	`Bob=secret`
	`Paula=secret1`
	
	roles.properties
	`Bob=users`
	`Paula=admins,users`

GlassFish / Payara	**domain.xml**
	`<auth-realm classname="com.sun.enterprise.security.auth.`
	`realm.file.FileRealm" name="file">`
	`<property name="file" value="${com.sun.aas.instanceRoot}/`
	`config/keyfile"></property>`
	`<property name="jaas-context" value="fileRealm"></property>`
	`</auth-realm>`
	
	Keyfile
	`Bob;{SSHA256}`
	`DZdjK7z7p+3kZxlF8Fk5XNT7pu/51LpVhgYqNZ8idgImiubk7gnOCQ==;users`
	`Paula;{SSHA256}g8Dxl1w5y8FSXqhPufokwCdNTuZmhoKgklQqA4f/`
	`j3Arm2nCXvId5g==;admin,users`

After credentials have been located and validated, it's time to map groups to roles and optionally assign some users roles directly. For some servers, groups **must** be mapped, for some servers it's optional, and some servers don't support mapping at all.

Liberty is a special case. Proprietary group mapping is mandated when groups are set by a proprietary authentication mechanism implementation (using a proprietary identity store called "user registry"), but is not supported when groups are set by a Java EE standard authentication mechanism (a JASPIC SAM).

In the examples below we map the group "users" to role "employees" using a proprietary deployment descriptor from within the application archive:

SERVER	EXAMPLE

Liberty (mandatory when not using JASPIC, otherwise not supported)	**META-INF/ibm-application-bnd.xml**
	`<?xml version="1.0" encoding="UTF-8"?>`
	`<application-bnd xmlns:xsi="http://www.w3.org/2001/`
	`XMLSchema-instance xsi:schemaLocation="http://websphere.`
	`ibm.com/xml/ns/javaee http://websphere.ibm.com/xml/ns/`
	`javaee/ibm-application-bnd_1_1.xsd"`
	`xmlns="http://websphere.ibm.com/xml/ns/javaee"`
	`version="1.1">`
	`<security-role name="employees">`
	`<group name="users" />`
	`</security-role>`
	`</application-bnd>`

GlassFish / Payara (mandatory, unless disabled)	**WEB-INF/glassfish-web.xml**
	`<?xml version="1.0" encoding="UTF-8"?>`
	`<!DOCTYPE glassfish-web-app PUBLIC "-//GlassFish.org//`
	`DTD GlassFish Application Server 3.1 Servlet 3.0//EN"`
	`"http://glassfish.org/dtds/glassfish-web-app_3_0-1.dtd">`
	`<glassfish-web-app>`
	`<security-role-mapping>`
	`<role-name>employees</role-name>`
	`<group-name>users</group-name>`
	`</security-role-mapping>`
	`</glassfish-web-app>`

WebLogic (optional)	**WEB-INF/weblogic.xml**
	`<?xml version = "1.0" encoding = "UTF-8"?>`
	`<weblogic-web-app xmlns:xsi="http://www.w3.org/2001/`
	`XMLSchema-instance"`
	`xsi:schemaLocation="http://www.bea.com/ns/weblogic/`
	`weblogic-web-app.xsd"`
	`xmlns="http://www.bea.com/ns/weblogic/weblogic-web-app">`
	`<security-role-assignment>`
	`<role-name>employees</role-name>`
	`<principal-name>users</principal-name>`
	`</security-role-assignment>`
	`</weblogic-web-app>`

Besides the simple mapping shown above, most vendors support more advanced mappings and support mapping externally to the application. The following shows an example of this for Liberty:

Liberty: server.xml file sample role mapping

```
<application type="war" id="myapp" name="myapp"
            location="${server.config.dir}/apps/myapp.war">
  <application-bnd>
    <security-role name="Employee">
      <user name="user1" />
     <group name="group1" />
    </security-role>
    <security-role name="Manager">
     <user name="user2" />
    <group name="group2" />
    </security-role>
    <security-role name="AllAuthenticated">
      <special-subject type="ALL_AUTHENTICATED_USERS" />
    </security-role>
  </application-bnd>
</application>
```

So far we specified how we can perform authentication and authorization using the container-provided features and vendor-specific mappings of roles. Now we need to address other considerations to secure your web applications.

Passwords and usernames are not protected from eavesdropping when we use FORM or BASIC authentication methods. To protect them from being viewed and intercepted by third parties we should enforce transport security.

ENFORCING TRANSPORT SECURITY

Transport security ensures that no one can tamper with the data being sent to a client or data we receive from a client. Java EE specification lets us enforce the transport security in two levels.

CONFIDENTIAL: By using SSL, this level guarantees that our data is encrypted so that it cannot be deciphered by third parties and the data remains confidential.

INTEGRAL: By using SSL, this level guarantees that the data will not be modified in transit by third parties.

NONE: This level does not apply SSL, and lets the data transport happen as usual.

We can enforce transport security in web.xml using the `user-data-constraint` element, which we should place inside the `security-constraint` tag containing the resource needing transport protection. For example, we can add the following snippet inside `security-constraint`:

```
<user-data-constraint>
  <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
```

We can define as many security-constraints as required and each one of them can use a different user-data-constraint level.

When we specify **CONFIDENTIAL** or **INTEGRAL** as transport guarantee level, the application server will use the HTTPS listener (HTTP listener with SSL enabled) to communicate with the client. Different application servers use a variety of methods to define and configure the HTTPS listeners. Each listener will have a dedicated port like 8080 for HTTP and 8181 for HTTPS.

In a production environment we usually front the application server with a Web server or a dedicated hardware appliance to accelerate the SSL access among other tasks like hosting static content, load distribution, decorating HTTP headers, and so on.

For security purposes, the frontend Web server or appliance can be used to accelerate SSL certificate processing, unify the access port to both HTTP and HTTPS, act as a firewall, and so on.

OTHER SECURITY ELEMENTS OF WEB APPLICATION DEPLOYMENT DESCRIPTORS

Other elements we can use in web.xml for security purposes are listed here:

ELEMENT	DESCRIPTION

`security-role`	Each role must be referenced in a security-role tag before it can be used in the auth-constraint element of a security-constraint. For example:
	`<security-role>`
	`<description>All Manager </description>`
	`<role-name>manager</role-name>`
	`</security-role>`

`session-config`	To specify for how long a session should remain valid. For example:
	`<session-config>`
	`<session-timeout>120</session-timeout>`
	`</session-config>`

`run-as`	To use an specific internal role for any out going call from the Servlet.
	`<run-as>`
	`<role-name>internal_role</role-name>`
	`</run-as>`
	This element resides inside the Servlet tag.

`security-role-ref`	We can alias a role with a more meaningful title and then link the alias to real realm using this element. For example:
	`<security-role-ref>`
	`<role-name>mid_level_managers</role-name>`
	`<role-link>manager</role-link>`
	`</security-role-ref>`

`http-method-omission`	Specify the http-methods where the specific security constraints should be omitted or not enforced. For example, the following constraint excludes access to all methods except GET and POST at the resources matched by the pattern /company/*:
	`<!-- SECURITY CONSTRAINT #5 -->`
	`<security-constraint>`
	`<display-name>Deny all HTTP methods except GET and POST</display-name>`
	`<web-resource-collection>`
	`<url-pattern>/company/*</url-pattern>`
	`<http-method-omission>GET</http-method-omission>`
	`<http-method-omission>POST</http-method-omission>`
	`</web-resource-collection>`
	`<auth-constraint/>`
	`</security-constraint>`

`deny-uncovered-http-methods`	Specify the deny-uncovered-http-methods element to deny access to any of the methods not specified (covered) in the security constraints. `<deny-uncovered-http-methods/>`

special role **	Specify the ** role to allow any authenticated user to access the resource. When the special role name "**" appears in an authorization constraint, it indicates that any authenticated user, independent of role, is authorized to perform the constrained requests.
	In the example below, any authenticated user can access the GET method:
	`<security-constraint>`
	`<display-name>SecurityConstraint1</display-name>`
	`<web-resource-collection>`
	`<web-resource-name>WebResourceCollection1</web-resource-name>`
	`<http-method>GET</http-method>`
	`</web-resource-collection>`
	`<auth-constraint id="AuthConstraint_1">`
	`<role-name>**</role-name>`
	`</auth-constraint>`
	`</security-constraint>`

We use the run-as element or its counterpart annotation to assign a specific role to all outgoing calls from a Servlet or an EJB. We use this element to ensure that an internal role required to access some secured internal EJBs is never assigned to a client and stays fully in control of the developers.

USING ANNOTATIONS TO ENFORCE SECURITY IN WEB MODULES

We can use annotations to enforce security in a Web module. For example, we can specify which roles can access a Servlet by adding some annotations in the Servlet, or we can mark a method in a Servlet stating that no one can access it.

Here is a list of all Java EE 6 annotations and their descriptions:

ANNOTATION	DESCRIPTION
`@DeclareRoles`	Prior to referencing any role, it should be defined. `@DeclareRoles` acts like the security-role element in defining the roles used in application.
`@RunAs`	Specifies the run-as role for the given Components.
`@ServletSecurity`	The `@ServletSecurity` can optionally get `@HttpMethodConstraint` and `@HttpConstraint` as its parameters. The `@HttpMethodConstraint` is an array specifying the HTTP methods specific constraint while `@HttpConstraint` specifies the protection for all HTTP methods which are not specified in the `@HttpMethodConstraint`.
`@PermitAll`	Permits users with any role to access the given method, EJB, or Servlet
`@DenyAll`	If placed on a method, no one can access that method. In case of class-level annotation, all methods of annotated EJB are inaccessible to all roles unless a method is annotated with a `@RolesAllowed` annotation.
`@RolesAllowed`	In case of method-level annotation, it permits the included roles to invoke the method. In case of class-level annotation, all methods of annotated EJB are accessible to included roles unless the method is annotated with a different set of roles using `@RolesAllowed` annotation.

Each of the annotations included here can be placed on different targets like methods, classes, or both, and on different Java EE components like Servlets and EJBs. The next table shows what kind of targets are supported for each one of these annotations.

ANNOTATION	TARGETS LEVEL	TARGET KIND
`@DeclareRoles`	Class	EJB, Servlet
`@RunAs`	Class	EJB, Servlet
`@ServletSecurity`	Class	Servlet
`@PermitAll`	Class, Method	EJB
`@DenyAll`	Method	EJB
`@RolesAllowed`	Class, Method	EJB

Some of the security annotations can not target a method like `@DeclareRoles` while others can target both methods and classes like `@PermitAll`. Annotation applied on a method will override the class level annotations. For example if we apply `@RolesAllowed("employee")` on an EJB class, and we apply `@RolesAllowed("manager")` on one specific method of that EJB, only the manager role will be able to invoke the marked method while all other methods will be available to the employee role.

A role can be mapped to one or more specific principals, groups, or to both of them. The principal or group names must be valid in the specified security realm. The role name we use in the mapping element must match the role-name in the security-role element of the deployment descriptor [web.xml, ejb-jar.xml] or the role name defined in the @DeclareRoles annotation.

PROGRAMMATIC SECURITY IN WEB MODULE

We can access some of the container security context programmatically from our Java source code. The next table shows the seven methods of the HTTPServletRequest class, which we can use to extract security-related attributes of the request and decide manually how to process the request.

METHOD	DESCRIPTIONS
`String getRemoteUser()`	If the user is authenticated, returns the username, otherwise returns null.
`boolean isUserInRole(String role)`	Returns whether the current user has the specified roles or not.
`Principal getUserPrincipal()`	Returns a java.security.Principal object containing the name of the current authenticated user.
`String getAuthType()`	Returns a String containing the authentication method used to protect this application.
`void login(String username, String password)`	This method authenticates the provided username and password against the security realm which the application is configured to use. We can say this method does anything that the BASIC or FORM authentication does but gives the developer total control over how it is going to happen.
`Void logout()`	Establish null as the value returned when `getUserPrincipal`, `getRemoteUser`, and `getAuthType` is called on the request.
`String getScheme()`	Returns the schema portion of the URL, for example HTTP or HTTPS.

The following snippet shows how we check the user role and decide where to redirect him.

```
protected void processRequest(HttpServletRequest request,
  HttpServletResponse response) throws ServletException,
  IOException {
  if (request.isUserInRole("manager")){
    response.sendRedirect("/mgr/index. jsp");
  } else {
    response.sendRedirect("/guests/index.jsp");
  }
}
```

The next snippet demonstrates the use of the `login` method to programmatically log in a user using the container security.

```
String userName = request.getParameter("user");
String password = request.getParameter("password");

try {
  request.login(userName, password);
} catch (ServletException ex) {
  //Handling Exception
  return;
}
```

In the sample code, which can happen inside the `doGet` or `doPost` of a Servlet, we are extracting the username and password from the request and then we use the `login` method to ask the container to authenticate the given username and password against the configured realm.

EJB MODULE SECURITY

Like Web Container and Web module, we can enforce security on EJB modules as well.

In an EJB module we can enforce security (Authentication & Authorization) on Entity Beans and Session Beans. No Security enforcement is defined for the MDBs.

We either access the EJBs through Web container or the ACC. In the first way, the Web container conducts the authentication and propagates the subject to EJB container when using EJBs. In the second way, the ACC performs authentication and passes on the subject during context initialization to the EJB container for authorization.

EJB MODULE DEPLOYMENT DESCRIPTORS

Each EJB module has one or more deployment descriptions containing standard EJB module deployment elements and vendor-specific information.

Let's assume we have an Entity Bean named Employee as follows:

```
@Entity
public class Employee implements Serializable {
  public String getName() { return "name"; }
  public void promote(Position position) { //promote the emplyee }
  public List<EvaluationRecords> getEvaluationRecords() {
    List<EvaluationRecords> evalRecord;
    return evalRecord;
  }

  public List<EvaluationRecords> getEvaluationRecords(Date from,
    Date to) {
    List<EvaluationRecords> evalRecord;
    return evalRecord;
  }

  @Id
  private Integer id;
  public Employee() { }
}
```

Now in the standard deployment descriptor we have can have something like the following snippet to restrict execution of the Employee Bean methods to certain roles:

```
<method-permission>
  <role-name>manager</role-name>
  <method>
    <ejb-name>Employee</ejb-name>
    <method-name>getName</method-name>
  </method>
</method-permission>

<method-permission>
  <role-name>manager</role-name>
  <method>
    <ejb-name>employee</ejb-name>
    <method-name>getEvaluationRecords</method-name>
    <method-params>
      <method-param>from</method-param>
      <method-param>to</method-param>
    </method-params>
  </method>
</method-permission>

<method-permission>
  <role-name>hr_manager</role-name>
  <method>
    <ejb-name>Employee</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>
```

This snippet should be placed inside the EJB declaration to invoke any method of the EJB under the given role.

The snippet is instructing the EJB container to allow any subject with the manager role to invoke `getName`, and only the `getEvaluationRecords` overloads, which takes a date range. Then it allows any subject with hr_manager role to invoke all methods of the Employee EJB.

Like web.xml, we will need to include role definitions in the deployment descriptor. So we will need to add three security-role elements in the ejb-jar.xml file to define the roles we are using. The syntax is the same as the web.xml element which is included in the web module section.

The EJB module performs authentication only when it is accessed from ACC and all configurations for the authentication are provided by the vendor-specific deployment descriptors.

SECURITY ANNOTATION OF EJB MODULES IN JAVA EE 6

Java EE provides a rich set of security-related annotations for the EJB modules. Each of these annotations can be applied on one or more types, as explained previously.

The following snippet shows how we can use these annotations to apply the same security restrictions we declared in the deployment descriptor on the entity source code in the EJB Module Deployment Descriptors section.

```
@Entity
@DeclareRoles({"manager","hr_manager"})
public class Employee implements Serializable {
  @RolesAllowed({"manager","hr_manager"})
  public String getName() { return "name"; }

  @RolesAllowed("hr_manager")
  public void promote(Position position) { ... }

  @RolesAllowed({"manager","hr_manager"})
  public List<EvaluationRecords> getEvaluationRecords() {
    List<EvaluationRecords> evalRecord;
    return evalRecord;
  }

  @RolesAllowed("hr_manager")
  public List<EvaluationRecords> getEvaluationRecords(Date from,
    Date to) {
    List<EvaluationRecords> evalRecord;
    return evalRecord;
  }

  @Id
  private Integer id;
  public Employee() { }
}
```

Using only two annotations, `@RolesAllowed` and `@DeclareRoles`, frees us from adding all deployment descriptor elements.

Similar to Web module, which had a `run-as` element in the standard deployment descriptor, here in EJB module we have the same element. This element will allow outgoing calls from the EJB to use a specific role included in the `role-name` element.

```
<security-identity>
  <run-as>
    <description/>
    <role-name>internal_role</role-name>
  </run-as>
</security-identity>
```

This snippet should be placed inside the EJB declaration element of the deployment descriptor.

Different vendors may have specific non-Java EE compliant annotations for different Java EE components, like JBoss's @SecurityDomain annotation. Using non-standard compliant annotations will make it harder to port an application between different application servers.

SECURING EJB MODULES PROGRAMMATICALLY

We can use EJB context, javax.ejb.EJBContext, to check whether the current user has a specific role using the `isCallerInRole` method, or we can extract the principal name of the subject using the `getCallerPrincipal` method. For example:

```
@Stateless
public class EmployeeServiceBean {
  @Resource SessionContext ctx;
  public void raiseEmployeePaygrade(int amount, long empID) {
    Employee employee = null;
    String raisedBy = ctx.getCallerPrincipal().getName();
    employee.raisePayGrade(850000, raisedBy);
  }
}
```

In the above sample code we injected the context and then we used it to get the principal name. Then we used it to keep record of who changed the salary of employee.

We can use Around Invoke Interceptors to intercept an EJB business method call. Intercepting the call lets us access the method name, its parameters, EJB context (and therefore `isCallerInRole` and `getCallerPrincipal` methods). We can perform tasks like security check, logging, and auditing, or even changing the values of method parameters, using interceptors.

```
public class SampleInterceptor {
  @Resource
  private EJBContext context;
  @AroundInvoke
  protected Object audit(InvocationContext ctx) throws Exception {
    Principal p = context.getCallerPrincipal();
    if (userIsValid(p)) {
      //do some logging... } else {
      //logging and raising exception... }
    return ctx.proceed();
  }
}
```

To use this interceptor we need only to place an annotation on the designated EJB; for example, to intercept any method call on `EmployeeServiceBean`, we can do the following:

```
@Interceptors(SampleInterceptor.class)
@Stateless
public class EmployeeServiceBean {
  // Source code omitted.
}
```

The `@Interceptors` can target classes, methods, or both. To exclude a method from a class-level interceptor, we can use `@ExcludeClassInterceptors` annotation for that method.

We can use interceptor element of ejb-jar.xml deployment descriptor to specify interceptors if preferred.

APPLICATION CLIENT SECURITY

Application Client Container, which can host first-tier clients for enterprise applications, conducts the authentication by itself, and when communicating with the EJBs, sends the authenticated subject along with the call. In the standard deployment descriptor we can configure the callback handler, which collects the user credentials for authentication, and all other measures are configured in the vendor-specific deployment descriptor.

For example, the following snippet specifies the callback handler to collect user identity information:

```
<callback-handler>
  security.refcard.SwingCallBackHandler
</callback-handler>
```

The callback-handler element specifies the name of a callback class provided by the application for JAAS authentication. This class must implement the javax.security.auth.callback.CallbackHandler interface.

```
<resource-ref>
  <res-ref-name>TaskQueueFactory</res-ref-name>
  <jndi-name>jms/TaskQueueFactory</jndi-name>
  <default-resource-principal>
    <name>user</name>
    <password>password</password>
  </default-resource-principal>
</resource-ref>
```

Application Client Container will use the authentication realm specified in the application.xml file to authenticate the users when a request for a constrained EJB is placed.

SECURITY ENFORCEMENT IN GERONIMO ACC

Similar to GlassFish, Geronimo provides some configuration elements in the vendor-specific file named geronimo-application-client.xml.

Notable configuration elements are `default-subject`, `realm-name` and `callback-handler`. For example, to configure the callback handler we can use the callback handler snippet from before.

SECURITY ENFORCEMENT IN JBOSS ACC

Configuration for the JBoss ACC container is stored in the jbossclient.xml file. This configuration file provides no further security customization for the application client.

SECURITY ENFORCEMENT IN LIBERTY ACC

The default CSIv2 configuration for the Liberty ACC can be customized in the vendor-specific client.xml file. For example, we can use the following to specify a user and password instead of using a CallbackHandler implementation:

```
<orb id="defaultOrb">
  <clientPolicy.clientContainerCsiv2>
    <layers>
      <authenticationLayer user="user2" password="usr2pwd" />
    </layers>
  </clientPolicy.clientContainerCsiv2>
</orb>
```

DEFINING SECURITY IN ENTERPRISE APPLICATION LEVEL

The enterprise application archive (EAR) file can contain multiple modules intended to be deployed into different containers like Web, EJB, or ACC. This EAR module has the deployment descriptor of its own, which we can use to include the shared deployment plan details in addition to EAR-specific declarations.

We can use the application-level deployment descriptors to define roles, include the required role mappings, and to specify the default security realm of all included modules.

We can use the standard deployment descriptor to define security roles. The syntax is the same as what we used in the web.xml and the ejb-jar.xml.

Similar to other vendor-specific deployment descriptors, we can use the application-level descriptor to define the application-wide role mapping and also to define the default security realm for all included modules.

The following table shows how we can use different vendor-specific deployment descriptors for role mapping and specifying the default security realm and shows what security measures are accessible through the vendor-specific enterprise application deployment descriptor.

APPLICATION SERVER	DESCRIPTION
GlassFish: sun-application.xml	Role mapping is similar to other Sun specific deployment descriptors. The element is an immediate child of the sun-application
Geronimo: geronimo-application.xml	Roles mapping syntax is similar to other Geronimo-specific deployment descriptors. This element is an immediate child element of the dependencies element which is a subelement of the environment element.
JBoss: jboss-app.xml	Role mapping and specifying the security realm is the same as with jboss-web.xml and jboss.xml using the security domain element.
Liberty: server.xml or at EAR's ibm-application-bnd.xml file	Role mapping in the server.xml file is the same as described for the Web and EJB modules.

SECURING JAVA EE WEB SERVICES

In the Java EE specification, Web services can be deployed as a part of a Web module or an Enterprise Java Bean module.

WEB SERVICES SECURITY IN WEB MODULES

In the Web module we can protect the Web service endpoint the same way we protect any other resource. We can define a resource collection and enforce access management and authentication on it. The most common form of protecting a Web service is using the HTTP Basic or HTTP Digest authentication.

For example, if we use the HTTP basic authentication and our Web service client uses the Dispatch client API to access the Web service we can use a snippet like the following to include the username and password with the right access role to invoke a Web service.

```
<resource-ref>
  <res-ref-name>TaskQueueFactory</res-ref-name>
  <jndi-name>jms/TaskQueueFactory</jndi-name>
  <default-resource-principal>
    <name>user</name>
     <password>password</password>
  </default-resource-principal>
</resource-ref>
```

The user and the password should be valid in the configured realm of the Web application and should have access rights to the endpoint URL.

Another way of authenticating the client to the server in HTTP level is using the Authenticator class, which provides more functionalities and flexibilities. For more information about the authenticator class check java.sun.com/javase/6/docs/technotes/guides/net/http-auth.html

WEB SERVICES SECURITY IN EJB MODULES

We can expose a Stateless Session Bean as a Web Service and therefore we can use all security annotations like `@RolesAllowed`, `@PermitAll`, and their corresponding deployment descriptor elements to define its security plan.

But the authentication enforcement of the Web Services is vendor-specific, and each vendor uses its own method to define the authentication, security realms, and so on.

WEB SERVICES AUTHENTICATION IN GLASSFISH

For GlassFish we should specify the authentication method and the security realm in the sun-ejb-jar.xml. For example, to specify HTTP Basic authentication method and a realm named file_realm as the security realm for a Web service called Echo, we will have something similar to the following snippet:

```
<ejb>
  <ejb-name>Echo</ejb-name>
  <webservice-endpoint>
    <port-component-name>Echo</port-component-name>
    <login-config>
      <auth-method>BASIC</auth-method>
      <realm>file_realm</realm>
    </login-config>
  </webservice-endpoint>
</ejb>
```

WEB SERVICES AUTHENTICATION IN LIBERTY

In Liberty we can authenticate a web service with both SOAP security and HTTP security.

To use HTTP security in an EJB-based web service in Liberty, you edit or add ibm-ws-bnd.xml in the /META-INF directory of your EJB module by adding the login-config attribute with a value of BASIC or CLIENT_CERT. The following snippet can be used in ibm-ws-bnd.xml file:

```
<webservices-bnd>
  <webservice-endpoint-properties>
  <webservice-endpoint>
    <properties>
  <http-publishing>
    <webservice-security>
      <login-config>
        <auth-method>BASIC</auth-method>
      </login-config>
    </webservice-security>
  </http-publishing>
</webservices-bnd>
```

You can use SOAP security in an EJB-based web service in Liberty by selecting one of the WS-Security tokens in SOAP header as an authentication token. To select WS-Security token as caller token, in server.xml file, you add the `callerToken` attribute to the `<wsSecurityProvider>` element with one of the values Usernametoken, X509token, or Samltoken. The following snippet is a sample `callerToken` configuration from server.xml:

```
<wsSecurityProvider ...>
<callerToken name="SamlToken"
...
</wsSecurityProvider>
```

In Geronimo we can use annotations to define the security plan and then the EJB deployment descriptor to specify the authentication mechanism and the security realm. If the following snippet is placed inside the openejb-jar.xml we can expect an HTTP Basic authentication to protect the Echo Web service.

```
<enterprise-beans>
<session>
<ejb-name>Echo</ejb-name>
<web-service-security>
<security-realm-name>file_realm</security-realm-name>
<transport-guarantee>CONFIDENTIAL</transport-guarantee>
<auth-method>BASIC</auth-method>
</web-service-security>
</session>
</enterprise-beans>
```

We simply specified HTTP Basic authentication and the file_realm to be used for this Web service.

WEB SERVICES AUTHENTICATION IN JBOSS

To specify the authentication realm for a Web service deployed as a Stateless Session Bean we can use both annotation and deployment descriptor elements in JBoss. For example, using annotation to secure a Web service can be as follows:

```
@WebService()
@WebContext(contextRoot="/EchoService",
  urlPattern="/Echo",
  authMethod="BASIC",
  secureWSDLAccess = false)
@SecurityDomain(value = "jboss-sec-domain")
@Stateless
...
```

The `@WebContext` annotation simply specifies the Web service endpoint, the authentication method, which can be CLIENT-CERT, BASIC, or DIGEST, and finally it specifies whether clients need to provide credentials to view the WSDL or not.

The `@SecurityDomain` specify which security domain should be used for this Web service authentication.

We can access EJB Web services in the same way we access the Servlet-powered Web services using Dynamic Proxy or the Dispatch API.

Each one of the studied application servers provides a plethora of configuration and tweaking for Web services security to comply with WS-Security profiles. You can check their websites to see what are the available options.

ABOUT THE AUTHOR

ARJAN TIJMS is a JSF (JSR 372) and Security API (JSR 375) EG member. He is the co-creator of the popular OmniFaces library for JSF that was a 2015 Duke's Choice Award winner, and is the main creator of a set of tests for the Java EE authentication SPI (JASPIC) that has been used by various Java EE vendors. Arjan holds an MSc degree in Computer Science from the University of Leiden, The Netherlands.

CONTRIBUTIONS FROM:

Michael Cheng, Lead Security Architect, IBM

Alaine DeMyers, WebSphere Security Development, IBM

Chunlong Liang, WebSphereWeb Service Security, IBM

Ajay Karkala, WebSphere Security, IBM

Teddy Torres, WebSphere Application Server Security Development

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513
888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795
9 781936 502776

BROUGHT TO YOU IN PARTNERSHIP WITH IBM