Getting Started Securing Enterprise Applications

The following parties are responsible for administering security for enterprise applications:

- System administrator: Responsible for setting up a database of users and assigning them to the proper group.

The system administrator is also responsible for setting GlassFish Serverproperties that enable the applications to run properly.

Some security-related examples set up a default principal-to-role mapping, anonymous users, default users, and propagated identities.

When needed for this tutorial, the steps for performing specific tasks are provided.

- **Application developer/bean provider:**
  Responsible for annotating the classes and methods of the enterprise application in order to provide information to the deployer about which methods need to have restricted access.

  This tutorial describes the steps necessary to complete this task.

- **Deployer:** Responsible for taking the security view provided by the application developer and implementing that security upon deployment.

  This document provides the information needed to accomplish this task for the tutorial example applications.

The following topics are addressed here:

Securing Enterprise Beans

Enterprise beans are Java EE components that implement EJB technology.

Enterprise beans run in the EJB container, a runtime environment within the GlassFish Server.

Although transparent to the application developer, the EJB container provides system-level services, such as transactions and security to its enterprise beans, which form the core of transactional Java EE applications.

Enterprise bean methods can be secured in either of the following ways:

- Declarative security (preferred): Expresses an application component's security requirements using either deployment descriptors or annotations.

  The presence of an annotation in the business method of an enterprise bean class that specifies method permissions is all that is needed for method protection and authentication in some situations.

This section discusses this simple and efficient method of securing enterprise beans.

Because of some limitations to the simplified method of securing enterprise beans, you would want to continue to use the deployment descriptor to specify security information in some instances.

An authentication mechanism must be configured on the server for the simple solution to work.

Basic authentication is the GlassFish Server’s default authentication method.

This tutorial explains how to invoke user name/password authentication of authorized users by decorating the enterprise application's business methods with annotations that specify method permissions.

To make the deployer's task easier, the application developer can define security roles.

A security role is a grouping of permissions that a given type of application users must have in order to successfully use the application.

For example, in a payroll application, some users will want to view their own payroll information (*employee*), some will need to view others' payroll information (*manager*), and some will need to be able to change others' payroll information (*payrollDept*).

The application developer would determine the potential users of the application and which methods would be accessible to which users.

The application developer would then decorate classes or methods of the enterprise bean with annotations that specify the types of users authorized to access those methods.

Using annotations to specify authorized users is described in Specifying Authorized Users by Declaring Security Roles.

When one of the annotations is used to define method permissions, the deployment system will automatically require user name/password authentication.

In this type of authentication, a user is prompted to enter a user name and password, which will be compared against a database of known users.

If the user is found and the password matches, the roles that the user is assigned will be compared against the roles that are authorized to access the method.

If the user is authenticated and found to have a role that is authorized to access that method, the data will be returned to the user.

Using declarative security is discussed in Securing an Enterprise Bean Using Declarative Security.

- Programmatic security: For an enterprise bean, code embedded in a business method that is used to access a caller's identity programmatically and that uses this information to make security decisions.

  Programmatic security is useful when declarative security alone is not sufficient to express the security model of an application.

In general, security management should be enforced by the container in a manner that is transparent to the enterprise beans' business methods.

The programmatic security APIs described in this chapter should be used only in the less frequent situations in which the enterprise bean business methods need to access the security-context information,

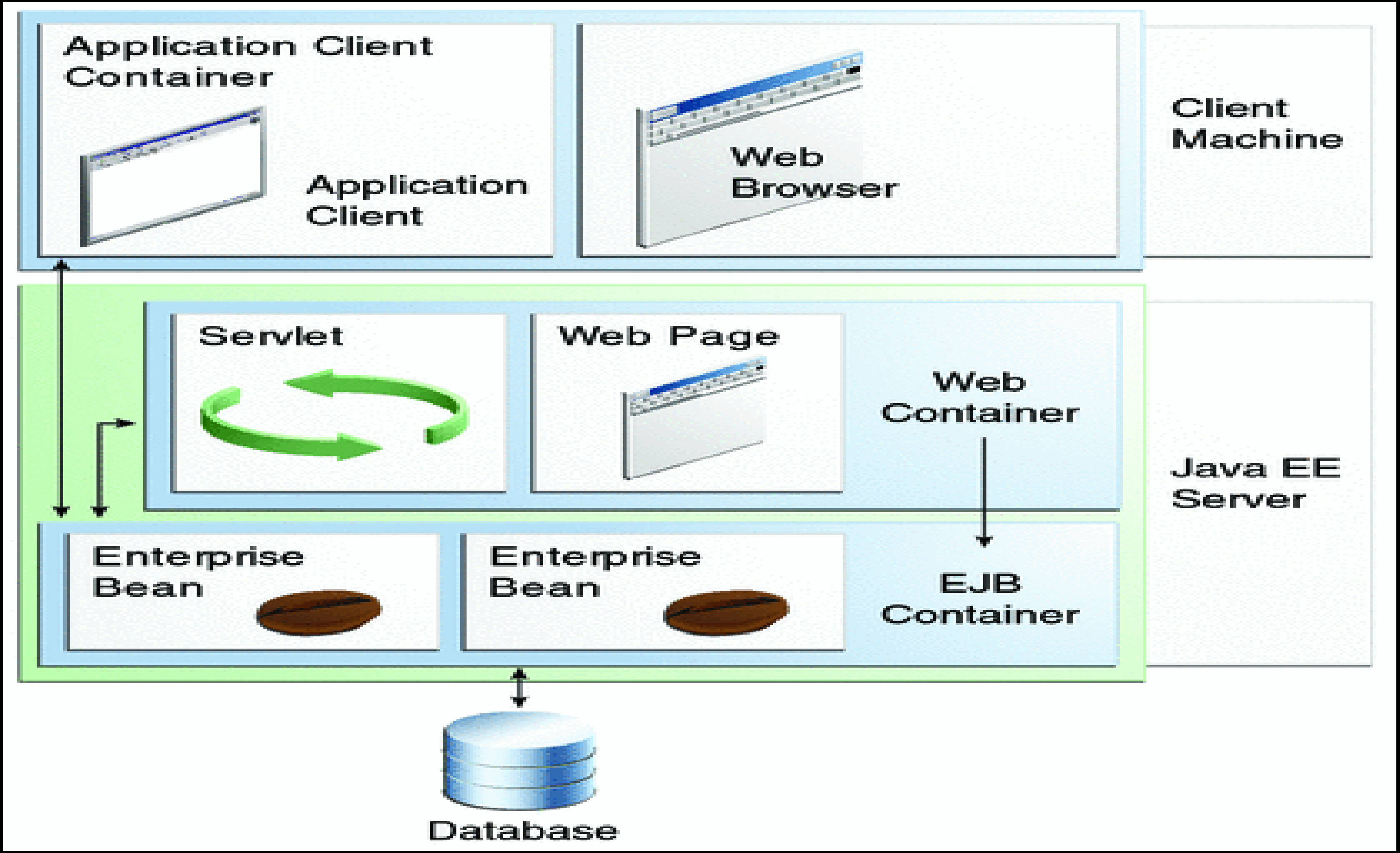
such as when you want to grant access based on the time of day or other nontrivial condition checks for a particular role.

Programmatic security is discussed in Securing an Enterprise Bean Programmatically.

Some of the material in this chapter assumes that you have already read Chapter 22, Enterprise Beans, Chapter 23, Getting Started with Enterprise Beans, and Chapter 39, Introduction to Security in the Java EE Platform.

As mentioned earlier, enterprise beans run in the EJB container, a runtime environment within the GlassFish Server, as shown in Figure 41-1.

Figure 41-1 Java EE Server and Containers



This section discusses securing a Java EE application where one or more modules, such as EJB JAR files, are packaged into an EAR file, the archive file that holds the application.

Security annotations will be used in the Java programming class files to specify authorized users and basic, or user name/password, authentication.

Enterprise beans often provide the business logic of a web application.

In these cases, packaging the enterprise bean within the web application's WAR module simplifies deployment and application organization.

Enterprise beans may be packaged within a WAR module as Java class files or within a JAR file that is bundled within the WAR module.

When a servlet or JavaServer Faces page handles the web front end and the application is packaged into a WAR module as a Java class file, security for the application can be handled in the application's `web.xml` file.

The EJB in the WAR file can have its own deployment descriptor, `ejb-jar.xml`, if required.

Securing web applications using `web.xml` is discussed in Chapter 40, Getting Started Securing Web Applications.

The following sections describe declarative and programmatic security mechanisms that can be used to protect enterprise bean resources.

The protected resources include enterprise bean methods that are called from application clients, web components, or other enterprise beans.

For more information on this topic, read the Enterprise JavaBeans 3.1 specification.

This document can be downloaded from http://jcp.org/en/jsr/detail?id=318.

Chapter 17 of this specification, “Security Management,” discusses security management for enterprise beans.

Securing an Enterprise Bean Using Declarative Security

Declarative security enables the application developer to specify which users are authorized to access which methods of the enterprise beans and to authenticate these users with basic, or username-password, authentication.

Frequently, the person who is developing an enterprise application is not the same person who is responsible for deploying the application.

An application developer who uses declarative security to define method permissions and authentications mechanisms is passing along to the deployer a security view of the enterprise beans contained in the EJB JAR.

When a security view is passed on to the deployer, he or she uses this information to define method permissions for security roles.

If you don’t define a security view, the deployer will have to determine what each business method does to determine which users are authorized to call each method.

A security view consists of a set of security roles, a semantic grouping of permissions that a given type of users of an application must have to successfully access the application.

Security roles are meant to be logical roles, representing a type of user.

You can define method permissions for each security role.

A method permission is a permission to invoke a specified group of methods of an enterprise bean's business interface, home interface, component interface, and/or web service endpoints.

After method permissions are defined, user name/password authentication will be used to verify the identity of the user.

It is important to keep in mind that security roles are used to define the logical security view of an application.

They should not be confused with the user groups, users, principals, and other concepts that exist in the GlassFish Server.

An additional step is required to map the roles defined in the application to users, groups, and principals that are the components of the user database in the file realm of the GlassFish Server.

These steps are outlined in Mapping Roles to Users and Groups.

The following sections show how an application developer uses declarative security to either secure an application or to create a security view to pass along to the deployer.

Specifying Authorized Users by Declaring Security Roles

This section discusses how to use annotations to specify the method permissions for the methods of a bean class.

For more information on these annotations, refer to the Common Annotations for the Java Platform specification at http://jcp.org/en/jsr/detail?id=250.

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.

The following annotations are used to specify method permissions:

- `@DeclareRoles`: Specifies all the roles that the application will use, including roles not specifically named in a `@RolesAllowed` annotation.

The set of security roles the application uses is the total of the security roles defined in the `@DeclareRoles` and `@RolesAllowed` annotations.

The `@DeclareRoles` annotation is specified on a bean class, where it serves to declare roles that can be tested (for example, by calling `isCallerInRole`) from within the methods of the annotated class.

When declaring the name of a role used as a parameter to the `isCallerInRole(String roleName)` method, the declared name must be the same as the parameter value.

The following example code demonstrates the use of the `@DeclareRoles` annotation:

```
@DeclareRoles("BusinessAdmin")
public class Calculator { ... }
```

The syntax for declaring more than one role is as shown in the following example:

```
@DeclareRoles({"Administrator",
"Manager", "Employee"})
```

- `@RolesAllowed("`*list-of-roles*`")`: Specifies the security roles permitted to access methods in an application.

This annotation can be specified on a class or on one or more methods.

When specified at the class level, the annotation applies to all methods in the class.

When specified on a method, the annotation applies to that method only and overrides any values specified at the class level.

To specify that no roles are authorized to access methods in an application, use the `@DenyAll` annotation.

To specify that a user in any role is authorized to access the application, use the `@PermitAll` annotation.

When used in conjunction with the `@DeclareRoles` annotation, the combined set of security roles is used by the application.

The following example code demonstrates the use of the `@RolesAllowed` annotation:

```
@DeclareRoles({"Administrator",
"Manager", "Employee"})
public class Calculator {
```

```
@RolesAllowed("Administrator")
public void setNewRate(int rate)
{ ... }
}
```

- **`@PermitAll`: Specifies that all security roles are permitted to execute the specified method or methods.**

The user is not checked against a database to ensure that he or she is authorized to access this application.

This annotation can be specified on a class or on one or more methods.

Specifying this annotation on the class means that it applies to all methods of the class.

Specifying it at the method level means that it applies to only that method.

The following example code demonstrates the use of the `@PermitAll` annotation:

```
import
javax.annotation.security.*;
@RolesAllowed("RestrictedUsers")
public class Calculator {
```

```
@RolesAllowed("Administrator")
public void setNewRate(int rate)
{//...}
@PermitAll
public long convertCurrency
(long amount) { //... }
}
```

- `@DenyAll`: Specifies that no security roles are permitted to execute the specified method or methods.

  This means that these methods are excluded from execution in the Java EE container.

  The following example code demonstrates the use of the `@DenyAll` annotation:

```
import
javax.annotation.security.*;
@RolesAllowed("Users")
public class Calculator {
@RolesAllowed("Administrator")
public void setNewRate(int rate)
{ //... }
@DenyAll
public long convertCurrency
(long amount){ //... }
}
```

The following code snippet demonstrates the use of the `@DeclareRoles` annotation with the `isCallerInRole` method.

In this example, the `@DeclareRoles` annotation declares a role that the enterprise bean `PayrollBean` uses to make the security check by using `isCallerInRole("payroll")` to verify that the caller is authorized to change salary data:

```
@DeclareRoles("payroll")
@Stateless public class PayrollBean
implements Payroll{
@Resource SessionContext ctx;
public void updateEmployeeInfo
(EmplInfo info){
oldInfo = ...
read from database;
// The salary field can be
// changed only by callers
// who have the security
// role "payroll"
Principal callerPrincipal =
ctx.getCallerPrincipal();
if(info.salary != oldInfo.salary &&
!ctx.isCallerInRole("payroll"))
{throw new SecurityException(...);}
...
}
...
}
```

The following example code illustrates the use of the @RolesAllowed annotation:

```
@RolesAllowed("admin")
public class SomeClass {
public void aMethod () {...}
public void bMethod () {...}
...
}
@Stateless public class MyBean
extends SomeClass implements A{
```

```
@RolesAllowed("HR")
public void aMethod () {...}
public void cMethod () {...}...
}
```

In this example, assuming that `aMethod`, `bMethod`, and `cMethod` are methods of business interface `A`, the method permissions values of methods `aMethod` and `bMethod` are `@RolesAllowed("HR")` and `@RolesAllowed("admin")`, respectively.

The method permissions for method `cMethod` have not been specified.

To clarify, the annotations are not inherited by the subclass itself.

Instead, the annotations apply to methods of the superclass that are inherited by the subclass.

Specifying an Authentication Mechanism and Secure Connection

When method permissions are specified, basic user name/password authentication will be invoked by the GlassFish Server.

To use a different type of authentication or to require a secure connection using SSL, specify this information in an application deployment descriptor.

Securing an Enterprise Bean Programmatically

Programmatic security, code that is embedded in a business method, is used to access a caller's identity programmatically and uses this information to make security decisions within the method itself.

Accessing an Enterprise Bean Caller’s Security Context

In general, security management should be enforced by the container in a manner that is transparent to the enterprise bean’s business methods.

The security API described in this section should be used only in the less frequent situations in which the enterprise bean business methods need to access the security context information, such as when you want to restrict access to a particular time of day.

The `javax.ejb.EJBContext` interface provides two methods that allow the bean provider to access security information about the enterprise bean's caller:

- `getCallerPrincipal`, which allows the enterprise bean methods to obtain the current caller principal's name.

The methods might, for example, use the name as a key to information in a database.

The following code sample illustrates the use of the `getCallerPrincipal` method:

```
@Stateless public class
EmployeeServiceBean implements
EmployeeService {
@Resource SessionContext ctx;
```

```
@PersistenceContext
EntityManager em;
public void
changePhoneNumber(...){...
// obtain the caller principal.
callerPrincipal =
ctx.getCallerPrincipal();
// obtain the caller
// principal's name.
callerKey =
callerPrincipal.getName();
// use callerKey as primary
// key to find EmployeeRecord
EmployeeRecord myEmployeeRecord =
em.find
(EmployeeRecord.class, callerKey);
// update phone number
myEmployeeRecord.setPhoneNumber(...);
...
}
}
```

In this example, the enterprise bean obtains the principal name of the current caller and uses it as the primary key to locate an `EmployeeRecord` entity.

This example assumes that application has been deployed such that the current caller principal contains the primary key used for the identification of employees (for example, employee number).

- `isCallerInRole`, which the enterprise bean code can use to allow the bean provider/application developer to code the security checks that cannot be easily defined using method permissions.

  Such a check might impose a role-based limit on a request, or it might depend on information stored in the database.

The enterprise bean code can use the `isCallerInRole` method to test whether the current caller has been assigned to a given security role.

Security roles are defined by the bean provider or the application assembler and are assigned by the deployer to principals or principal groups that exist in the operational environment.

The following code sample illustrates the use of the `isCallerInRole` method:

```
@Stateless public class
PayrollBean implements Payroll{
@Resource SessionContext ctx;
public void updateEmployeeInfo
(EmplInfo info){
oldInfo = ...
read from database;
// The salary field can be
// changed only by callers
// who have the security
// role "payroll"
if (info.salary != oldInfo.salary
&& !ctx.isCallerInRole("payroll")){
throw new SecurityException(...);
}...
}
...
}
```

You would use programmatic security in this way to dynamically control access to a method, for example, when you want to deny access except during a particular time of day.

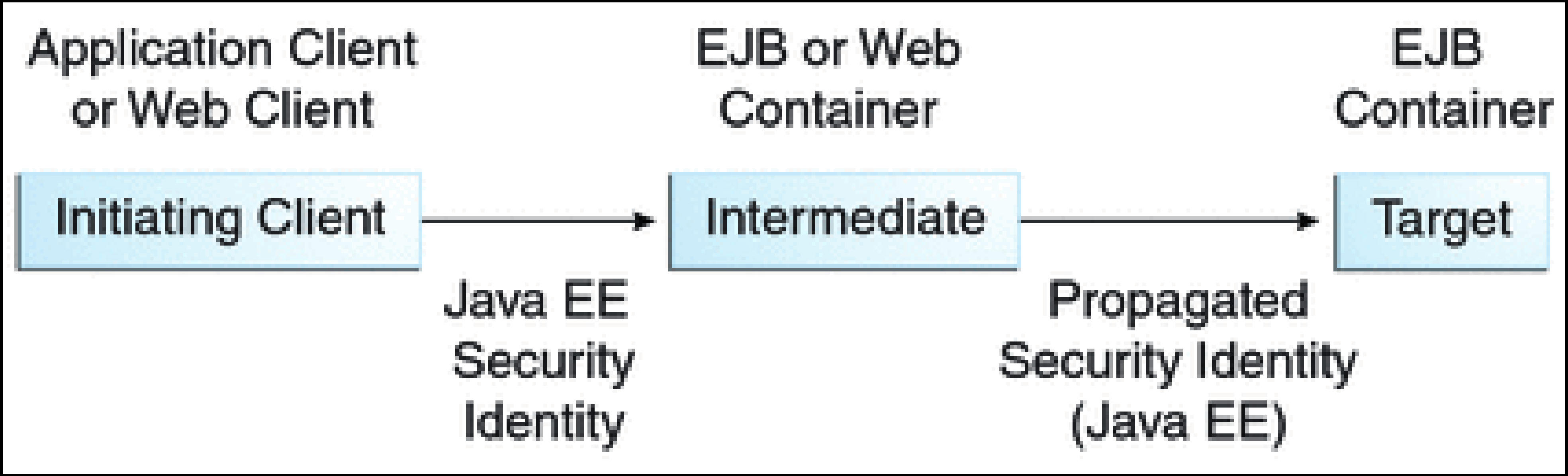
An example application that uses the `getCallerPrincipal` and `isCallerInRole` methods is described in Example: Securing an Enterprise Bean with Programmatic Security.

Propagating a Security Identity (Run-As)

You can specify whether a caller's security identity should be used for the execution of specified methods of an enterprise bean or whether a specific run-as identity should be used.

Figure 41-2 illustrates this concept.

Figure 41-2 Security Identity Propagation


Application Client or Web Client
EJB or Web Container
EJB Container
Initiating Client
Intermediate
Target
Java EE Security Identity
Propagated Security Identity (Java EE)

In this illustration, an application client is making a call to an enterprise bean method in one EJB container.

This enterprise bean method, in turn, makes a call to an enterprise bean method in another container.

The security identity during the first call is the identity of the caller.

The security identity during the second call can be any of the following options.

- By default, the identity of the caller of the intermediate component is propagated to the target enterprise bean.

  This technique is used when the target container trusts the intermediate container.
- A specific identity is propagated to the target enterprise bean.

This technique is used when the target container expects access using a specific identity.

To propagate an identity to the target enterprise bean, configure a run-as identity for the bean, as described in Configuring a Component's Propagated Security Identity.

Establishing a run-as identity for an enterprise bean does not affect the identities of its callers, which are the identities tested for permission to access the methods of the enterprise bean.

The run-as identity establishes the identity that the enterprise bean will use when it makes calls.

The run-as identity applies to the enterprise bean as a whole, including all the methods of the enterprise bean's business interface, local and remote interfaces, component interface, and web service endpoint interfaces, the message listener methods of a message-driven bean, the timeout method of an enterprise bean, and all internal methods of the bean that might be called in turn.

Configuring a Component's Propagated Security Identity

You can configure an enterprise bean's run-as, or propagated, security identity by using the `@RunAs` annotation, which defines the role of the application during execution in a Java EE container.

The annotation can be specified on a class, allowing developers to execute an application under a particular role.

The role must map to the user/group information in the container's security realm.

The `@RunAs` annotation specifies the name of a security role as its parameter.

Here is some example code that demonstrates the use of the `@RunAs` annotation.

```
@RunAs("Admin")
public class Calculator
{ //.... }
```

You will have to map the run-as role name to a given principal defined on the GlassFish Server if the given roles are associated with more than one user principal.

Trust between Containers

When an enterprise bean is designed so that either the original caller identity or a designated identity is used to call a target bean, the target bean will receive the propagated identity only.

The target bean will not receive any authentication data.

There is no way for the target container to authenticate the propagated security identity.

However, because the security identity is used in authorization checks (for example, method permissions or with the `isCallerInRole` method), it is vitally important that the security identity be authentic.

Because no authentication data is available to authenticate the propagated identity, the target must trust that the calling container has propagated an authenticated security identity.

By default, the GlassFish Server is configured to trust identities that are propagated from different containers.

Therefore, you do not need to take any special steps to set up a trust relationship.

Deploying Secure Enterprise Beans

The deployer is responsible for ensuring that an assembled application is secure after it has been deployed in the target operational environment.

If a security view has been provided to the deployer through the use of security annotations and/or a deployment descriptor, the security view is mapped to the mechanisms and policies used by the security domain in the target operational environment, which in this case is the GlassFish Server.

If no security view is provided, the deployer must set up the appropriate security policy for the enterprise bean application.

Deployment information is specific to a web or application server.

Examples: Securing Enterprise Beans

The following examples show how to secure enterprise beans using declarative and programmatic security.

Example: Securing an Enterprise Bean with Declarative Security

This section discusses how to configure an enterprise bean for basic user name/password authentication.

When a bean that is constrained in this way is requested, the server requests a user name and password from the client and verifies that the user name and password are valid by comparing them against a database of authorized users on the GlassFish Server.

If the topic of authentication is new to you, see Specifying an Authentication Mechanism in the Deployment Descriptor.

This example demonstrates security by starting with the unsecured enterprise bean application, `cart`, which is found in the directory

`tut-install``/examples/ejb/cart/` and is discussed in The `cart` Example.

In general, the following steps are necessary to add user name/password authentication to an existing application that contains an enterprise bean.

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

1. **Create an application like the one in The `cart` Example.**

The example in this tutorial starts with this example and demonstrates adding basic authentication of the client to this application.

The example application discussed in this section can be found at

tut-install`/examples/security/cart-secure/`.

2. If you have not already done so, complete the steps in To Set Up Your System for Running the Security Examples to configure your system for running the tutorial applications.

3. Modify the source code for the enterprise bean, `CartBean.java`, to specify which roles are authorized to access which protected methods.

This step is discussed in Annotating the Bean.

4. Build, package, and deploy the enterprise bean; then build and run the client application by following the steps in To Build, Package, Deploy, and Run the Secure Cart Example Using NetBeans IDE or To Build, Package, Deploy, and Run the Secure Cart Example Using Ant.
5.

Annotating the Bean

The source code for the original `cart` application was modified as shown in the following code snippet (modifications in bold).

The resulting file can be found in the following location:

```
tut-install/examples/security/
cart-secure/
cart-secure-ejb/src/java/cart/
ejb/CartBean.java
```

The code snippet is as follows:

```
package cart.ejb;
```

```
import cart.util.BookException;
import cart.util.IdVerifier;
import java.util.ArrayList;
import java.util.List;
import javax.ejb.Remove;
import javax.ejb.Stateful;
import javax.annotation.security.
DeclareRoles;
import javax.annotation.security.
RolesAllowed;
@Stateful
@DeclareRoles("TutorialUser")
```

```
public class CartBean implements
Cart {
List<String> contents;
String customerId;
String customerName;
public void initialize
(String person)
throws BookException{
if (person == null){
throw new BookException
("Null person not allowed.");
} else {customerName = person;}
customerId = "0";
contents = new ArrayList<String>();
}
public void initialize
(String person, String id)
throws BookException{
if (person == null){
throw new BookException
("Null person not allowed.");
} else {customerName = person;}
```

```
IdVerifier idChecker =
new IdVerifier();
if (idChecker.validate(id))
{ customerId = id; }
else {
throw new BookException
("Invalid id: " + id);
}
contents = new ArrayList<String>();
}
```

```
@RolesAllowed("TutorialUser")
public void addBook(String title)
{ contents.add(title); }
@RolesAllowed("TutorialUser")
public void removeBook
(String title)
throws BookException {
boolean result =
contents.remove(title);
if (result == false) {
throw new BookException
("\"" + title + "\" not in cart.");
} }
@RolesAllowed("TutorialUser")
public List<String> getContents()
{ return contents; }
@Remove()
@RolesAllowed("TutorialUser")
public void remove()
{contents = null;} }
```

The `@RolesAllowed` annotation is specified on methods for which you want to restrict access.

In this example, only users in the role of `TutorialUser` will be allowed to add and remove books from the cart and to list the contents of the cart.

A `@RolesAllowed` annotation implicitly declares a role that will be referenced in the application; therefore, no `@DeclareRoles` annotation is required.

The presence of the `@RolesAllowed` annotation also implicitly declares that authentication will be required for a user to access these methods.

If no authentication method is specified in the deployment descriptor, the type of authentication will be user name/password authentication.

To Build, Package, Deploy, and Run the Secure Cart Example Using NetBeans IDE

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

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

3. In the Open Project dialog, navigate to:
   `tut-install/examples/security/`

4. Select the `cart-secure` folder.

5. Select the Open as Main Project and Open Required Projects check boxes.

6. Click Open Project.

7. In the Projects tab, right-click the `cart-secure` project and select Build.

8. In the Projects tab, right-click the `cart-secure` project and select Deploy.

This step builds and packages the application into `cart-secure.ear`, located in the directory *tut-install*`/examples/security/cart-secure/dist/`, and deploys this EAR file to your GlassFish Server instance.

9. To run the application client, right-click the `cart-secure` project and select Run.

A `Login for user:` dialog box appears.

10. In the dialog box, type the user name and password of a file realm user created on the GlassFish Server and assigned to the group `TutorialUser`; then click OK.

If the user name and password you enter are authenticated, the output of the application client appears in the Output pane:

```
...
Retrieving book title from cart:
Infinite Jest
Retrieving book title from cart:
Bel Canto
Retrieving book title from cart:
Kafka on the Shore
Removing "Gravity's Rainbow" from
cart.
Caught a BookException:
"Gravity's Rainbow" not in cart.
```

```
Java Result: 1
...
```

If the user name and password are not authenticated, the dialog box reappears until you type correct values.

To Build, Package, Deploy, and Run the Secure Cart Example Using Ant

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

2. In a terminal window, go to:

   `tut-install``/examples/security/cart-secure/`

3. To build the application and package it into an EAR file, type the following command at the terminal window or command prompt:

```
ant
```

4. To deploy the application to the GlassFish Server, type the following command:

```
ant deploy
```

5. **To run the application client, type the following command:**

```
ant run
```

This task retrieves the application client JAR and runs the application client.

A `Login for user:` dialog box appears.

6. In the dialog box, type the user name and password of a file realm user created on the GlassFish Server and assigned to the group `TutorialUser`; then click OK.

If the user name and password are authenticated, the client displays the following output:

```
[echo] running application client
container.
```

```
[exec] Retrieving book title from
cart: Infinite Jest
[exec] Retrieving book title from
cart: Bel Canto
[exec] Retrieving book title from
cart: Kafka on the Shore
[exec] Removing "Gravity's
Rainbow" from cart.
[exec] Caught a BookException:
"Gravity's Rainbow" not in cart.
[exec] Result: 1
```

If the username and password are not authenticated, the dialog box reappears until you type correct values.

Example: Securing an Enterprise Bean with Programmatic Security

This example demonstrates how to use the `getCallerPrincipal` and `isCallerInRole` methods with an enterprise bean.

This example starts with a very simple EJB application, `converter`, and modifies the methods of the `ConverterBean` so that currency conversion will occur only when the requester is in the role of `TutorialUser`.

The completed version of this example can be found in the directory *tut-install*/`examples/security/converter-secure`.

This example is based on the unsecured enterprise bean application, `converter`, which is discussed in Chapter 23, Getting Started with Enterprise Beans and is found in the directory *tut-install*/`examples/ejb/converter/`.

This section builds on the example by adding the necessary elements to secure the application by using the `getCallerPrincipal` and `isCallerInRole` methods, which are discussed in more detail in Accessing an Enterprise Bean Caller's Security Context.

In general, the following steps are necessary when using the `getCallerPrincipal` and `isCallerInRole` methods with an enterprise bean.

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

1. Create a simple enterprise bean application.

2. Set up a user on the GlassFish Server in the `file` realm, in the group `TutorialUser`, and set up default principal to role mapping.

   To do this, follow the steps in To Set Up Your System for Running the Security Examples.

3. Modify the bean to add the `getCallerPrincipal` and `isCallerInRole` methods.

4. If the application contains a web client that is a servlet, specify security for the servlet, as described in Specifying Security for Basic Authentication Using Annotations.

5. **Build, package, deploy, and run the application.**

Modifying `ConverterBean`

The source code for the original `ConverterBean` class was modified to add the `if..else` clause that tests whether the caller is in the role of `TutorialUser`..

If the user is in the correct role, the currency conversion is computed and displayed.

If the user is not in the correct role, the computation is not performed, and the application displays the result as `0`.

The code example can be found in the following file:

`*tut-install*/examples/ejb/converter-secure/converter-secure-ejb/src/java/converter/ejb/ConverterBean.java`

The code snippet (with modifications shown in bold) is as follows:

```
package converter.ejb;
import java.math.BigDecimal;
import javax.ejb.Stateless;
import java.security.Principal;
import javax.annotation.Resource;
import javax.ejb.SessionContext;
import javax.annotation.security.
DeclareRoles;
```

```
import javax.annotation.security.
RolesAllowed;
@Stateless()
@DeclareRoles("TutorialUser")
public class ConverterBean{
@Resource SessionContext ctx;
private BigDecimal yenRate =
new BigDecimal("89.5094");
private BigDecimal euroRate =
new BigDecimal("0.0081");
```

```
@RolesAllowed("TutorialUser")
public BigDecimal dollarToYen
(BigDecimal dollars) {
BigDecimal result =
new BigDecimal("0.0");
Principal callerPrincipal =
ctx.getCallerPrincipal();
if(ctx.isCallerInRole("TutorialUser")){
result = dollars.multiply(yenRate);
return result.setScale
(2, BigDecimal.ROUND_UP);
} else {
return result.setScale
(2, BigDecimal.ROUND_UP);
}
}
@RolesAllowed("TutorialUser")
public BigDecimal yenToEuro
(BigDecimal yen){
BigDecimal result =
new BigDecimal("0.0");
```

```
Principal callerPrincipal =
ctx.getCallerPrincipal();
if
(ctx.isCallerInRole("TutorialUser")){
result = yen.multiply(euroRate);
return result.setScale
(2, BigDecimal.ROUND_UP);
} else {
return result.setScale
(2, BigDecimal.ROUND_UP);
} } }
```

Modifying `ConverterServlet`

The following annotations specify security for the `converter` web client, `ConverterServlet`:

```
@WebServlet(
name = "ConverterServlet",
urlPatterns = {"/"})

@ServletSecurity(
@HttpConstraint(transportGuarantee
= TransportGuarantee.CONFIDENTIAL,
 rolesAllowed = {"TutorialUser"}))
```

To Build, Package, and Deploy the Secure Converter Example Using NetBeans IDE

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

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

3. In the Open Project dialog, navigate to:

   `tut-install/examples/security/`

4. Select the `converter-secure` folder.

5. Select the Open as Main Project check box.

6. Click Open Project.

7. Right-click the `converter-secure` project and select Build.

8. Right-click the `converter-secure` project and select Deploy.

To Build, Package, and Deploy the Secure Converter Example Using Ant

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

2. In a terminal window, go to:

   `tut-install/examples/security/converter-secure/`

3. Type the following command:

```
ant all
```

This command both builds and deploys the example.

To Run the Secure Converter Example

1. Open a web browser to the following URL:

   `http://localhost:8080/converter-secure`

   An Authentication Required dialog box appears.

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

The screen shown in Figure 23-1 appears.

3. Type `100` in the input field and click Submit.

A second page appears, showing the converted values.

Securing Application Clients

The Java EE authentication requirements for application clients are the same as for other Java EE components, and the same authentication techniques can be used as for other Java EE application components.

No authentication is necessary when accessing unprotected web resources.

When accessing protected web resources, the usual varieties of authentication can be used: HTTP basic authentication, SSL client authentication, or HTTP login-form authentication.

These authentication methods are discussed in Specifying an Authentication Mechanism in the Deployment Descriptor.

Authentication is required when accessing protected enterprise beans.

The authentication mechanisms for enterprise beans are discussed in Securing Enterprise Beans.

An application client makes use of an authentication service provided by the application client container for authenticating its users.

The container's service can be integrated with the native platform's authentication system, so that a single sign-on capability is used.

The container can authenticate the user either when the application is started or when a protected resource is accessed.

An application client can provide a class, called a login module, to gather authentication data.

If so, the `javax.security.auth.callback.CallbackHandler` interface must be implemented, and the class name must be specified in its deployment descriptor.

The application's callback handler must fully support `Callback` objects specified in the `javax.security.auth.callback` package.

Using Login Modules

An application client can use the Java Authentication and Authorization Service (JAAS) to create login modules for authentication.

A JAAS-based application implements the `javax.security.auth.callback.CallbackHandler` interface so that it can interact with users to enter specific authentication data, such as user names or passwords, or to display error and warning messages.

Applications implement the `CallbackHandler` interface and pass it to the login context, which forwards it directly to the underlying login modules.

A login module uses the callback handler both to gather input, such as a password or smart card PIN, from users and to supply information, such as status information, to users.

Because the application specifies the callback handler, an underlying login module can remain independent of the various ways that applications interact with users.

For example, the implementation of a callback handler for a GUI application might display a window to solicit user input.

Or the implementation of a callback handler for a command-line tool might simply prompt the user for input directly from the command line.

The login module passes an array of appropriate callbacks to the callback handler's `handle` method, such as a `NameCallback` for the user name and a `PasswordCallback` for the password;

the callback handler performs the requested user interaction and sets appropriate values in the callbacks.

For example, to process a `NameCallback`, the `CallbackHandler` might prompt for a name, retrieve the value from the user, and call the `setName` method of the `NameCallback` to store the name.

For more information on using JAAS for login modules for authentication, refer to the following sources (see Further Information about Security for the URLs):

- ***Java Authentication and Authorization Service (JAAS) Reference Guide***
- ***Java Authentication and Authorization Service (JAAS): LoginModule Developer's Guide***

Using Programmatic Login

Programmatic login enables the client code to supply user credentials.

If you are using an EJB client, you can use the `com.sun.appserv.security.ProgrammaticLogin` class with its convenient `login` and `logout` methods.

Programmatic login is specific to a server.

Securing Enterprise Information Systems Applications

In EIS applications, components request a connection to an EIS resource.

As part of this connection, the EIS can require a sign-on for the requester to access the resource.

The application component provider has two choices for the design of the EIS sign-on:

- **Container-managed sign-on:** The application component lets the container take the responsibility of configuring and managing the EIS sign-on.

The container determines the user name and password for establishing a connection to an EIS instance.

For more information, see Container-Managed Sign-On.

- **Component-managed sign-on: The application component code manages EIS sign-on by including code that performs the sign-on process to an EIS.**

  For more information, see Component-Managed Sign-On.

You can also configure security for resource adapters.

See Configuring Resource Adapter Security for more information.

Container-Managed Sign-On

In container-managed sign-on, an application component does not have to pass any sign-on security information to the `getConnection()` method.

The security information is supplied by the container, as shown in the following example:

```
// Business method in
// an application component
Context initctx =
new InitialContext();
// Perform JNDI lookup to obtain
// a connection factory
javax.resource.cci.
ConnectionFactory cxf =
(javax.resource.cci.ConnectionFactory)
```

```
initctx.lookup
("java:comp/env/eis/MainframeCxFactory");
// Invoke factory to obtain
// a connection. The security
// information is not passed in
// the getConnection method
javax.resource.cci.Connection cx =
cxf.getConnection();
...
```

Component-Managed Sign-On

In component-managed sign-on, an application component is responsible for passing the needed sign-on security information to the resource to the `getConnection` method.

For example, security information might be a user name and password, as shown here:

```
// Method in
// an application component
Context initctx =
new InitialContext();
// Perform JNDI lookup to obtain
// a connection factory
javax.resource.cci.
ConnectionFactory cxf =
(javax.resource.cci.
ConnectionFactory)initctx.lookup
("java:comp/env/eis/MainframeCxFactory");
```

```
// Get a new ConnectionSpec
com.myeis.ConnectionSpecImpl
properties = //..
// Invoke factory to obtain
// a connection
properties.setUserName("...");
properties.setPassword("...");
javax.resource.cci.Connection cx =
cxf.getConnection(properties);
...
```

Configuring Resource Adapter Security

A resource adapter is a system-level software component that typically implements network connectivity to an external resource manager.

A resource adapter can extend the functionality of the Java EE platform either by implementing one of the Java EE standard service APIs, such as a JDBC driver, or by defining and implementing a resource adapter for a connector to an external application system.

Resource adapters can also provide services that are entirely local, perhaps interacting with native resources.

Resource adapters interface with the Java EE platform through the Java EE service provider interfaces (Java EE SPI).

A resource adapter that uses the Java EE SPIs to attach to the Java EE platform will be able to work with all Java EE products.

To configure the security settings for a resource adapter, you need to edit the resource adapter descriptor file, `ra.xml`.

Here is an example of the part of an `ra.xml` file that configures the following security properties for a resource adapter:

```
<authentication-mechanism>
<authentication-mechanism-type>
BasicPassword
</authentication-mechanism-type>
<credential-interface>
javax.resource.spi.security.
PasswordCredential
```

```
</credential-interface>
</authentication-mechanism>
<reauthentication-support>
false
</reauthentication-support>
```

You can find out more about the options for configuring resource adapter security by reviewing

as-install`/lib/dtds/connector_1_0.dtd`.

You can configure the following elements in the resource adapter deployment descriptor file:

- **Authentication mechanisms:** Use the `authentication-mechanism` element to specify an authentication mechanism supported by the resource adapter.

  This support is for the resource adapter, not for the underlying EIS instance.

There are two supported mechanism types:

- `BasicPassword`, which supports the following interface:

  `javax.resource.spi.security.PasswordCredential`

- `Kerbv5`, which supports the following interface:

  `javax.resource.spi.security.GenericCredential`

  The GlassFish Server does not currently support this mechanism type.

- **Reauthentication support:** Use the `reauthentication-support` element to specify whether the resource adapter implementation supports reauthentication of existing `Managed-Connection` instances.

  Options are `true` or `false`.

- **Security permissions: Use the `security-permission` element to specify a security permission that is required by the resource adapter code.**

  Support for security permissions is optional and is not supported in the current release of the GlassFish Server.

You can, however, manually update the `server.policy` file to add the relevant permissions for the resource adapter.

The security permissions listed in the deployment descriptor are different from those required by the default permission set as specified in the connector specification.

For more information on the implementation of the security permission specification, visit http://download.oracle.com/javase/6/docs/technotes/guides/security/PolicyFiles.html#FileSyntax.

In addition to specifying resource adapter security in the `ra.xml` file, you can create a security map for a connector connection pool to map an application principal or a user group to a back-end EIS principal.

The security map is usually used if one or more EIS back-end principals are used to execute operations (on the EIS) initiated by various principals or user groups in the application.

To Map an Application Principal to EIS Principals

When using the GlassFish Server, you can use security maps to map the caller identity of the application (principal or user group) to a suitable EIS principal in container-managed transaction-based scenarios.

When an application principal initiates a request to an EIS, the GlassFish Server first checks for an exact principal by using the security map defined for the connector connection pool to determine the mapped back-end EIS principal.

If there is no exact match, the GlassFish Server uses the wildcard character specification, if any, to determine the mapped back-end EIS principal.

Security maps are used when an application user needs to execute EIS operations that require to be executed as a specific identity in the EIS.

To work with security maps, use the Administration Console.

From the Administration Console, follow these steps to get to the security maps page.

1. In the navigation tree, expand the Resources node.

2. Expand the Connectors node.

3. Select the Connector Connection Pools node.

4. On the Connector Connection Pools page, click the name of the connection pool for which you want to create a security map.

5. Click the Security Maps tab.

6. Click New to create a new security map for the connection pool.

7. Type a name by which you will refer to the security map, as well as the other required information.

   Click the Help button for more information on the individual options.