Part VII
Security

Part VII explores security concepts and examples.

This part contains the following chapters:

Introduction to Security in the Java EE Platform

The chapters in Part VII discuss security requirements in web tier and enterprise tier applications.

Every enterprise that has either sensitive resources that can be accessed by many users or resources that traverse unprotected, open, networks, such as the Internet, needs to be protected.

This chapter introduces basic security concepts and security mechanisms.

More information on these concepts and mechanisms can be found in the chapter on security in the Java EE 6 specification.

This document is available for download online at http://www.jcp.org/en/jsr/detail?id=316.

In this tutorial, security requirements are also addressed in the following chapters.

- Chapter 40, Getting Started Securing Web Applications explains how to add security to web components, such as servlets.

- Chapter 41, Getting Started Securing Enterprise Applications explains how to add security to Java EE components, such as enterprise beans and application clients.

Some of the material in this chapter assumes that you understand basic security concepts.

To learn more about these concepts before you begin this chapter, you should explore the Java SE security web site at http://download.oracle.com/javase/6/docs/technotes/guides/security/.

The following topics are addressed here:

- Overview of Java EE Security
- Security Mechanisms
- Securing Containers
- Securing the GlassFish Server
- Working with Realms, Users, Groups, and Roles
- Establishing a Secure Connection Using SSL
- Further Information about Security

Overview of Java EE Security

Enterprise tier and web tier applications are made up of components that are deployed into various containers.

These components are combined to build a multitier enterprise application.

Security for components is provided by their containers.

A container provides two kinds of security: declarative and programmatic.

- Declarative security expresses an application component's security requirements by using either deployment descriptors or annotations.

A deployment descriptor is an XML file that is external to the application and that expresses an application's security structure, including security roles, access control, and authentication requirements.

For more information about deployment descriptors, read Using Deployment Descriptors for Declarative Security.

Annotations, also called metadata, are used to specify information about security within a class file.

When the application is deployed, this information can be either used by or overridden by the application deployment descriptor.

Annotations save you from having to write declarative information inside XML descriptors.

Instead, you simply put annotations on the code, and the required information gets generated.

For this tutorial, annotations are used for securing applications wherever possible.

For more information about annotations, see Using Annotations to Specify Security Information.

- Programmatic security is embedded in an application and is used to make security decisions.
-
-

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

For more information about programmatic security, read Using Programmatic Security.

A Simple Security Example

The security behavior of a Java EE environment may be better understood by examining what happens in a simple application with a web client, a user interface, and enterprise bean business logic.

In the following example, which is taken from the Java EE 6 Specification, the web client relies on the web server to act as its authentication proxy by collecting user authentication data from the client and using it to establish an authenticated session.

Step 1: Initial Request

In the first step of this example, the web client requests the main application URL.

This action is shown in Figure 39-1.

Figure 39-1 Initial Request

Since the client has not yet authenticated itself to the application environment, the server responsible for delivering the web portion of the application, hereafter referred to as the web server, detects this and invokes the appropriate authentication mechanism for this resource.

For more information on these mechanisms, see Security Mechanisms.

Step 2: Initial Authentication

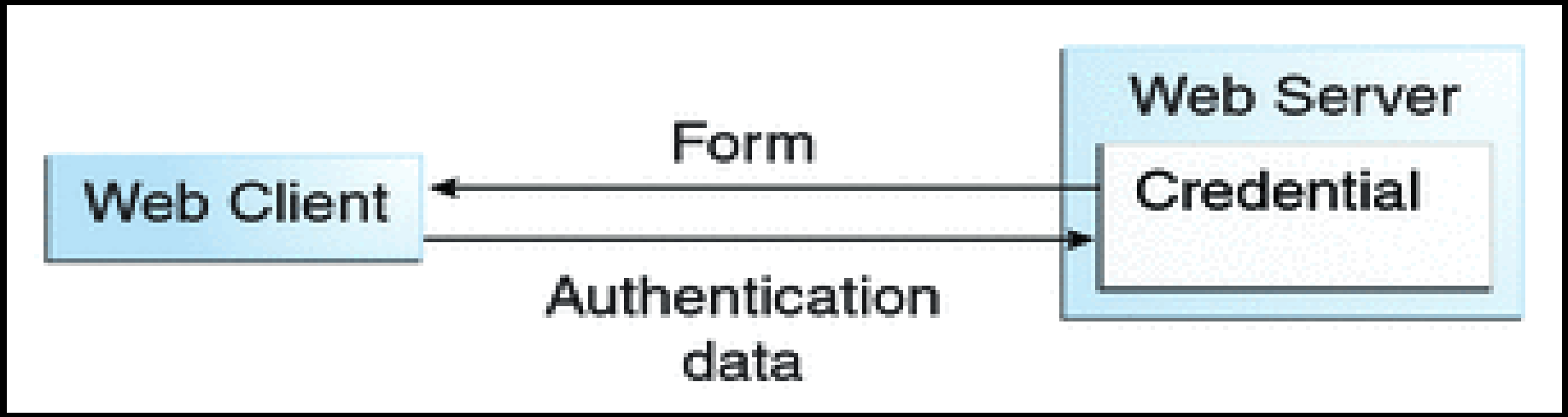
The web server returns a form that the web client uses to collect authentication data, such as user name and password, from the user.

The web client forwards the authentication data to the web server, where it is validated by the web server, as shown in Figure 39-2.

The validation mechanism may be local to a server or may leverage the underlying security services.

On the basis of the validation, the web server sets a credential for the user.

Figure 39-2 Initial Authentication



Step 3: URL Authorization

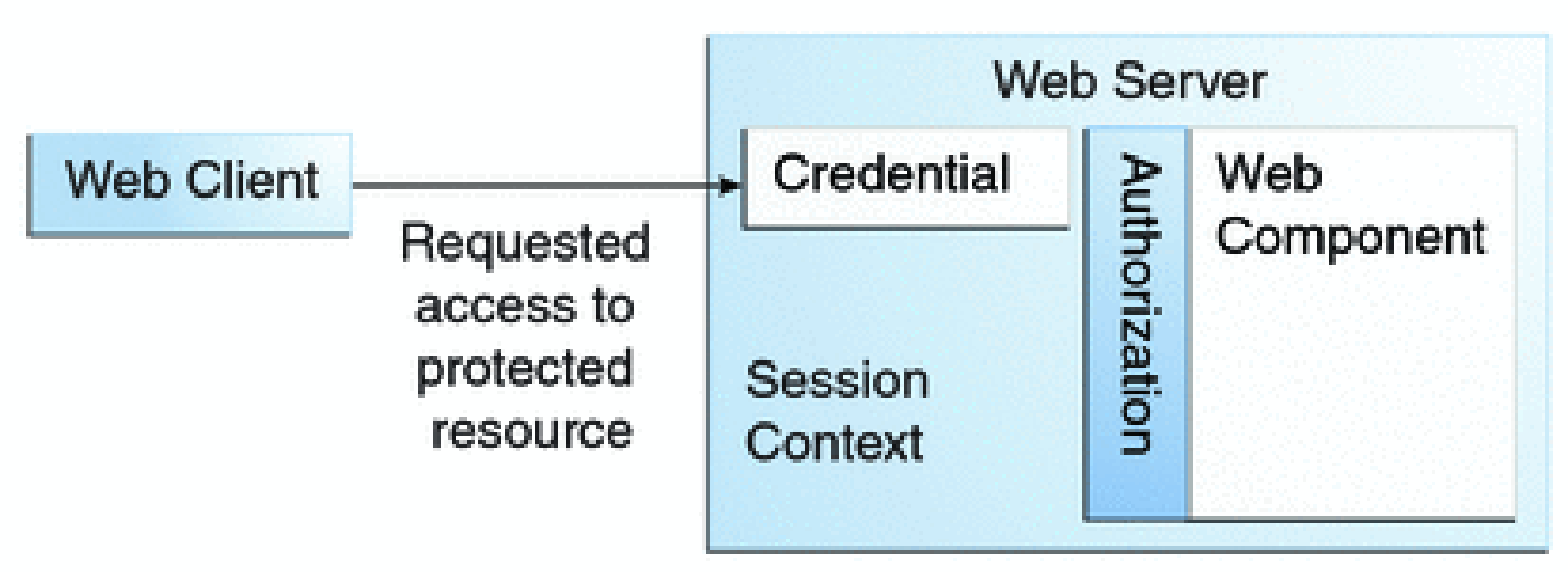
The credential is used for future determinations of whether the user is authorized to access restricted resources it may request.

The web server consults the security policy associated with the web resource to determine the security roles that are permitted access to the resource.

The security policy is derived from annotations or from the deployment descriptor.

The web container then tests the user's credential against each role to determine whether it can map the user to the role. Figure 39-3 shows this process.

Figure 39-3 URL Authorization



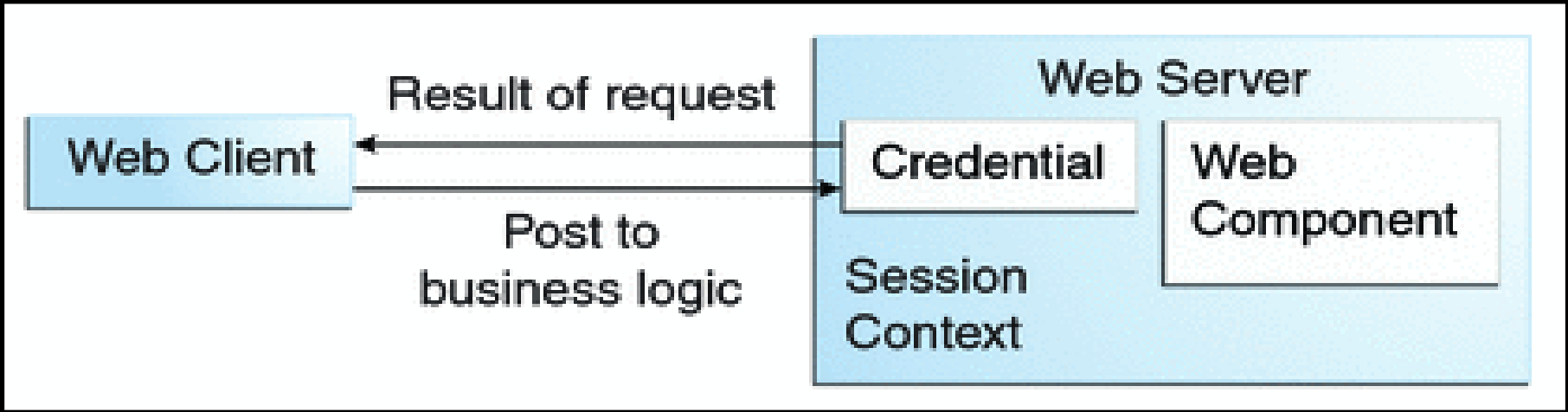
The web server's evaluation stops with an "is authorized" outcome when the web server is able to map the user to a role.

A "not authorized" outcome is reached if the web server is unable to map the user to any of the permitted roles.

Step 4: Fulfilling the Original Request

If the user is authorized, the web server returns the result of the original URL request, as shown in Figure 39-4.

Figure 39-4 Fulfilling the Original Request



In our example, the response URL of a web page is returned, enabling the user to post form data that needs to be handled by the business-logic component of the application.

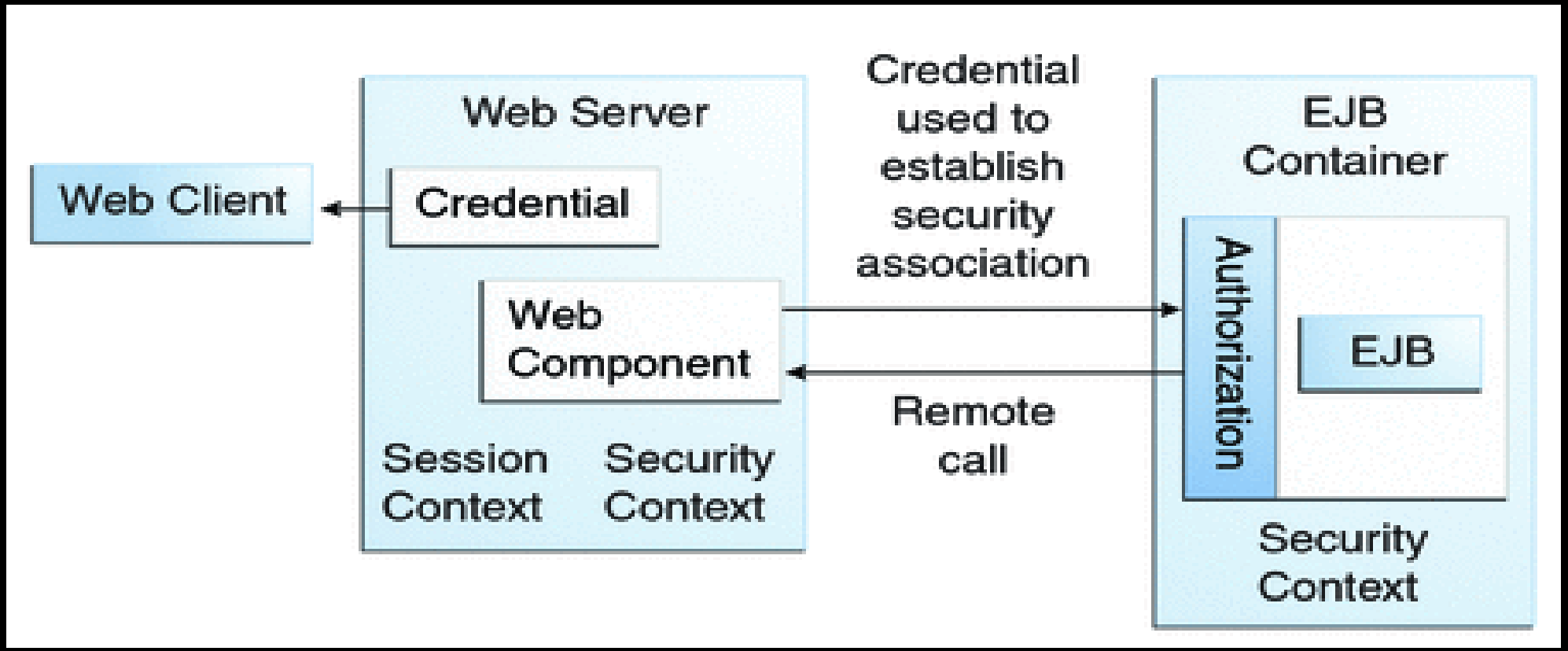
See Chapter 40, Getting Started Securing Web Applications for more information on protecting web applications.

Step 5: Invoking Enterprise Bean Business Methods

The web page performs the remote method call to the enterprise bean, using the user's credential to establish a secure association between the web page and the enterprise bean, as shown in Figure 39-5.

The association is implemented as two related security contexts: one in the web server and one in the EJB container.

Figure 39-5 Invoking an Enterprise Bean Business Method



The EJB container is responsible for enforcing access control on the enterprise bean method.

The container consults the security policy associated with the enterprise bean to determine the security roles that are permitted access to the method.

The security policy is derived from annotations or from the deployment descriptor.

For each role, the EJB container determines whether it can map the caller to the role by using the security context associated with the call.

The container's evaluation stops with an "is authorized" outcome when the container is able to map the caller's credential to a role.

A "not authorized" outcome is reached if the container is unable to map the caller to any of the permitted roles.

A "not authorized" result causes an exception to be thrown by the container and propagated back to the calling web page.

If the call is authorized, the container dispatches control to the enterprise bean method.

The result of the bean's execution of the call is returned to the web page and ultimately to the user by the web server and the web client.

Features of a Security Mechanism

A properly implemented security mechanism will provide the following functionality:

- Prevent unauthorized access to application functions and business or personal data (authentication)
- Hold system users accountable for operations they perform (non-repudiation)

- Protect a system from service interruptions and other breaches that affect quality of service

Ideally, properly implemented security mechanisms will also be

- Easy to administer
- Transparent to system users
- Interoperable across application and enterprise boundaries

Characteristics of Application Security

Java EE applications consist of components that can contain both protected and unprotected resources.

Often, you need to protect resources to ensure that only authorized users have access.

Authorization provides controlled access to protected resources. Authorization is based on identification and authentication.

Identification is a process that enables recognition of an entity by a system, and **authentication** is a process that verifies the identity of a user, device, or other entity in a computer system, usually as a prerequisite to allowing access to resources in a system.

Authorization and authentication are not required for an entity to access unprotected resources.

Accessing a resource without authentication is referred to as unauthenticated, or anonymous, access.

The characteristics of application security that, when properly addressed, help to minimize the security threats faced by an enterprise include the following:

- **Authentication:** The means by which communicating entities, such as client and server, prove to each other that they are acting on behalf of specific identities that are authorized for access.

This ensures that users are who they say they are.

- Authorization, or access control: The means by which interactions with resources are limited to collections of users or programs for the purpose of enforcing integrity, confidentiality, or availability constraints.

This ensures that users have permission to perform operations or access data.

- Data integrity: The means used to prove that information has not been modified by a third party, an entity other than the source of the information.

For example, a recipient of data sent over an open network must be able to detect and discard messages that were modified after they were sent.

This ensures that only authorized users can modify data.

- **Confidentiality, or data privacy: The means used to ensure that information is made available only to users who are authorized to access it.**

  This ensures that only authorized users can view sensitive data.

- **Non-repudiation: The means used to prove that a user who performed some action cannot reasonably deny having done so.**

  This ensures that transactions can be proved to have happened.

- **Quality of Service: The means used to provide better service to selected network traffic over various technologies.**

- **Auditing: The means used to capture a tamper-resistant record of security-related events for the purpose of being able to evaluate the effectiveness of security policies and mechanisms.**

To enable this, the system maintains a record of transactions and security information.

Security Mechanisms

The characteristics of an application should be considered when deciding the layer and type of security to be provided for applications.

The following sections discuss the characteristics of the common mechanisms that can be used to secure Java EE applications.

Each of these mechanisms can be used individually or with others to provide protection layers based on the specific needs of your implementation.

Java SE Security Mechanisms

Java SE provides support for a variety of security features and mechanisms:

- Java Authentication and Authorization Service (JAAS): JAAS is a set of APIs that enable services to authenticate and enforce access controls upon users.

JAAS provides a pluggable and extensible framework for programmatic user authentication and authorization.

JAAS is a core Java SE API and is an underlying technology for Java EE security mechanisms.

- **Java Generic Security Services (Java GSS-API): Java GSS-API is a token-based API used to securely exchange messages between communicating applications.**

  The GSS-API offers application programmers uniform access to security services atop a variety of underlying security mechanisms, including Kerberos.

- **Java Cryptography Extension (JCE): JCE provides a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code (MAC) algorithms.**

  Support for encryption includes symmetric, asymmetric, block, and stream ciphers.

  Block ciphers operate on groups of bytes; stream ciphers operate on one byte at a time.

The software also supports secure streams and sealed objects.

- Java Secure Sockets Extension (JSSE): JSSE provides a framework and an implementation for a Java version of the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols and includes functionality for data encryption, server authentication,

message integrity, and optional client authentication to enable secure Internet communications.

- Simple Authentication and Security Layer (SASL): SASL is an Internet standard (RFC 2222) that specifies a protocol for authentication and optional establishment of a security layer between client and server applications.

SASL defines how authentication data is to be exchanged but does not itself specify the contents of that data.

SASL is a framework into which specific authentication mechanisms that specify the contents and semantics of the authentication data can fit.

Java SE also provides a set of tools for managing keystores, certificates, and policy files; generating and verifying JAR signatures; and obtaining, listing, and managing Kerberos tickets.

For more information on Java SE security, visit http://download.oracle.com/javase/6/docs/technotes/guides/security/.

Java EE Security Mechanisms

Java EE security services are provided by the component container and can be implemented by using declarative or programmatic techniques (see Securing Containers).

Java EE security services provide a robust and easily configured security mechanism for authenticating users and authorizing access to application functions and associated data at many different layers.

Java EE security services are separate from the security mechanisms of the operating system.

Application-Layer Security

In Java EE, component containers are responsible for providing application-layer security, security services for a specific application type tailored to the needs of the application.

At the application layer, application firewalls can be used to enhance application protection by protecting the communication stream and all associated application resources from attacks.

Java EE security is easy to implement and configure and can offer fine-grained access control to application functions and data.

However, as is inherent to security applied at the application layer, security properties are not transferable to applications running in other environments and protect data only while it is residing in the application environment.

In the context of a traditional enterprise application, this is not necessarily a problem, but when applied to a web services application, in which data often travels across several intermediaries,

you would need to use the Java EE security mechanisms along with transport-layer security and message-layer security for a complete security solution.

The advantages of using application-layer security include the following.

- Security is uniquely suited to the needs of the application.

- Security is fine grained, with application-specific settings.

The disadvantages of using application-layer security include the following.

- The application is dependent on security attributes that are not transferable between application types.

- Support for multiple protocols makes this type of security vulnerable.

- Data is close to or contained within the point of vulnerability.

For more information on providing security at the application layer, see Securing Containers.

Transport-Layer Security

Transport-layer security is provided by the transport mechanisms used to transmit information over the wire between clients and providers; thus, transport-layer security relies on secure HTTP transport (HTTPS) using Secure Sockets Layer (SSL).

Transport security is a point-to-point security mechanism that can be used for authentication, message integrity, and confidentiality.

When running over an SSL-protected session, the server and client can authenticate each other and negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data.

Security is active from the time the data leaves the client until it arrives at its destination, or vice versa, even across intermediaries.

The problem is that the data is not protected once it gets to the destination.

One solution is to encrypt the message before sending.

Transport-layer security is performed in a series of phases, as follows.

- The client and server agree on an appropriate algorithm.
- A key is exchanged using public-key encryption and certificate-based authentication.
- A symmetric cipher is used during the information exchange.

Digital certificates are necessary when running HTTPS using SSL.

The HTTPS service of most web servers will not run unless a digital certificate has been installed.

Digital certificates have already been created for the GlassFish Server.

The advantages of using transport-layer security include the following.

- It is relatively simple, well-understood, standard technology.
- It applies to both a message body and its attachments.

The disadvantages of using transport-layer security include the following.

- It is tightly coupled with the transport-layer protocol.
- It represents an all-or-nothing approach to security.

This implies that the security mechanism is unaware of message contents, so that you cannot selectively apply security to portions of the message as you can with message-layer security.

- Protection is transient.

  The message is protected only while in transit.

Protection is removed automatically by the endpoint when it receives the message.

- It is not an end-to-end solution, simply point-to-point.

For more information on transport-layer security, see Establishing a Secure Connection Using SSL.

Message-Layer Security

In message-layer security, security information is contained within the SOAP message and/or SOAP message attachment, which allows security information to travel along with the message or attachment.

For example, a portion of the message may be signed by a sender and encrypted for a particular receiver.

When sent from the initial sender, the message may pass through intermediate nodes before reaching its intended receiver.

In this scenario, the encrypted portions continue to be opaque to any intermediate nodes and can be decrypted only by the intended receiver.

For this reason, message-layer security is also sometimes referred to as end-to-end security.

The advantages of message-layer security include the following.

- Security stays with the message over all hops and after the message arrives at its destination.

- Security can be selectively applied to different portions of a message and, if using XML Web Services Security, to attachments.

- Message security can be used with intermediaries over multiple hops.

- Message security is independent of the application environment or transport protocol.

The disadvantage of using message-layer security is that it is relatively complex and adds some overhead to processing.

The GlassFish Server supports message security using Metro, a web services stack that uses Web Services Security (WSS) to secure messages.

Because this message security is specific to Metro and is not a part of the Java EE platform, this tutorial does not discuss using WSS to secure messages.

See the *Metro User's Guide* at http://metro.java.net/guide/.

Securing Containers

In Java EE, the component containers are responsible for providing application security.

A container provides two types of security: declarative and programmatic.

Using Annotations to Specify Security Information

Annotations enable a declarative style of programming and so encompass both the declarative and programmatic security concepts.

Users can specify information about security within a class file by using annotations.

The GlassFish Server uses this information when the application is deployed.

Not all security information can be specified by using annotations, however.

Some information must be specified in the application deployment descriptors.

Specific annotations that can be used to specify security information within an enterprise bean class file are described in Securing an Enterprise Bean Using Declarative Security.

Chapter 40, Getting Started Securing Web Applications, describes how to use annotations to secure web applications where possible.

Deployment descriptors are described only where necessary.

For more information on annotations, see Further Information about Security.

Using Deployment Descriptors for Declarative Security

Declarative security can express an application component's security requirements by using deployment descriptors.

Because deployment descriptor information is declarative, it can be changed without the need to modify the source code.

At runtime, the Java EE server reads the deployment descriptor and acts upon the corresponding application, module, or component accordingly.

Deployment descriptors must provide certain structural information for each component if this information has not been provided in annotations or is not to be defaulted.

This part of the tutorial does not document how to create deployment descriptors; it describes only the elements of the deployment descriptor relevant to security.

NetBeans IDE provides tools for creating and modifying deployment descriptors.

Different types of components use different formats, or schemas, for their deployment descriptors.

The security elements of deployment descriptors discussed in this tutorial include the following.

- Web components may use a web application deployment descriptor named `web.xml`.

The schema for web component deployment descriptors is provided in Chapter 14 of the Java Servlet 3.0 specification (JSR 315), which can be downloaded from http://jcp.org/en/jsr/detail?id=315.

- Enterprise JavaBeans components may use an EJB deployment descriptor named `META-INF/ejb-jar.xml`, contained in the EJB JAR file.

The schema for enterprise bean deployment descriptors is provided in Chapter 19 of the EJB 3.1 specification (JSR 318), which can be downloaded from http://jcp.org/en/jsr/detail?id=318.

Using Programmatic Security

Programmatic security is embedded in an application and is used to make security decisions.

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

The API for programmatic security consists of methods of the `EJBContext` interface and the `HttpServletRequest` interface.

These methods allow components to make business-logic decisions based on the security role of the caller or remote user.

Programmatic security is discussed in more detail in the following sections:

- Using Programmatic Security with Web Applications
- Securing an Enterprise Bean Programmatically

Securing the GlassFish Server

This tutorial describes deployment to the GlassFish Server, which provides highly secure, interoperable, and distributed component computing based on the Java EE security model.

GlassFish Server supports the Java EE 6 security model.

You can configure GlassFish Server for the following purposes:

- Adding, deleting, or modifying authorized users.

For more information on this topic, see Working with Realms, Users, Groups, and Roles.

- Configuring secure HTTP and Internet Inter-Orb Protocol (IIOP) listeners.

- Configuring secure Java Management Extensions (JMX) connectors.

- Adding, deleting, or modifying existing or custom realms.

- Defining an interface for pluggable authorization providers using Java Authorization Contract for Containers (JACC).

  JACC defines security contracts between the GlassFish Server and authorization policy modules.

These contracts specify how the authorization providers are installed, configured, and used in access decisions.

- Using pluggable audit modules.

- Customizing authentication mechanisms.

All implementations of Java EE 6 compatible Servlet containers are required to support the Servlet Profile of JSR 196, which offers an avenue for customizing the authentication mechanism applied by the web container on behalf of one or more applications.

- Setting and changing policy permissions for an application.

Working with Realms, Users, Groups, and Roles

You often need to protect resources to ensure that only authorized users have access.

See Characteristics of Application Security for an introduction to the concepts of authentication, identification, and authorization.

This section discusses setting up users so that they can be correctly identified and either given access to protected resources or denied access if they are not authorized to access the protected resources.

To authenticate a user, you need to follow these basic steps.

1. The application developer writes code to prompt for a user name and password.

The various methods of authentication are discussed in Specifying an Authentication Mechanism in the Deployment Descriptor.

2. The application developer communicates how to set up security for the deployed application by use of a metadata annotation or deployment descriptor.

   This step is discussed in Setting Up Security Roles.

3. The server administrator sets up authorized users and groups on the GlassFish Server.

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

4. The application deployer maps the application's security roles to users, groups, and principals defined on the GlassFish Server.

This topic is discussed in Mapping Roles to Users and Groups.

What Are Realms, Users, Groups, and Roles?

A realm is a security policy domain defined for a web or application server.

A realm contains a collection of users, who may or may not be assigned to a group.

Managing users on the GlassFish Server is discussed in Managing Users and Groups on the GlassFish Server.

An application will often prompt for a user name and password before allowing access to a protected resource.

After the user name and password have been entered, that information is passed to the server, which either authenticates the user and sends the protected resource or does not authenticate the user, in which case access to the protected resource is denied.

This type of user authentication is discussed in Specifying an Authentication Mechanism in the Deployment Descriptor.

In some applications, authorized users are assigned to roles.

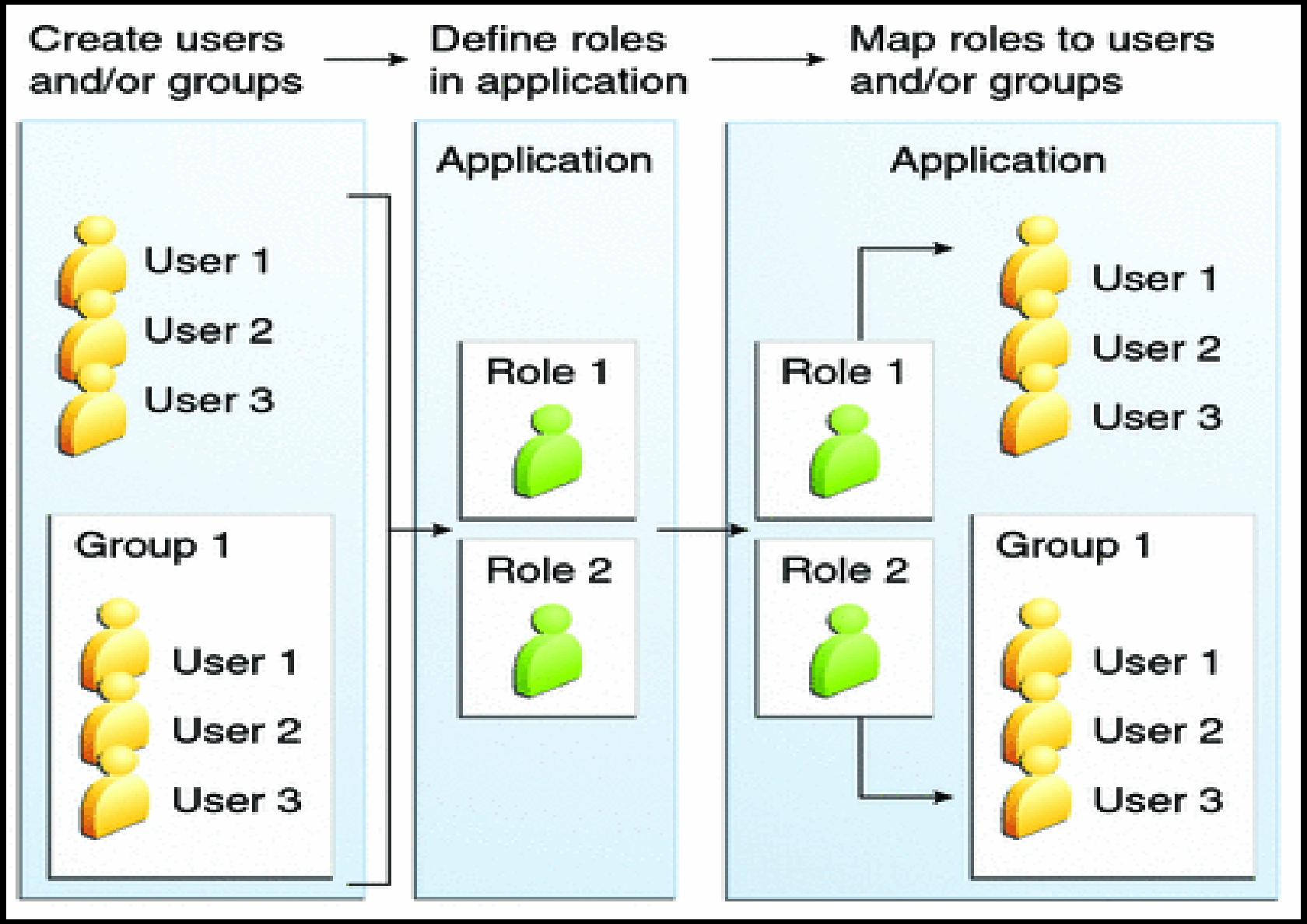
In this situation, the role assigned to the user in the application must be mapped to a principal or group defined on the application server.

Figure 39-6 shows this.

More information on mapping roles to users and groups can be found in Setting Up Security Roles.

The following sections provide more information on realms, users, groups, and roles.

Figure 39-6 Mapping Roles to Users and Groups



What Is a Realm?

A realm is a security policy domain defined for a web or application server.

The protected resources on a server can be partitioned into a set of protection spaces, each with its own authentication scheme and/or authorization database containing a collection of users and groups.

For a web application, a realm is a complete database of users and groups identified as valid users of a web application or a set of web applications and controlled by the same authentication policy.

The Java EE server authentication service can govern users in multiple realms.

The `file`, `admin-realm`, and `certificate` realms come preconfigured for the GlassFish Server.

In the `file` realm, the server stores user credentials locally in a file named `keyfile`.

You can use the Administration Console to manage users in the `file` realm.

When using the `file` realm, the server authentication service verifies user identity by checking the `file` realm.

This realm is used for the authentication of all clients except for web browser clients that use HTTPS and certificates.

In the `certificate` realm, the server stores user credentials in a certificate database.

When using the `certificate` realm, the server uses certificates with HTTPS to authenticate web clients.

To verify the identity of a user in the `certificate` realm, the authentication service verifies an X.509 certificate.

For step-by-step instructions for creating this type of certificate, see Working with Digital Certificates.

The common name field of the X.509 certificate is used as the principal name.

The `admin-realm` is also a `file` realm and stores administrator user credentials locally in a file named `admin-keyfile`.

You can use the Administration Console to manage users in this realm in the same way you manage users in the `file` realm.

For more information, see Managing Users and Groups on the GlassFish Server.

What Is a User?

A user is an individual or application program identity that has been defined in the GlassFish Server.

In a web application, a user can have associated with that identify a set of roles that entitle the user to access all resources protected by those roles.

Users can be associated with a group.

A Java EE user is similar to an operating system user.

Typically, both types of users represent people.

However, these two types of users are not the same.

The Java EE server authentication service has no knowledge of the user name and password you provide when you log in to the operating system.

The Java EE server authentication service is not connected to the security mechanism of the operating system.

The two security services manage users that belong to different realms.

What Is a Group?

A **group** is a set of authenticated users, classified by common traits, defined in the GlassFish Server.

A Java EE user of the `file` realm can belong to a group on the GlassFish Server.

(A user in the `certificate` realm cannot.)

A group on the GlassFish Server is a category of users classified by common traits, such as job title or customer profile.

For example, most customers of an e-commerce application might belong to the `CUSTOMER` group, but the big spenders would belong to the `PREFERRED` group.

Categorizing users into groups makes it easier to control the access of large numbers of users.

A group on the GlassFish Server has a different scope from a role.

A group is designated for the entire GlassFish Server, whereas a role is associated only with a specific application in the GlassFish Server.

What Is a Role?

A role is an abstract name for the permission to access a particular set of resources in an application.

A role can be compared to a key that can open a lock.

Many people might have a copy of the key.

The lock doesn't care who you are, only that you have the right key.

Some Other Terminology

The following terminology is also used to describe the security requirements of the Java EE platform:

- **Principal: An entity that can be authenticated by an authentication protocol in a security service that is deployed in an enterprise.**

A principal is identified by using a principal name and authenticated by using authentication data.

- Security policy domain, also known as security domain or realm: A scope over which a common security policy is defined and enforced by the security administrator of the security service.

- **Security attributes:** A set of attributes associated with every principal.

  The security attributes have many uses: for example, access to protected resources and auditing of users.

  Security attributes can be associated with a principal by an authentication protocol.

- **Credential: An object that contains or references security attributes used to authenticate a principal for Java EE services.**

  A principal acquires a credential upon authentication or from another principal that allows its credential to be used.

Managing Users and Groups on the GlassFish Server

Follow these steps for managing users before you run the tutorial examples.

To Add Users to the GlassFish Server

1. Start the GlassFish Server, if you haven't already done so.

Information on starting the GlassFish Server is available in Starting and Stopping the GlassFish Server.

2. Start the Administration Console, if you haven't already done so.

To start the Administration Console, open a web browser and specify the URL `http://localhost:4848/`.

If you changed the default Admin port during installation, type the correct port number in place of `4848`.

3. In the navigation tree, expand the Configurations node, then expand the server-config node.

4. Expand the Security node.

5. Expand the Realms node.

6. Select the realm to which you are adding users.

   - Select the `file` realm to add users you want to access applications running in this realm.

     For the example security applications, select the `file` realm.

The Edit Realm page opens.

- Select the `admin-realm` to add users you want to enable as system administrators of the GlassFish Server.

The Edit Realm page opens.

7. You cannot add users to the `certificate` realm by using the Administration Console.

   In the `certificate` realm, you can add only certificates.

   For information on adding (importing) certificates to the `certificate` realm, see Adding Users to the Certificate Realm.

8. On the Edit Realm page, click the Manage Users button.

The File Users or Admin Users page opens.

9. On the File Users or Admin Users page, click New to add a new user to the realm.

The New File Realm User page opens.

10. Type values in the User ID, Group List, New Password, and Confirm New Password fields.

For the Admin Realm, the Group List field is read-only, and the group name is `asadmin`.

Restart the GlassFish Server and Administration Console after you add a user to the Admin Realm.

For more information on these properties, see Working with Realms, Users, Groups, and Roles.

For the example security applications, specify a user with any name and password you like, but make sure that the user is assigned to the group `TutorialUser`.

The user name and password are case-sensitive.

Keep a record of the user name and password for working with the examples later in this tutorial.

11. Click OK to add this user to the realm, or click Cancel to quit without saving.

Adding Users to the Certificate Realm

In the `certificate` realm, user identity is set up in the GlassFish Server security context and populated with user data obtained from cryptographically verified client certificates.

For step-by-step instructions for creating this type of certificate, see Working with Digital Certificates.

Setting Up Security Roles

When you design an enterprise bean or web component, you should always think about the kinds of users who will access the component.

For example, a web application for a human resources department might have a different request URL for someone who has been assigned the role of `DEPT_ADMIN` than for someone

who has been assigned the role of `DIRECTOR`.

The `DEPT_ADMIN` role may let you view employee data, but the `DIRECTOR` role enables you to modify employee data, including salary data.

Each of these security roles is an abstract logical grouping of users that is defined by the person who assembles the application.

When an application is deployed, the deployer will map the roles to security identities in the operational environment, as shown in Figure 39-6.

For Java EE components, you define security roles using the `@DeclareRoles` and `@RolesAllowed` metadata annotations.

The following is an example of an application in which the role of `DEPT-ADMIN` is authorized for methods that review employee payroll data, and the role of `DIRECTOR` is authorized for methods that change employee payroll data.

The enterprise bean would be annotated as shown in the following code:

```
import javax.annotation.security.
DeclareRoles;
import javax.annotation.security.
RolesAllowed;
...
@DeclareRoles
({"DEPT-ADMIN", "DIRECTOR"})
@Stateless public class PayrollBean
implements Payroll {
@Resource SessionContext ctx;
@RolesAllowed("DEPT-ADMIN")
```

```
public void reviewEmployeeInfo
(EmplInfo info){
oldInfo = ...
read from database; // ...
}
@RolesAllowed("DIRECTOR")
public void
updateEmployeeInfo(EmplInfo info){
newInfo = ...
update database; // ...
 } ... }
```

For a servlet, you can use the `@HttpConstraint` annotation within the `@ServletSecurity` annotation to specify the roles that are allowed to access the servlet.

For example, a servlet might be annotated as follows:

```
@WebServlet
(name = "PayrollServlet",
urlPatterns = {"/payroll"})
@ServletSecurity(
@HttpConstraint
(transportGuarantee =
TransportGuarantee.CONFIDENTIAL,
rolesAllowed =
{"DEPT-ADMIN", "DIRECTOR"}))
public class GreetingServlet
extends HttpServlet {
```

These annotations are discussed in more detail in Specifying Security for Basic Authentication Using Annotations and Securing an Enterprise Bean Using Declarative Security.

After users have provided their login information and the application has declared what roles are authorized to access protected parts of an application, the next step is to map the security role to the name of a user, or principal.

Mapping Roles to Users and Groups

When you are developing a Java EE application, you don't need to know what categories of users have been defined for the realm in which the application will be run.

In the Java EE platform, the security architecture provides a mechanism for mapping the roles defined in the application to the users or groups defined in the runtime realm.

The role names used in the application are often the same as the group names defined on the GlassFish Server.

Under these circumstances, you can enable a default principal-to-role mapping on the GlassFish Server by using the Administration Console.

The task To Set Up Your System for Running the Security Examples explains how to do this.

All the tutorial security examples use default principal-to-role mapping.

If the role names used in an application are not the same as the group names defined on the server, use the runtime deployment descriptor to specify the mapping.

The following example demonstrates how to do this mapping in the `glassfish-web.xml` file, which is the file used for web applications:

```
<glassfish-web-app>
...
<security-role-mapping>
<role-name>Mascot</role-name>
<principal-name>
Duke
</principal-name>
</security-role-mapping>
<security-role-mapping>
<role-name>Admin</role-name>
<group-name>Director</group-name>
```

```
</security-role-mapping>
...
</glassfish-web-app>
```

A role can be mapped to specific principals, specific groups, or both.

The principal or group names must be valid principals or groups in the current default realm or in the realm specified in the `login-config` element.

In this example, the role of `Mascot` used in the application is mapped to a principal, named `Duke`, that exists on the application server.

Mapping a role to a specific principal is useful when the person occupying that role may change.

For this application, you would need to modify only the runtime deployment descriptor rather than search and replace throughout the application for references to this principal.

Also in this example, the role of `Admin` is mapped to a group of users assigned the group name of `Director`.

This is useful because the group of people authorized to access director-level administrative data has to be maintained only on the GlassFish Server.

The application developer does not need to know who these people are, but only needs to define the group of people who will be given access to the information.

The `role-name` must match the `role-name` in the `security-role` element of the corresponding deployment descriptor or the role name defined in a `@DeclareRoles` annotation.

Establishing a Secure Connection Using SSL

Secure Socket Layer (SSL) technology is security that is implemented at the transport layer (see Transport-Layer Security for more information about transport-layer security).

SSL allows web browsers and web servers to communicate over a secure connection.

In this secure connection, the data is encrypted before being sent and then is decrypted upon receipt and before processing.

Both the browser and the server encrypt all traffic before sending any data.

SSL addresses the following important security considerations:

- **Authentication:** During your initial attempt to communicate with a web server over a secure connection, that server will present your web browser with a set of credentials in the form of a server certificate.

  The purpose of the certificate is to verify that the site is who and what it claims to be.

In some cases, the server may request a certificate proving that the client is who and what it claims to be; this mechanism is known as client authentication.

- Confidentiality: When data is being passed between the client and the server on a network, third parties can view and intercept this data.

SSL responses are encrypted so that the data cannot be deciphered by the third party and the data remains confidential.

- Integrity: When data is being passed between the client and the server on a network, third parties can view and intercept this data.

SSL helps guarantee that the data will not be modified in transit by that third party.

The SSL protocol is designed to be as efficient as securely possible.

However, encryption and decryption are computationally expensive processes from a performance standpoint.

It is not strictly necessary to run an entire web application over SSL, and it is customary for a developer to decide which pages require a secure connection and which do not.

Pages that might require a secure connection include those for login, personal information, shopping cart checkouts, or credit card information transmittal.

Any page within an application can be requested over a secure socket by simply prefixing the address with `https:` instead of `http:`.

Any pages that absolutely require a secure connection should check the protocol type associated with the page request and take the appropriate action if `https:` is not specified.

Using name-based virtual hosts on a secured connection can be problematic.

This is a design limitation of the SSL protocol itself.

The SSL handshake, whereby the client browser accepts the server certificate, must occur before the HTTP request is accessed.

As a result, the request information containing the virtual host name cannot be determined before authentication, and it is therefore not possible to assign multiple certificates to a single IP address.

If all virtual hosts on a single IP address need to authenticate against the same certificate, the addition of multiple virtual hosts should not interfere with normal SSL operations on the server.

Be aware, however, that most client browsers will compare the server's domain name against the domain name listed in the certificate, if any; this is applicable primarily to official certificates signed by a certificate authority (CA).

If the domain names do not match, these browsers will display a warning to the client.

In general, only address-based virtual hosts are commonly used with SSL in a production environment.

Verifying and Configuring SSL Support

As a general rule, you must address the following issues to enable SSL for a server:

- There must be a `Connector` element for an SSL connector in the server deployment descriptor.

- There must be valid keystore and certificate files.

- The location of the keystore file and its password must be specified in the server deployment descriptor.

An SSL HTTPS connector is already enabled in the GlassFish Server.

For testing purposes and to verify that SSL support has been correctly installed, load the default introduction page with a URL that connects to the port defined in the server deployment descriptor:

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

The `https` in this URL indicates that the browser should be using the SSL protocol.

The `localhost` in this example assumes that you are running the example on your local machine as part of the development process.

The `8181` in this example is the secure port that was specified where the SSL connector was created.

If you are using a different server or port, modify this value accordingly.

The first time that you load this application, the New Site Certificate or Security Alert dialog box appears.

Select Next to move through the series of dialog boxes, and select Finish when you reach the last dialog box. The certificates will display only the first time.

When you accept the certificates, subsequent hits to this site assume that you still trust the content.

Working with Digital Certificates

Digital certificates for the GlassFish Server have already been generated and can be found in the directory *as-install*`/`*domain-dir*`/config/`.

These digital certificates are self-signed and are intended for use in a development environment; they are not intended for production purposes.

For production purposes, generate your own certificates and have them signed by a CA.

To use SSL, an application or web server must have an associated certificate for each external interface, or IP address, that accepts secure connections.

The theory behind this design is that a server should provide some kind of reasonable assurance that its owner is who you think it is, particularly before receiving any sensitive information.

It may be useful to think of a certificate as a "digital driver's license" for an Internet address.

The certificate states with which company the site is associated, along with some basic contact information about the site owner or administrator.

The digital certificate is cryptographically signed by its owner and is difficult for anyone else to forge.

For sites involved in e-commerce or in any other business transaction in which authentication of identity is important, a certificate can be purchased from a well-known CA such as VeriSign or Thawte.

If your server certificate is self-signed, you must install it in the GlassFish Server keystore file (`keystore.jks`).

If your client certificate is self-signed, you should install it in the GlassFish Server truststore file (`cacerts.jks`).

Sometimes, authentication is not really a concern.

For example, an administrator might simply want to ensure that data being transmitted and received by the server is private and cannot be snooped by anyone eavesdropping on the connection.

In such cases, you can save the time and expense involved in obtaining a CA certificate and simply use a self-signed certificate.

SSL uses public-key cryptography, which is based on key pairs.

Key pairs contain one public key and one private key.

Data encrypted with one key can be decrypted only with the other key of the pair.

This property is fundamental to establishing trust and privacy in transactions.

For example, using SSL, the server computes a value and encrypts it by using its private key.

The encrypted value is called a digital signature.

The client decrypts the encrypted value by using the server's public key and compares the value to its own computed value.

If the two values match, the client can trust that the signature is authentic, because only the private key could have been used to produce such a signature.

Digital certificates are used with HTTPS to authenticate web clients.

The HTTPS service of most web servers will not run unless a digital certificate has been installed.

Use the procedure outlined in the next section, Creating a Server Certificate, to set up a digital certificate that can be used by your application or web server to enable SSL.

One tool that can be used to set up a digital certificate is `keytool`, a key and certificate management utility that ships with the JDK.

This tool enables users to administer their own public/private key pairs and associated certificates for use in self-authentication, whereby the user authenticates himself or herself to other users or services, or data integrity and authentication services, using digital signatures.

The tool also allows users to cache the public keys, in the form of certificates, of their communicating peers.

For a better understanding of `keytool` and public-key cryptography, see the `keytool` documentation at http://download.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html.

Creating a Server Certificate

A server certificate has already been created for the GlassFish Server and can be found in the *domain-dir*`/config/` directory.

The server certificate is in `keystore.jks`.

The `cacerts.jks` file contains all the trusted certificates, including client certificates.

If necessary, you can use `keytool` to generate certificates.

The `keytool` utility stores the keys and certificates in a file termed a keystore, a repository of certificates used for identifying a client or a server.

Typically, a keystore is a file that contains one client's or one server's identity.

The keystore protects private keys by using a password.

If you don't specify a directory when specifying the keystore file name, the keystores are created in the directory from which the `keytool` command is run.

This can be the directory where the application resides, or it can be a directory common to many applications.

The general steps for creating a server certificate are as follows.

1. Create the keystore.
2. Export the certificate from the keystore.
3. Sign the certificate.
4. Import the certificate into a truststore: a repository of certificates used for verifying the certificates.

A truststore typically contains more than one certificate.

To Use `keytool` to Create a Server Certificate provides specific information on using the `keytool` utility to perform these steps.

To Use `keytool` to Create a Server Certificate

Run `keytool` to generate a new key pair in the default development keystore file, `keystore.jks`.

This example uses the alias `server-alias` to generate a new public/private key pair and wrap the public key into a self-signed certificate inside `keystore.jks`.

The key pair is generated by using an algorithm of type RSA, with a default password of `changeit`.

For more information and other examples of creating and managing keystore files, read the `keytool` online help at http://download.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html.

Note - RSA is public-key encryption technology developed by RSA Data Security, Inc.

From the directory in which you want to create the key pair, run `keytool` as shown in the following steps.

1. **Generate the server certificate.**

Type the `keytool` command all on one line:

```
java-home/bin/keytool -genkey -
alias server-alias -keyalg RSA -
keypass changeit
-storepass changeit -keystore
keystore.jks
```

When you press Enter, `keytool` prompts you to enter the server name, organizational unit, organization, locality, state, and country code.

You must type the server name in response to `keytool`'s first prompt, in which it asks for first and last names.

For testing purposes, this can be `localhost`.

When you run the example applications, the host (server name) specified in the keystore must match the host identified in the `javaee.server.name` property specified in the file

tut-install`/examples/bp-project/build.properties`

(by default, this is `localhost`).

2. Export the generated server certificate in `keystore.jks` into the file `server.cer`.

Type the `keytool` command all on one line:

```
java-home/bin/keytool -export -
alias server-alias -storepass
changeit
-file server.cer -keystore
keystore.jks
```

3. If you want to have the certificate signed by a CA, read the example at http://download.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html.

To add the server certificate to the truststore file, `cacerts.jks`, run `keytool` from the directory where you created the keystore and server certificate.

Use the following parameters:

```
java-home/bin/keytool -import -v
-trustcacerts -alias server-alias
-file server.cer -keystore
cacerts.jks -keypass changeit -
storepass changeit
```

Information on the certificate, such as that shown next, will appear:

```
Owner: CN=localhost,
OU=Sun Micro, O=Docs,
L=Santa Clara, ST=CA,
C=USIssuer: CN=localhost,
OU=Sun Micro, O=Docs,
L=Santa Clara, ST=CA,
C=USSerial number: 3e932169Valid
from: Tue Apr 08Certificate
fingerprints:MD5:
52:9F:49:68:ED:78:6F:39:87:F3:98:
B3:6A:6B:0F:90 SHA1:
```

```
EE:2E:2A:A6:9E:03:9A:3A:1C:17:4A:
28:5E:97:20:78:3F:
Trust this certificate? [no]:
```

4. Type `yes`, then press the `Enter` or `Return` key.

The following information appears:

```
Certificate was added to
keystore[Saving cacerts.jks]
```

Further Information about Security

For more information about security in Java EE applications, see

- Java EE 6 specification:

  http://jcp.org/en/jsr/detail?id=316

- Enterprise JavaBeans 3.1 specification:

  http://jcp.org/en/jsr/detail?id=318

- Implementing Enterprise Web Services 1.3 specification:

  http://jcp.org/en/jsr/detail?id=109

- Java SE security information:

  http://download.oracle.com/javase/6/docs/technotes/guides/security/

- Java Servlet 3.0 specification:

  http://jcp.org/en/jsr/detail?id=315

- Java Authorization Contract for Containers 1.3 specification:

  http://jcp.org/en/jsr/detail?id=115

- *Java Authentication and Authorization Service (JAAS) Reference Guide:*

  http://download.oracle.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html

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

  http://download.oracle.com/javase/6/docs/technotes/guides/security/jaas/JAASLMDevGuide.html