BEA WebLogic Server™

Programming WebLogic Security

Version 8.1
Revised: August 2005

Contents

About This Document

This document is organized as follows:

- Chapter 1, "Introduction to Programing WebLogic Security," discusses the audiences of this document, the need for security, and the WebLogic Security application programming Interfaces (APIs).
- Chapter 2, "Securing Web Applications," describes how to implement security in Web applications.
- Chapter 3, "Using JAAS Authentication in Java Clients," describes how to implement JAAS authentication in Java clients.
- Chapter 4, "Using SSL Authentication in Java Clients," describes how to implement SSL and digital certificate authentication in Java clients.
- Chapter 5, "Securing Enterprise JavaBeans (EJBs)," describes how to implement security in Enterprise JavaBeans.
- Chapter 6, "Using Network Connection Filters," describes how to implement network connection filters.
- Chapter 7, "Using Java Security to Protect WebLogic Resources," discusses using Java security to protect WebLogic resources.
- Appendix A, "Deprecated Security APIs," provides a list of `weblogic.security` packages in which APIs have been deprecated.

Note: This document does not provide instructions on how to configure WebLogic Security providers and custom security providers. For information on configuring WebLogic security providers and custom security providers, see *Managing WebLogic Security*.

Note: This document is not intended for developers who want to write custom security providers for use with WebLogic Server. It does not describe how to write custom security providers. For information on developing custom security providers, see *Developing Security Providers for WebLogic Server*.

Audience for This Guide

This document is intended for the following audiences:

- Application Developers

  Developers who are Java programmers that focus on developing client applications, adding security to Web applications and Enterprise JavaBeans (EJBs). They work with other engineering, Quality Assurance (QA), and database teams to implement security features. Application Developers have in-depth/working knowledge of Java (including J2EE components such as servlets/JSPs and JSEE) and Java security.

  Application developers use the WebLogic security and Java 2 security application programming interfaces (APIs) to secure their applications. Therefore, this document provides instructions for using those APIs for securing Web applications, Java applications, and Enterprise JavaBeans (EJBs).

- Security Developers

  Developers who focus on defining the system architecture and infrastructure for security products that integrate into WebLogic Server and on developing custom security providers for use with WebLogic Server. They work with Application Architects to ensure that the security architecture is implemented according to design and that no security holes are introduced. They also work with Server Administrators to ensure that security is properly configured. Security Developers have a solid understanding of security concepts, including authentication, authorization, auditing (AAA), in-depth knowledge of Java (including Java Management eXtensions (JMX), and working knowledge of WebLogic Server and security provider functionality.

  Security developers use the Security Service Provider Interfaces (SSPIs) to develop custom security providers for use with WebLogic Server, however, this document does not address this task. For information on how to use the SSPIs to develop custom security providers, see *Developing Security Providers for WebLogic Server*.

- Server Administrators

Administrators who work closely with Application Architects to design a security scheme for the server and the applications running on the server, to identify potential security risks, and to propose configurations that prevent security problems. Related responsibilities may include maintaining critical production systems, configuring and managing security realms, implementing authentication and authorization schemes for server and application resources, upgrading security features, and maintaining security provider databases. Server Administrators have in-depth knowledge of the Java security architecture, including Web application and EJB security, Public Key security, and SSL.

- Application Administrators

  Administrators who work with Server Administrators to implement and maintain security configurations and authentication and authorization schemes, and to set up and maintain access to deployed application resources in defined security realms. Application Administrators have general knowledge of security concepts and the Java Security architecture. They understand Java, XML, deployment descriptors, and can identify security events in server and audit logs.

  While administrators typically use the Administration Console to deploy, configure, and manage applications when they put the applications into production, application developers may also use the Administration Console to test their applications before they are put into production. At a minimum, testing requires that applications be deployed and configured. This document does not cover some aspects of administration as it relates to security, rather, it references *Managing WebLogic Security*, *Securing WebLogic Resources*, and *Administration Console Online Help* for descriptions of how to use the Administration Console to perform security tasks.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation.

How to Print the Document

You can print a copy of this document from a Web browser, one main topic at a time, by using the File→Print option on your Web browser.

A PDF version of this document is available on the WebLogic Server documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Server documentation Home page, click Download Documentation, and select the document you want to print.

Adobe Acrobat Reader is available at no charge from the Adobe Web site at http://www.adobe.com.

Related Information

In addition to this document, *Programming WebLogic Security*, the following documents provide information on the WebLogic Security Service:

- *Introduction to WebLogic Security*—This document summarizes the features of the WebLogic Security Service and presents an overview of the architecture and capabilities of the WebLogic Security Service. It is the starting point for understanding the WebLogic Security Service.
- *Securing a Production Environment*—This document highlights essential security measures for you to consider before you deploy WebLogic Server into a production environment.
- *Developing Security Providers for WebLogic Server*—This document provides security vendors and application developers with the information needed to develop custom security providers that can be used with WebLogic Server.
- *Managing WebLogic Security*—This document explains how to configure security for WebLogic Server and how to use Compatibility security.
- *Securing WebLogic Resources*—This document introduces the various types of WebLogic resources, and provides information that allows you to secure these resources using WebLogic Server.

- *WebLogic Server 8.1 Upgrade Guide*—This document provides procedures and other information you need to upgrade 6.x and earlier versions of WebLogic Server to WebLogic Server 8.1. It also provides information about moving applications from a 6.x or earlier version of WebLogic Server to 8.1. For specific information on upgrading WebLogic Server security, see *Security* in the *WebLogic Server 8.1 Upgrade Guide*.
- *Administration Console Online Help—*This document describes how to use the Administration Console to perform security tasks.
- *Security FAQ*—This document gives answers to frequently asked questions about WebLogic Server security.
- Javadocs for WebLogic Classes—This document includes reference documentation for the WebLogic security packages that are provided with and supported by the WebLogic Server 7.0 software.

Contact Us!

Your feedback on BEA documentation is important to us. Send us e-mail at docsupport@bea.com if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the documentation.

In your e-mail message, please indicate the software name and version you are using, as well as the title and document date of your documentation. If you have any questions about this version of BEA WebLogic Server, or if you have problems installing and running BEA WebLogic Server, contact BEA Customer Support through BEA WebSupport at http://www.bea.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Usage
Ctrl+Tab	Keys you press simultaneously.
italics	Emphasis and book titles.

`monospace text`	Code samples, commands and their options, Java classes, data types, directories, and filenames and their extensions. Monospace text also indicates text that you enter from the keyboard.
	Examples:
	`import java.util.Enumeration;`
	`chmod u+w *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

`monospace italic text`	Variables in code. *Example*: `String` *`CustomerName;`*
UPPERCASE TEXT	Device names, environment variables, and logical operators. *Example*s: LPT1 BEA_HOME OR
`{ }`	A set of choices in a syntax line.

`[ ]`	Optional items in a syntax line. *Example*:
	`java utils.MulticastTest -n` *`name`* `-a` *`address`*
	`[-p` *`portnumber`*`] [-t` *`timeout`*`] [-s` *`send`*`]`

Convention	Usage

`|`	Separates mutually exclusive choices in a syntax line. *Example*:
	
	`java weblogic.deploy [list|deploy|undeploy|update]`
	`     password {application} {source}`

`...`	Indicates one of the following in a command line: ■ An argument can be repeated several times in the command line. ■ The statement omits additional optional arguments. ■ You can enter additional parameters, values, or other information
. . .	Indicates the omission of items from a code example or from a syntax line.

CHAPTER 1

Introduction to Programing WebLogic Security

The following topics are covered in this section:

Audience for This Guide

This document is intended for the following audiences:

- Application Developers

  Developers who are Java programmers that focus on developing client applications, adding security to Web applications and Enterprise JavaBeans (EJBs). They work with other engineering, Quality Assurance (QA), and database teams to implement security features. Application Developers have in-depth/working knowledge of Java (including J2EE components such as servlets/JSPs and JSEE) and Java security.

  Application developers use the WebLogic security and Java 2 security application programming interfaces (APIs) to secure their applications. Therefore, this document provides instructions for using those APIs for securing Web applications, Java applications, and Enterprise JavaBeans (EJBs).

- Security Developers

  Developers who focus on defining the system architecture and infrastructure for security products that integrate into WebLogic Server and on developing custom security providers for use with WebLogic Server. They work with Application Architects to ensure that the security architecture is implemented according to design and that no security holes are introduced. They also work with Server Administrators to ensure that security is properly configured. Security Developers have a solid understanding of security concepts, including authentication, authorization, auditing (AAA), in-depth knowledge of Java (including Java Management eXtensions (JMX), and working knowledge of WebLogic Server and security provider functionality.

  Security developers use the Security Service Provider Interfaces (SSPIs) to develop custom security providers for use with WebLogic Server, however, this document does not address this task. For information on how to use the SSPIs to develop custom security providers, see *Developing Security Providers for WebLogic Server*.

- Server Administrators

  Administrators who work closely with Application Architects to design a security scheme for the server and the applications running on the server, to identify potential security risks, and to propose configurations that prevent security problems. Related responsibilities may include maintaining critical production systems, configuring and managing security realms, implementing authentication and authorization schemes for server and application resources, upgrading security features, and maintaining security provider databases. Server Administrators have in-depth knowledge of the Java security architecture, including Web application and EJB security, Public Key security, and SSL.

- Application Administrators

  Administrators who work with Server Administrators to implement and maintain security configurations and authentication and authorization schemes, and to set up and maintain access to deployed application resources in defined security realms. Application Administrators have general knowledge of security concepts and the Java Security architecture. They understand Java, XML, deployment descriptors, and can identify security events in server and audit logs.

  While administrators typically use the Administration Console to deploy, configure, and manage applications when they put the applications into production, application developers may also use the Administration Console to test their applications before they are put into production. At a minimum, testing requires that applications be deployed and configured.

This document does not cover some aspects of administration as it relates to security, rather, it references *Managing WebLogic Security*, *Securing WebLogic Resources*, and *Administration Console Online Help* for descriptions of how to use the Administration Console to perform security tasks.

This document does not provide instructions on how to configure WebLogic Security providers and Custom security providers. For information on configuring WebLogic security providers and Custom security providers, see *Managing WebLogic Security*.

Note: This document is not intended for developers who want to write Custom security providers for use with WebLogic Server. It does not describe how to write Custom security providers. For information on developing Custom security providers, see *Developing Security Providers for WebLogic Server*.

What Is Security?

Security refers to techniques for ensuring that data stored in a computer or passed between computers is not compromised. Most security measures involve proof material and data encryption. Proof material is typically a secret word or phrase that gives a user access to a particular application or system. Data encryption is the translation of data into a form that cannot be interpreted without holding or supplying the same secret.

Distributed applications, such as those used for electronic commerce (e-commerce), offer many access points at which malicious people can intercept data, disrupt operations, or generate fraudulent input. As a business becomes more distributed the probability of security breaches increases. Accordingly, as a business distributes its applications, it becomes increasingly important for the distributed computing software upon which such applications are built to provide security.

An application server resides in the sensitive layer between end users and your valuable data and resources. WebLogic Server provides authentication, authorization, and encryption services with which you can guard these resources. These services cannot provide protection, however, from an intruder who gains access by discovering and exploiting a weakness in your deployment environment.

Therefore, whether you deploy WebLogic Server on the Internet or on an intranet, it is a good idea to hire an independent security expert to go over your security plan and procedures, audit your installed systems, and recommend improvements.

Another good strategy is to read as much as possible about security issues and appropriate security measures. The document *Securing a Production Environment* highlights essential security measures for you to consider before you deploy WebLogic Server into a production environment. The document *Securing WebLogic Resources* introduces the various types of WebLogic resources, and provides information that allows you to secure these resources using WebLogic Server. For the latest information about securing Web servers, BEA also recommends reading the Security Improvement Modules, Security Practices, and Technical Implementations information available from the CERT™ Coordination Center operated by Carnegie Mellon University.

BEA suggests that you apply the remedies recommended in our security advisories. In the event of a problem with a BEA product, BEA distributes an advisory and instructions with the appropriate course of action. If you are responsible for security related issues at your site, please register to receive future notifications. BEA has established an e-mail address (`security-report@bea.com`) to which you can send reports of any possible security issues in BEA products. In addition, you are advised to apply every Service Pack as they are released. Service Packs include a roll up of all bug fixes for each version of the product, as well as each of the previously released Service Packs.

Product provided by BEA partners can also help you in your effort to secure the WebLogic Server production environment. For more information, see the BEA Partner's Page.

Types of Security Supported by WebLogic Server

WebLogic Server supports the following security mechanisms:

- "Authentication" on page 1-4
- "Authorization" on page 1-5
- "J2EE Security" on page 1-5

Authentication

Authentication is the mechanism by which callers and service providers prove that they are acting on behalf of specific users or systems. Authentication answers the question, "Who are you?" using credentials. When the proof is bidirectional, it is referred to as mutual authentication.

WebLogic Server supports username and password authentication and certificate authentication. For certificate authentication, WebLogic Server supports both one-way and two-way SSL authentication. Two-way SSL authentication is a form of mutual authentication.

In WebLogic Server, Authentication providers are used to prove the identity of users or system processes. Authentication providers also remember, transport, and make identity information available to various components of a system (via subjects) when needed. You can configure the

Authentication providers using the Web application and EJB deployment descriptor files, or the Administration Console, or a combination of both.

Authorization

Authorization is the process whereby the interactions between users and WebLogic resources are controlled, based on user identity or other information. In other words, authorization answers the question, "What can you access?"

In WebLogic Server, a WebLogic Authorization provider is used to limit the interactions between users and WebLogic resources to ensure integrity, confidentiality, and availability. You can configure the Authorization provider using the Web application and EJB deployment descriptor files, or the Administration Console, or a combination of both.

WebLogic Server also supports the use of programmatic authorization (also referred to in this document as programmatic security) to limit the interactions between users and WebLogic resources.

J2EE Security

For implementation and use of user authentication and authorization, BEA WebLogic Server utilizes the security services of the SDK version 1.4.1 for the Java 2 Platform, Enterprise Edition (J2EE). Like the other J2EE components, the security services are based on standardized, modular components. BEA WebLogic Server implements these Java security service methods according to the standard, and adds extensions that handle many details of application behavior automatically, without requiring additional programming.

Security APIs

This section lists the Security packages and classes that are implemented and supported by WebLogic Server. You use these packages to secure interactions between WebLogic Server and client applications, Enterprise JavaBeans (EJBs), and Web applications.

Note: Several of the WebLogic security packages, classes, and methods are deprecated in this release of WebLogic Server. For more detailed information on deprecated packages and classes, see Appendix A, “Deprecated Security APIs.”

The following topics are covered in this section:

- “JAAS Client Application APIs” on page 1-6
- “SSL Client Application APIs” on page 1-6

- “Other APIs” on page 1-7

JAAS Client Application APIs

You use Java APIs and WebLogic APIs to write client applications that use JAAS authentication.

The following topics are covered in this section:

- “Java JAAS Client Application APIs” on page 1-6
- “WebLogic JAAS Client Application APIs” on page 1-6

Java JAAS Client Application APIs

You use the following Java APIs to write JAAS client applications.

- `javax.naming`
- `javax.security.auth`
- `javax.security.auth.Callback`
- `javax.security.auth.login`
- `javax.security.auth.SPI`

For information on how to use these APIs, see “JAAS Authentication APIs” on page 3-3.

WebLogic JAAS Client Application APIs

You use the following WebLogic APIs to write JAAS client applications.

- `weblogic.security`
- `weblogic.security.auth`
- `weblogic.security.auth.callback`

For information on how to use these APIs, see “JAAS Authentication APIs” on page 3-3.

SSL Client Application APIs

You use Java and WebLogic APIs to write client applications that use SSL authentication.

The following topics are covered in this section:

- “Java SSL Client Application APIs” on page 1-7
- “WebLogic SSL Client Application APIs” on page 1-7

Java SSL Client Application APIs

You use the following Java APIs to write SSL client applications.

- `java.security`
- `java.security.cert`
- `javax.crypto`
- `javax.naming`
- `javax.net`
- `javax.security`
- `javax.servlet`
- `javax.servlet.http`

WebLogic Server also supports the javax.net.SSL API, but BEA recommends that you use the `weblogic.security.SSL` package when you use SSL with WebLogic Server.

For information on how to use these APIs, see "SSL Authentication APIs" on page 4-4.

WebLogic SSL Client Application APIs

You use the following WebLogic APIs to write SSL client applications.

- `weblogic.net.http`
- `weblogic.security.SSL`

For information on how to use these APIs, see "SSL Authentication APIs" on page 4-4.

Other APIs

Additionally, you use the following APIs to develop WebLogic Server applications:

- `weblogic.security.net`

  This API provides interfaces and classes that are used to implement network connection filters. Network connection filters allow or deny connections to WebLogic Server based on attributes such as the IP address, domain, or protocol of the initiator of the network connection. For more information about how to use this API, see "Using Network Connection Filters" on page 6-1.

- `weblogic.security.service`

  This API includes interfaces, classes, and exceptions that support security providers. The WebLogic Security Framework consists of interfaces, classes, and exceptions provided by this API. The interfaces, classes, and exceptions in this API should be used in conjunction with those in the `weblogic.security.spi` package. For more information about how to use this API, see *Developing Security Providers for WebLogic Server*.

- `weblogic.security.services`

  This API provides the server-side authentication class. This class is used to perform a local login to the server. It provides login methods that are used with CallbackHandlers to authenticate the user and return credentials using the default security realm.

- `weblogic.security.spi`

  This package provides the Security Service Provider Interfaces (SSPIs). It provides interfaces, classes, and exceptions that are used for developing custom security providers. In many cases, these interfaces, classes, and exceptions should be used in conjunction with those in the `weblogic.security.service` API. You implement interfaces, classes, and exceptions from this package to create runtime classes for security providers. For more information about how to use the SSPIs, see *Developing Security Providers for WebLogic Server*.

- `weblogic.servlet.security`

  This API provides a server-side API that supports programmatic authentication from within a servlet application. For more about how to use this API, see, "Using the Programmatic Authentication API" on page 2-41.

Administration Console and Security

With regard to security, you can use the Administration Console to define and edit deployment descriptors for Web Applications, EJBs, J2EE Connectors, and Enterprise Applications. This document, *Programming WebLogic Security*, does not describe how to use the Administration Console to configure security. For information on how to use the Administration Console to define and edit deployment descriptors, see *Securing WebLogic Resources* and *Managing WebLogic Security*.

Security Tasks and Code Examples

The security tasks and code examples provided in this document assume that you are using the WebLogic security providers that are included in the WebLogic Server distribution, not custom security providers. The usage of the WebLogic security APIs does not change if you elect to use custom security providers, however, the management procedures of your custom security providers may be different.

Note: This document does not provide comprehensive instructions on how to configure WebLogic Security providers or custom security providers. For information on configuring WebLogic security providers and custom security providers, see *Managing WebLogic Security*.

Securing Web Applications

WebLogic Server supports the J2EE architecture security model for securing Web applications, which includes support for declarative authorization (also referred to in this document as declarative security) and programmatic authorization (also referred to in this document as programmatic security).

This section covers the following topics:

Note: You can use deployment descriptor files and the Administration Console to secure Web applications. This document describes how to use deployment descriptor files. For information on using the Administration Console to secure Web applications, see *Securing WebLogic Resources*.

J2EE Security Model

The document *Designing Enterprise Applications with the J2EE Platform, Second Edition*, published by Sun Microsystems, Inc., states in Section 9.3 Authorization:

> "In the J2EE architecture, a container serves as an authorization boundary between the components it hosts and their callers. The authorization boundary exists inside the container's authentication boundary so that authorization is considered in the context of successful authentication. For inbound calls, the container compares security attributes from the caller's credential with the access control rules for the target component. If the rules are satisfied, the call is allowed. Otherwise, the call is rejected."
>
> "There are two fundamental approaches to defining access control rules: capabilities and permissions. Capabilities focus on what a caller can do. Permissions focus on who can do something. The J2EE application programming model focuses on permissions. In the J2EE architecture, the job of the deployer is to map the permission model of the application to the capabilities of users in the operational environment."

The same document then discusses two ways to control access to application resources using the J2EE architecture, declarative authorization and programmatic authorization.

The document *Designing Enterprise Applications with the J2EE Platform, Second Edition*, published by Sun Microsystems, Inc., in available online at `http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/security/security4.html`.

Declarative Authorization

The document *Designing Enterprise Applications with the J2EE Platform, Second Edition*, published by Sun Microsystems, Inc., states in Section 9.3.1 Authorization:

> "The deployer establishes the container-enforced access control rules associated with a J2EE application. The deployer uses a deployment tool to map an application permission model, which is typically supplied by the application assembler, to policy and mechanisms specific to the operational environment. The application permission model is defined in a deployment descriptor."

WebLogic Server supports the use of deployment descriptors to implement declarative authorization in Web applications.

Note: Declarative authorization is also referred to in this document as declarative security.

The document *Designing Enterprise Applications with the J2EE Platform, Second Edition*, published by Sun Microsystems, Inc., in available online at http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/security/security4.html.

Programmatic Authorization

The document *Designing Enterprise Applications with the J2EE Platform, Second Edition*, published by Sun Microsystems, Inc., states in Section 9.3.2 Programmatic Authorization:

> "A J2EE container makes access control decisions before dispatching method calls to a component. The logic or state of the component doesn't factor in these access decisions. However, a component can use two methods, `EJBContext.isCallerInRole` (for use by enterprise bean code) and `HttpServletRequest.isUserInRole` (for use by Web components), to perform finer-grained access control. A component uses these methods to determine whether a caller has been granted a privilege selected by the component based on the parameters of the call, the internal state of the component, or other factors such as the time of the call."
>
> "The application component provider of a component that calls one of these functions must declare the complete set of distinct roleName values to be used in all calls. These declarations appear in the deployment descriptor as `security-role-ref` elements. Each `security-role-ref` element links a privilege name embedded in the application as a roleName to a security role. Ultimately, the deployer establishes the link between the privilege names embedded in the application and the security roles defined in the deployment descriptor. The link between privilege names and security roles may differ for components in the same application."
>
> "In addition to testing for specific privileges, an application component can compare the identity of its caller, acquired using `EJBContext.getCallerPrincipal` or `HttpServletRequest.getUserPrincipal`, to the distinguished caller identities embedded in the state of the component when it was created. If the identity of the caller is equivalent to a distinguished caller, the component can allow the caller to proceed. If not, the component can prevent the caller from further interaction. The caller principal returned by a container depends on the authentication mechanism used by the caller. Also, containers from different vendors may return different principals for the same user authenticating by the same mechanism. To account for variability in principal forms, an application developer who chooses to apply distinguished caller state in component access decisions should allow multiple distinguished caller identities, representing the same user, to be associated with components. This is recommended especially where application flexibility or portability is a priority."

WebLogic Server supports the use of the `HttpServletRequest.isUserInRole` and `HttpServletRequest.getUserPrincipal` methods and the use of the `security-role-ref` element in deployment descriptors to implement programmatic authorization in Web applications.

Note: Programmatic authorization is also referred to in this document as programmatic security.

The document *Designing Enterprise Applications with the J2EE Platform, Second Edition*, published by Sun Microsystems, Inc., in available online at `http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/security/security4.html`.

Declarative Versus Programmatic Authorization

The document *Designing Enterprise Applications with the J2EE Platform, Second Edition*, published by Sun Microsystems, Inc., states in Section 9.3.3 Declarative Versus Programmatic Authorization:

> "There is a trade-off between the external access control policy configured by the deployer and the internal policy embedded in the application by the component provider. The external policy is more flexible after the application has been written. The internal policy provides more flexible functionality while the application is being written. In addition, the external policy is transparent and completely comprehensible to the deployer, while internal policy is buried in the application and may only be completely understood by the application developer. These trade-offs should be considered in choosing the authorization model for particular components and methods."

The document *Designing Enterprise Applications with the J2EE Platform, Second Edition*, published by Sun Microsystems, Inc., in available online at `http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/security/security4.html`.

Authentication With Web Browsers

Web browsers can connect to WebLogic Server over either a HyperText Transfer Protocol (HTTP) port or an HTTP with SSL (HTTPS) port. The benefits of using an HTTPS port versus an HTTP port is two-fold. With HTTPS connections:

- All communication on the network between the Web browser and the server is encrypted. None of the communication, including the user name and password, flows in clear text.
- As a minimum authentication requirement, the server is required to present a digital certificate to the Web browser client to prove its identity.

If the server is configured for two-way SSL authentication, both the server and client are required to present a digital certificate to each other to prove their identity.

User Name and Password Authentication

WebLogic Server performs user name and password authentication when users use a Web browser to connect to the server via the HTTP port. In this scenario, the browser and an instance of WebLogic Server interact in the following manner to authenticate a user (see Figure 2-1):

1. A user invokes a WebLogic resource in WebLogic Server by entering the URL for that resource in a Web browser. The URL contains HTTP and the HTTP listen port, for example, `http://myserver:7001`.
2. The Web server in WebLogic Server receives the request.

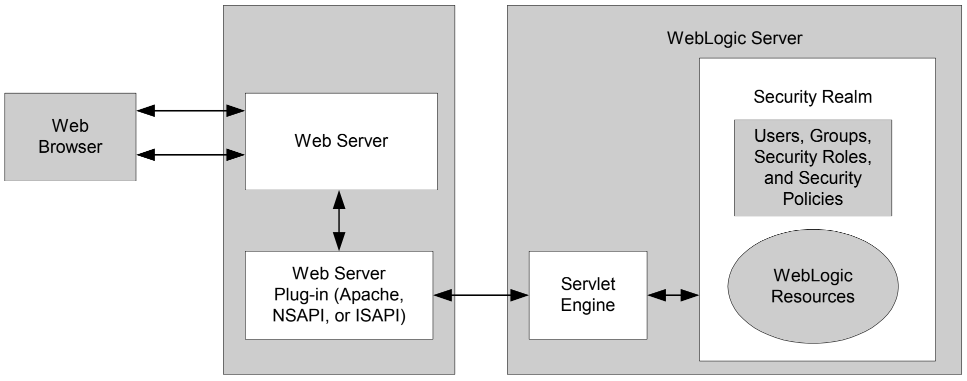
   Note: WebLogic Server provides its own Web server but also supports the use of Apache Server, Microsoft Internet Information Server, and Netscape Enterprise Server as Web servers.
3. The Web server checks whether the WebLogic resource is protected by a security policy. If the WebLogic resource is protected, the Web server uses the established HTTP connection to request a user name and password from the user.
4. When the user's Web browser receives the request from the Web server, it prompts the user for a user name and password.
5. The Web browser sends the request to the Web server again, along with the user name and password.
6. The Web server forwards the request to the Web server plug-in. WebLogic Server provides the following plug-ins for Web servers:
   – Apache-WebLogic Server plug-in
   – Netscape Server Application Programming Interface (NSAPI)
   – Internet Information Server Application Programming Interface (ISAPI)

   The Web server plug-in performs authentication by sending the request, via the HTTP protocol, to WebLogic Server, along with the authentication data (user name and password) received from the user.
7. Upon successful authentication, WebLogic Server proceeds to determine whether the user is authorized to access the WebLogic resource.

8. Before invoking a method on the WebLogic resource, the WebLogic Server instance performs a security authorization check. During this check, the server security extracts the user's credentials from the security context, determines the user's security role, compares the user's security role to the security policy for the requested WebLogic resource, and verifies that the user is authorized to invoke the method on the WebLogic resource.

9. If authorization succeeds, the server fulfills the request.

Figure 2-1 Secure Login for Web Browsers


WebLogic Server
Security Realm
Users, Groups, Security Roles, and Security Policies
WebLogic Resources
Web Browser
Web Server
Web Server Plug-in (Apache, NSAPI, or ISAPI)
Servlet Engine

Note: Username/Password authentication can be required for HTTP and one-way SSL authentication. HTTPS connections can be configured for one-way or two-way SSL authentication.

Digital Certificate Authentication

WebLogic Server uses encryption and digital certificate authentication when Web browser users connect to the server via the HTTPS port. In this scenario, the browser and WebLogic Server instance interact in the following manner to authenticate and authorize a user (see Figure 2-1):

1. A user invokes a WebLogic resource in WebLogic Server by entering the URL for that resource in a Web browser. The URL contains the SSL listen port and the HTTPS schema, for example, `https://myserver:7002`.

2. The Web server in WebLogic Server receives the request.

   Note: WebLogic Server provides its own Web server but also supports the use of Apache Server, Microsoft Internet Information Server, and Netscape Enterprise Server as Web servers.

3. The Web server checks whether the WebLogic resource is protected by a security policy. If the WebLogic resource is protected, the Web server uses the established HTTPS connection to request a user name and password from the user.
4. When the user's Web browser receives the request from WebLogic Server, it prompts the user for a user name and password. (This step is optional.)
5. The Web browser sends the request again, along with the user name and password. (Only supplied if requested by the server.)
6. WebLogic Server presents its digital certificate to the Web browser.
7. The Web browser checks that the server's name used in the URL (for example, `myserver`) matches the name in the digital certificate and that the digital certificate was issued by a trusted third party, that is, a trusted CA
8. If two-way SSL authentication is in force on the server, the server requests a digital certificate from the client.

   Note: Even though WebLogic Server cannot be configured to enforce the full two-way SSL handshake with Web Server proxy plug-ins, proxy plug-ins can be configured to provide the client certificate to the server if it is needed. To do this, configure the proxy plug-in to export the client certificate in the HTTP Header for WebLogic Server. For instructions on how to configure proxy plug-ins to export the client certificate to WebLogic Server, see the configuration information for the specific plug-in in *Using Web Server Plug-Ins With WebLogic Server*.
9. The Web server forwards the request to the Web server plug-in. If secure proxy is set (this is the case if the HTTPS protocol is being used), the Web server plug-in also performs authentication by sending the request, via the HTTPS protocol, to the WebLogic resource in WebLogic Server, along with the authentication data (user name and password) received from the user.

   Note: When using two-way SSL authentication, you can also configure the server to do identity assertion based on the client's certificate, where, instead of supplying a user name and password, the server extracts the user name and password from the client's certificate.
10. Upon successful authentication, WebLogic Server proceeds to determine whether the user is authorized access the WebLogic resource.

11. Before invoking a method on the WebLogic resource, the server performs a security authorization check. During this check, the server extracts the user's credentials from the security context, determines the user's security role, compares the user's security role to the security policy for the requested WebLogic resource, and verifies that the user is authorized to invoke the method on the WebLogic resource.

12. If authorization succeeds, the server fulfills the request.

For more information, see the following documents:

- *Managing WebLogic Security*
- Installing and Configuring the Apache HTTP Server Plug-In
- Installing and Configuring the Microsoft Internet Information Server (IIS) Plug-In
- Installing and Configuring the Netscape Enterprise Server (NES) Plug-In

Multiple Web Applications, Cookies, and Authentication

By default, WebLogic Server assigns the same cookie name (`JSESSIONID`) to all Web applications. When you use any type of authentication, all Web applications that use the same cookie name use a single sign-on for authentication. Once a user is authenticated, that authentication is valid for requests to any Web Application that uses the same cookie name. The user is not prompted again for authentication.

If you want to require separate authentication for a Web application, you can specify a unique cookie name or cookie path for the Web application. Specify the cookie name using the `CookieName` parameter and the cookie path with the `CookiePath` parameter, defined in the WebLogic-specific deployment descriptor `weblogic.xml <session-descriptor>` element. For more information, see session-descriptor in *Assembling and Configuring Web Applications*.

If you want to retain the cookie name and still require independent authentication for each Web application, you can set the cookie path parameter (`CookiePath`) differently for each Web application.

As of Service Pack 1, BEA Systems added a new capability to WebLogic Server that allows a user to securely access HTTPS resources in a session that was initiated using HTTP, without loss of session data. This feature enables Web site designers to prevent session stealing. For more information on this feature, see "Using Secure Cookies to Prevent Session Stealing" on page 2-9.

Using Secure Cookies to Prevent Session Stealing

A common Web security problem is session stealing. This happens when an attacker manages to get a copy of your session cookie, generally while the cookie is being transmitted over the network. This can only happen when the data is being sent in clear-text, that is, it is not encrypted.

As of Service Pack 1, BEA Systems added a new capability to WebLogic Server that allows a user to securely access HTTPS resources in a session that was initiated using HTTP, without loss of session data. To enable this new feature, add `AuthCookieEnabled="true"` to the `WebServer` element in `config.xml`:

```
<WebServer Name="myserver" AuthCookieEnabled="true"/>
```

Setting `AuthCookieEnabled` to `true`, which is the default setting, causes the WebLogic Server instance to send a new secure cookie, `_wl_authcookie_`, to the browser when authenticating via an HTTPS connection. Once the secure cookie is set, the session is allowed to access other security-constrained HTTPS resources only if the cookie is sent from the browser.

Note: Prior to Service Pack 5, this feature requires that a browser uses cookies. If a browser does not support cookies and this feature is enabled, a user will not be able to log in over HTTPS. However, if Service Pack 5 is installed, this feature will work even when cookies are disabled; WebLogic Server will use URL rewriting over secure connections to rewrite secure URLs in order to encode the authCookieID in the URL along with the JSESSIONID.

Thus, WebLogic Server now uses two cookies: the `JSESSIONID` cookie and the `_wl_authcookie_` cookie. By default, the `JSESSIONID` cookie is never secure, but the `_wl_authcookie_` cookie is always secure. A secure cookie is only sent when an encrypted communication channel is in use. Assuming a standard HTTPS login (HTTPS is an encrypted HTTP connection), your browser gets both cookies. For subsequent HTTP access, you are considered authenticated if you have a valid `JSESSIONID` cookie, but for HTTPS access, you must have both cookies to be considered authenticated. If you only have the `JSESSIONID` cookie, you must re-authenticate.

With this feature enabled, once you have logged in over HTTPS, the secure cookie is only sent encrypted over the network and therefore can never be stolen in transit. The `JSESSIONID` cookie is still subject to in-transit hijacking. Therefore, a Web site designer can ensure that session stealing is not a problem by making all sensitive data require HTTPS. While the HTTP session cookie is still vulnerable to being stolen and used, all sensitive operations require the `_wl_authcookie_` cookie, which cannot be stolen, so those operations are protected.

Developing Secure Web Applications

WebLogic Server supports three types of authentication for Web browsers:

- BASIC
- FORM
- CLIENT-CERT

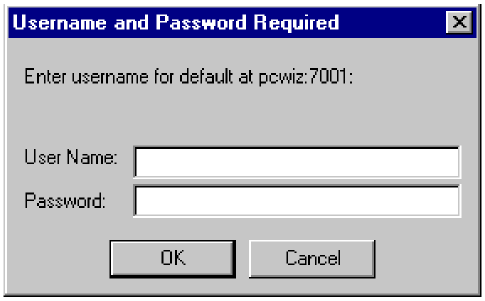
The following sections cover the different ways to use these types of authentication:

- "Developing BASIC Authentication Web Applications" on page 2-10
- "Developing FORM Authentication Web Applications" on page 2-16
- "Using Identity Assertion for Web Application Authentication" on page 2-23
- "Using Two-Way SSL for Web Application Authentication" on page 2-24
- "Developing Swing-Based Authentication Web Applications" on page 2-24
- "Deploying Web Applications" on page 2-25

Developing BASIC Authentication Web Applications

With basic authentication, the Web browser pops up a login screen in response to a WebLogic resource request. The login screen prompts the user for a user name and password. Figure 2-2 shows a typical login screen.

Figure 2-2 Basic Authentication Login Screen


Username and Password Required
Enter username for default at pcwiz:7001:
User Name:
Password:
OK
Cancel

To develop a Web application that provides basic authentication, perform these steps:

1. Create the `web.xml` deployment descriptor. In this file you include the following information (see Listing 2-1):

   a. Define the welcome file. The welcome file name is `welcome.jsp`.

   b. Define a security constraint for each set of Web application resources, that is, URL resources, that you plan to protect. Each set of resources share a common URL. URL resources such as HTML pages, JSPs, and servlets are the most commonly protected, but other types of URL resources are supported. In Listing 2-1, the URL pattern points to the `welcome.jsp` file located in the Web application's top-level directory, the HTTP methods that are allowed to access the URL resource, POST and GET, and the security role name, `webuser`.

   Note: When specifying security role names, observe the following conventions and restrictions:

   - The proper syntax for a security role name is as defined for an `Nmtoken` in the Extensible Markup Language (XML) recommendation available on the Web at: `http://www.w3.org/TR/REC-xml#NT-Nmtoken`.
   - Do not use blank spaces, commas, hyphens, or any characters in this comma-separated list: \t, < >, #, |, &, ~, ?, ( ), { }.
   - Security role names are case sensitive.
   - The BEA suggested convention for security role names is that they be singular.

   c. Use the `<login-config>` to define the type of authentication you want to use and the security realm to which the security constraints will be applied. In Listing 2-1, the BASIC type is specified and the realm is the default realm, which means that the security constraints will apply to the active security realm when the WebLogic Server instance boots.

   Note: In this release of WebLogic Server, the realm name is defined using the `<login-config>` tag and the `<realm-name>` sub-tag is ignored.

   d. Define one or more security roles and map them to your security constraints. In our sample, only one security role, `webuser`, is defined in the security constraint so only one security role name is defined here (see the `<security-role>` tag in Listing 2-1). However, any number of security roles can be defined.

Listing 2-1 Basic Authentication web.xml File

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
    <web-app>
          <welcome-file-list>
               <welcome-file>welcome.jsp</welcome-file>
          </welcome-file-list>
          <security-constraint>
                <web-resource-collection>
                      <web-resource-name>Success</web-resource-name>
                      <url-pattern>/welcome.jsp</url-pattern>
                     <http-method>GET</http-method>
                     <http-method>POST</http-method>
                </web-resource-collection>
                <auth-constraint>
                    <role-name>webuser</role-name>
                </auth-constraint>
          </security-constraint>
          <login-config>
              <auth-method>BASIC</auth-method>
              <realm-name>default</realm-name>
          </login-config>
           <security-role>
               <role-name>webuser</role-name>
           </security-role>
    </web-app>
```

2. Create the `weblogic.xml` deployment descriptor. In this file you map security role names to users and groups. Listing 2-2 shows a sample `weblogic.xml` file that maps the `webuser` security role defined in the `<security-role>` tag in the `web.xml` file to a group named `myGroup`. Note that principals can be users or groups, so the `<principal-tag>` can be used for either.With this configuration, WebLogic Server will only allow users in `myGroup`

to access the protected URL resource—`welcome.jsp`. However, you can use the Administration Console to modify the Web application's security role so that other groups can be allowed to access the protected resource.

Note: Creating the `weblogic.xml` deployment descriptor is optional. If you do not include this file, or include the file but do not include mappings for all security roles, all security roles without mappings will default to any user or group whose name matches the role name. For example, if you name a security role "SampleTester," then any user or group with the name "SampleTester" will be included in that security role.

Listing 2-2 BASIC Authentication weblogic.xml File

```
<!DOCTYPE weblogic-web-app PUBLIC "-//BEA Systems, Inc.//DTD Web
Application 8.1//EN" "http://www.bea.com/servers/wls810/dtd/weblogic
810-web-jar.dtd">

<weblogic-web-app>

     <security-role-assignment>
         <role-name>webuser</role-name>
         <principal-name>myGroup</principal-name>
     </security-role-assignment>

</weblogic-web-app>
```

3. Create a file that produces the Welcome screen that displays when the user enters a user name and password and is granted access. Listing 2-3 shows a sample `welcome.jsp` file. Figure 2-3 shows the Welcome screen.

Listing 2-3 BASIC Authentication welcome.jsp File

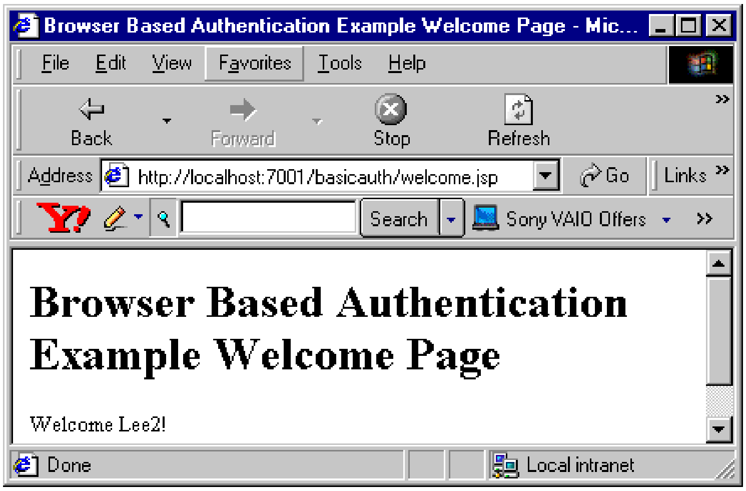
```
<html>
  <head>
    <title>Browser Based Authentication Example Welcome Page</title>
  </head>
  <h1> Browser Based Authentication Example Welcome Page </h1>

  <p> Welcome <%= request.getRemoteUser() %>!
```

```
    </blockquote>
    </body>
</html>
```

Note: In Listing 2-3, notice that the JSP is calling an API (`request.getRemoteUser()`) to get the name of the user that logged in. A different API, `weblogic.security.Security.getCurrentSubject()`, could be used instead. To use this API to get the name of the user, use it with the `SubjectUtils` API as follows:

```
String username = weblogic.security.SubjectUtils.getUsername(
                         weblogic.security.Security.getCurrentSubject());
```

Figure 2-3 Welcome screen


Browser Based Authentication Example Welcome Page - Mic...
File Edit View Favorites Tools Help
Back Forward Stop Refresh
Address http://localhost:7001/basicauth/welcome.jsp Go Links
Search Sony VAIO Offers
Done
Local intranet

Browser Based Authentication Example Welcome Page

Welcome Lee2!

4. Start WebLogic Server and define the users and groups that will have access to the URL resource. In the `weblogic.xml` file (see Listing 2-2), the `<principal-name>` tag defines `myGroup` as the group that has access to the `welcome.jsp`. Therefore, use the Administration Console to define the `myGroup` group, define a user, and add that user to the `myGroup` group. For information on adding users and groups, see Users and Groups in *Securing WebLogic Resources*.

5. Deploy the Web application and use the user defined in the previous step to access the protected URL resource.

   a. For deployment instructions, see "Deploying Web Applications" on page 2-25.

b. Open a Web browser and enter this URL:

```
http://localhost:7001/basicauth/welcome.jsp
```

c. Enter the user name and password. The Welcome screen displays.

Using HttpSessionListener to Account for Browser Caching of Credentials

The browser caches user credentials and frequently resends them to the server automatically. This can give the appearance that WebLogic Server sessions are not being destroyed after logout or timeout. Depending on the browser, the credentials can be cached just for the current browser session, or across browser sessions.

You can validate that a WebLogic Server's session was destroyed by creating a class that implements the `javax.servlet.http.HttpSessionListener` interface. Implementations of this interface are notified of changes to the list of active sessions in a web application. To receive notification events, the implementation class must be configured in the deployment descriptor for the web application in `web.xml`.

To configure a session listener class:

1. Open the `web.xml` deployment descriptor of the Web application for which you are creating a session listener class in a text editor. The `web.xml` file is located in the WEB-INF directory of your Web application.
2. Add an event declaration using the listener element of the web.xml deployment descriptor. The event declaration defines the event listener class that is invoked when the event occurs. For example:

   ```
   <listener>
     <listener-class>myApp.MySessionListener</listener-class>
   </listener>
   ```

   See Configuring an Event Listener Class for additional information and guidelines.

Write and deploy the session listener class. The example shown in Listing 2-4 uses a simple counter to track the session count.

Listing 2-4 Tracking the Session Count

```
package myApp;

import javax.servlet.http.HttpSessionListener;
import javax.servlet.http.HttpSessionEvent;
```

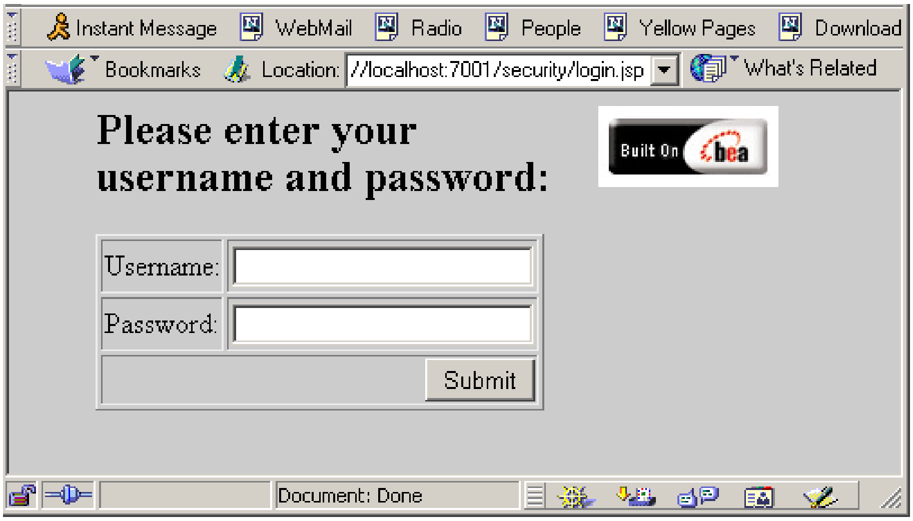
```
public class MySessionListener implements HttpSessionListener {
       private static int sessionCount = 0;

       public void sessionCreated(HttpSessionEvent se) {
               sessionCount++;
                // Write to a log or do some other processing.
       }
       public void sessionDestroyed(HttpSessionEvent se) {
                if(sessionCount > 0)
                         sessionCount--;
                     //Write to a log or do some other processing.
           }
}
```

Developing FORM Authentication Web Applications

When using FORM authentication with Web applications, you provide a custom login screen that the Web browser displays in response to a Web application resource request and an error screen that displays if the login fails. The login screen can be generated using an HTML page, JSP, or servlet. The benefit of form-based login is that you have complete control over these screens so that you can design them to meet the requirements of your application or enterprise policy/guideline.

The login screen prompts the user for a user name and password. Figure 2-4 shows a typical login screen generated using a JSP and Listing 2-5 shows the source code.

Figure 2-4 Form-Based Login Screen (login.jsp)



Please enter your username and password:

Username:

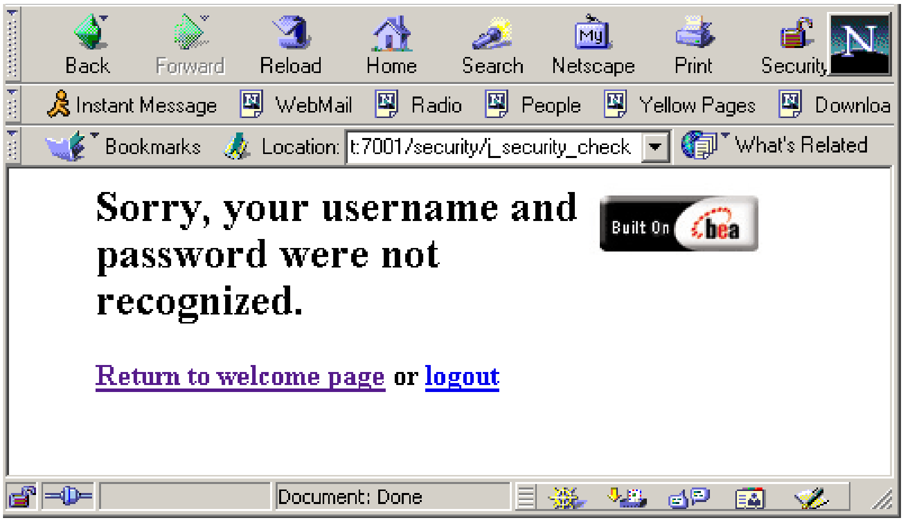
Password:

Listing 2-5 Form-Based Login Screen Source Code (login.jsp)

```
<html>
  <head>)
    <title>Security WebApp login page</title>
  </head>
  <body bgcolor="#cccccc">
  <blockquote>
  <img src=BEA_Button_Final_web.gif align=right>
  <h2>Please enter your user name and password:</h2>
  <p>
  <form method="POST" action="j_security_check">
  <table border=1>
    <tr>
      <td>Username:</td>
      <td><input type="text" name="j_username"></td>
    </tr>
    <tr>
      <td>Password:</td>
      <td><input type="password" name="j_password"></td>
    </tr>
    <tr>
      <td colspan=2 align=right><input type=submit
                                                value="Submit"></td>
    </tr>
  </table>
  </form>
  </blockquote>
  </body>
</html>
```

Figure 2-5 shows a typical login error screen generated using HTML and Listing 2-6 shows the source code.

Figure 2-5 Login Error Screen



Sorry, your username and password were not recognized.

Return to welcome page or logout

Listing 2-6 Login Error Screen Source Code

```
<html>
  <head>
    <title>Login failed</title>
  </head>
  <body bgcolor=#ffffff>
  <blockquote>
  <img src=/security/BEA_Button_Final_web.gif align=right>
  <h2>Sorry, your user name and password were not recognized.</h2>
  <p><b>
  <a href="/security/welcome.jsp">Return to welcome page</a> or
           <a href="/security/logout.jsp">logout</a>
  </b>
  </blockquote>
  </body>
</html>
```

To develop a Web application that provides FORM authentication, perform these steps:

1. Create the `web.xml` deployment descriptor. In this file you include the following information (see Listing 2-7):

   a. Define the welcome file. The welcome file name is `welcome.jsp`.

   b. Define a security constraint for each set of URL resources that you plan to protect. Each set of URL resources share a common URL. URL resources such as HTML pages, JSPs, and servlets are the most commonly protected, but other types of URL resources are supported. In Listing 2-7, the URL pattern points to `/admin/edit.jsp`, thus protecting the `edit.jsp` file located in the Web application's `admin` sub-directory, defines the HTTP method that is allowed to access the URL resource, `GET`, and defines the security role name, `admin`.

   Note: Do not use hyphens in security role names. Security role names with hyphens cannot be modified in the Administration Console. Also, the BEA suggested convention for security role names is that they be singular.

   c. Define the type of authentication you want to use and the security realm to which the security constraints will be applied. In this case, the `FORM` type is specified and no realm is specified, so the realm is the default realm, which means that the security constraints will apply to the security realm that is activated when a WebLogic Server instance boots.

   d. Define one or more security roles and map them to your security constraints. In our sample, only one security role, `admin`, is defined in the security constraint so only one security role name is defined here. However, any number of security roles can be defined.

Listing 2-7 FORM Authentication web.xml File

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

    <welcome-file-list>
        <welcome-file>welcome.jsp</welcome-file>
    </welcome-file-list>

    <security-constraint>
        <web-resource-collection>
            <web-resource-name>AdminPages</web-resource-name>
            <description>
                These pages are only accessible by authorized
                 administrators.
            </description>
            <url-pattern>/admin/edit.jsp</url-pattern>
            <http-method>GET</http-method>
        </web-resource-collection>
        <auth-constraint>
            <description>
                These are the roles who have access.
            </description>
            <role-name>
                admin
            </role-name>
        </auth-constraint>
        <user-data-constraint>
            <description>
                This is how the user data must be transmitted.
            </description>
            <transport-guarantee>NONE</transport-guarantee>
        </user-data-constraint>
    </security-constraint>

    <login-config>
        <auth-method>FORM</auth-method>
        <form-login-config>
```

```
                <form-login-page>/login.jsp</form-login-page>
                <form-error-page>/fail_login.html</form-error-page>
            </form-login-config>
        </login-config>

        <security-role>
            <description>
                An administrator
            </description>
            <role-name>
                admin
            </role-name>
        </security-role>
</web-app>
```

2. Create the `weblogic.xml` deployment descriptor. In this file you map security role names to users and groups. Listing 2-8 shows a sample `weblogic.xml` file that maps the `admin` security role defined in the `<security-role>` tag in the `web.xml` file to the group `supportGroup`. With this configuration, WebLogic Server will only allow users in the `supportGroup` group to access the protected WebLogic resource. However, you can use the Administration Console to modify the Web application's security role so that other groups can be allowed to access the protected WebLogic resource.

Listing 2-8 FORM Authentication weblogic.xml File

```
<!DOCTYPE weblogic-web-app PUBLIC "-//BEA Systems, Inc.//DTD Web
Application 7.0//EN"
"http://www.bea.com/servers/wls700/dtd/weblogic700-web-jar.dtd">

<weblogic-web-app>

     <security-role-assignment>
         <role-name>admin</role-name>
         <principal-name>supportGroup</principal-name>
     </security-role-assignment>

</weblogic-web-app>
```

3. Create a Web application file that produces the welcome screen when the user requests the protected Web application resource by entering the URL. Listing 2-9 shows a sample `welcome.jsp` file. Figure 2-3 shows the Welcome screen.

Listing 2-9 Form Authentication welcome.jsp File

```
<html>
  <head>
    <title>Security login example</title>
  </head>

  <%
    String bgcolor;
    if ((bgcolor=(String)application.getAttribute("Background")) ==
        null)
    {
        bgcolor="#cccccc";
    }
  %>

  <body bgcolor=<%="\""+bgcolor+"\""%>>

  <blockquote>
  <img src=BEA_Button_Final_web.gif align=right>
  <h1> Security Login Example </h1>

  <p> Welcome <%= request.getRemoteUser() %>!

  <p> If you are an administrator, you can configure the background
  color of the Web Application.
  <br> <b><a href="admin/edit.jsp">Configure background</a></b>.

  <% if (request.getRemoteUser() != null) { %>
    <p> Click here to <a href="logout.jsp">logout</a>.
  <% } %>

  </blockquote>
  </body>
</html>
```

Note: In Listing 2-3, notice that the JSP is calling an API (`request.getRemoteUser()`) to get the name of the user that logged in. A different API, `weblogic.security.Security.getCurrentSubject()`, could be used instead. To use this API to get the name of the user, use it with the `SubjectUtils` API as follows:

```
String username = weblogic.security.SubjectUtils.getUsername(
                              weblogic.security.Security.getCurrentSubject());
```

4. Start WebLogic Server and define the users and groups that will have access to the URL resource. In the `weblogic.xml` file (see Listing 2-8), the `<role-name>` tag defines `admin` as the group that has access to the `edit.jsp`, file and defines the user `joe` as a member of that group. Therefore, use the Administration Console to define the `admin` group, and define user `joe` and add `joe` to the `admin` group. You can also define other users and add them to the group and they will also have access to the protected WebLogic resource. For information on adding users and groups, see Users and Groups in *Securing WebLogic Resources*.

5. Deploy the Web application and use the user(s) defined in the previous step to access the protected Web application resource.

   a. For deployment instructions, see "Deploying Web Applications" on page 2-25.

   b. Open a Web browser and enter this URL:

      `http://`*`hostname`*`:7001/security/welcome.jsp`

   c. Enter the user name and password. The Welcome screen displays.

Using Identity Assertion for Web Application Authentication

You use identity assertion in Web applications to verify client identities for authentication purposes. When using identity assertion, the following requirements must be met:

1. The authentication type must be set to CLIENT-CERT.

2. An Identity Assertion provider must be configured in the server. If the Web browser or Java client requests a WebLogic Server resource protected by a security policy, WebLogic Server requires that the Web browser or Java client have an identity. The WebLogic Identity Assertion provider maps the token from a Web browser or Java client to a user in a WebLogic Server security realm. For information on how to configure an Identity Assertion provider, see Configuring a WebLogic Identity Assertion Provider.

3. The user corresponding to the token's value must be defined in the server's security realm; otherwise the client will not be allowed to access a protected WebLogic resource. For information on configuring users on the server, see Creating Users in *Managing WebLogic Security.*

Using Two-Way SSL for Web Application Authentication

You use two-way SSL in Web applications to verify that clients are whom they claim to be. When using two-way SSL, the following requirements must be met:

1. The authentication type must be set to CLIENT-CERT.
2. The server must be configured for two-way SSL. For information on using SSL and digital certificates, see "Using SSL Authentication in Java Clients" on page 4-1. For information on configuring SSL on the server, see Configuring SSL in *Managing WebLogic Security*.
3. The client must use HTTPS to access the Web application on the server.
4. An Identity Assertion provider must be configured in the server. If the Web browser or Java client requests a WebLogic Server resource protected by a security policy, WebLogic Server requires that the Web browser or Java client have an identity. The WebLogic Identity Assertion provider allows you to enable a user name mapper in the server that maps the digital certificate of a Web browser or Java client to a user in a WebLogic Server security realm. For information on how to configure an Identity Assertion provider and a user name mapper, see Configuring a WebLogic Identity Assertion Provider and Configuring a User Name Mapper in *Managing WebLogic Security.*
5. The user corresponding to the Subject's Distinguished Name (SubjectDN) attribute in the client's digital certificate must be defined in the server's security realm; otherwise the client will not be allowed to access a protected WebLogic resource. For information on configuring users on the server, see Creating Users in *Managing WebLogic Security.*

Note: When you use SSL authentication, it is not necessary to use `web.xml` and `weblogic.xml` files to specify server configuration because you use the Administration Console to specify the server's SSL configuration. For information on configuring SSL on the server, see Configuring SSL in *Managing WebLogic Security*.

Developing Swing-Based Authentication Web Applications

Web browsers can also be used to run graphical user interfaces (GUIs) that were developed using Swing components. The Swing components, which are part of the Java Foundation Classes (JFC), can be used with either JDK 1.1 or the Java 2 platform.

For information on how to create a graphical user interface (GUI) for applications and applets using the Swing components, see the *Creating a GUI with JFC/Swing* tutorial (also known as The Swing Tutorial) produced by Sun Microsystems, Inc. You can access this tutorial on the Web at `http://java.sun.com/docs/books/tutorial/uiswing/`.

After you have developed your Swing-based GUI, refer to "Developing FORM Authentication Web Applications" on page 2-16 and use the Swing-based screens to perform the steps required to develop a Web application that provides FORM authentication.

Note: When developing a Swing-based GUI, do not rely on the Java Virtual Machine-wide user for child threads of the swing event thread. This is not J2EE compliant and does not work in thin clients, or in IIOP in general. Instead, take either of the following approaches:

- Make sure an `InitialContext` is created before any Swing artefacts.
- Or, use the Java Authentication and Authorization Service (JAAS) to login and then use the `Security.runAs()` method inside the Swing event thread and its children.

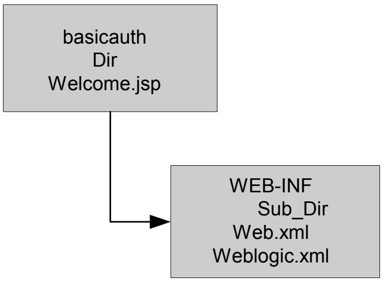
Deploying Web Applications

To deploy a Web application on a server running in *development* mode, perform the following steps:

Note: For more information about deploying Web applications in either development of production mode, see Deploying Web Applications in *Developing Web Applications for WebLogic Server*.

1. Set up a directory structure for the Web application's files. Figure 2-6 shows the directory structure for the Web application named `basicauth`. The top-level directory must be assigned the name of the Web application and the sub-directory must be named `WEB-INF`.

Figure 2-6 Basicauth Web Application Directory Structure



2. To deploy the Web application in exploded directory format, that is, not in the Java archive (jar) format, simply move your directory to the `applications` directory on your server. For example, you would deploy the `basicauth` Web application in the following location:

   ```
   WL_HOME\user_projects\domains\mydomain\applications\basicauth
   ```

   If the WebLogic Server instance is running, the application should auto-deploy. Use the Administration Console to verify that the application deployed.

   If the WebLogic Server instance is not running, the Web application should auto-deploy when you start the server.

3. If you have not done so already, use the Administration Console to configure the users and groups that will have access to the Web application. To determine the users and groups that are allowed access to the protected WebLogic resource, examine the `weblogic.xml` file. For example, the `weblogic.xml` file for the `basicauth` sample (see Listing 2-2) defines `myGroup` as the only group to have access to the `welcome.jsp` file.

For more information on deploying secure Web applications, see Deploying Web Applications in *Developing Web Applications for WebLogic Server*.

Using Declarative Security With Web Applications

To implement declarative security in Web applications, you use deployment descriptors (`web.xml` and `weblogic.xml`) to define security requirements. The deployment descriptors map the application's logical security requirements to its runtime definitions. And at runtime, the servlet container uses the security definitions to enforce the requirements. For a discussion of using deployment descriptors, see "Developing Secure Web Applications" on page 2-10.

For information about how to use deployment descriptors and the `externally-defined` element to configure security in Web applications declaratively, see "externally-defined" on page 2-33.

For information about how to use the Administration Console to configure security in Web applications, see *Securing WebLogic Resources*.

Web Application Security-Related Deployment Descriptors

The following topics describe the deployment descriptor elements that are used in the `web.xml` and `weblogic.xml` files to define security requirements in Web applications:

- "Web.xml Deployment Descriptors" on page 2-27
- "Weblogic.xml Deployment Descriptors" on page 2-32

Web.xml Deployment Descriptors

The following `web.xml` security-related deployment descriptor elements are supported by WebLogic Server:

- "auth-constraint" on page 2-27
- "security-constraint" on page 2-28
- "security-role" on page 2-29
- "security-role-ref" on page 2-30
- "user-data-constraint" on page 2-31
- "web-resource-collection" on page 2-31

The information in this section is based on the Document Type Descriptor (DTD) for `web.xml` provided by Sun Microsystems, Inc. The DTD for `web.xml` is available on the Web at `http://java.sun.com/dtd/web-app_2_3.dtd.`

auth-constraint

The optional `auth-constraint` element defines which groups or principals have access to the collection of Web resources defined in this security constraint.

The following table describes the elements you can define within an `auth-constraint` element.

Element	Required/ Optional	Description
<description>	Optional	A text description of this security constraint.
<role-name>	Optional	Defines which security roles can access resources defined in this security-constraint. Security role names are mapped to principals using the security-role-ref element. See "security-role-ref" on page 2-30.

Used Within

The `auth-constraint` element is used within the `security-constraint` element.

Example

See Listing 2-10 for an example of how to use the `auth-constraint` element in a `web.xml` file.

security-constraint

The `security-constraint` element is use in the web.xml file to define the access privileges to a collection of resources defined by the `web-resource-collection` element.

The following table describes the elements you can define within a `security-constraint` element.

Element	Required/ Optional	Description
<web-resource-collection>	Required	Defines the components of the Web Application to which this security constraint is applied. For more information, see "web-resource-collection" on page 2-31.
<auth-constraint>	Optional	Defines which groups or principals have access to the collection of web resources defined in this security constraint.For more information, see "auth-constraint" on page 2-27.
<user-data-constraint>	Optional	Defines defines how data communicated between the client and the server should be protected. For more information, see "user-data-constraint" on page 2-31.

Example

Listing 2-10 shows how to use the `security-constraint` element to defined security for the `SecureOrdersEast` in a `web.xml` file.

Listing 2-10 Security Constraint Example

```
web.xml entries:
<security-constraint>
     <web-resource-collection>
          <web-resource-name>SecureOrdersEast</web-resource-name>
          <description>
             Security constraint for
             resources in the orders/east directory
          </description>
          <url-pattern>/orders/east/*</url-pattern>
          <http-method>POST</http-method>
          <http-method>GET</http-method>
     </web-resource-collection>
     <auth-constraint>
          <description>
           constraint for east coast sales
          </description>
          <role-name>east</role-name>
          <role-name>manager</role-name>
     </auth-constraint>
 <user-data-constraint>
          <description>SSL not required</description>
          <transport-guarantee>NONE</transport-guarantee>
     </user-data-constraint>
</security-constraint>
...
```

security-role

The `security-role` element contains the definition of a security role. The definition consists of an optional description of the security role, and the security role name.

The following table describes the elements you can define within a `security-role` element.

Element	Required/ Optional	Description
<description>	Optional	A text description of this security role.
<role-name>	Required	The role name. The name you use here must have a corresponding entry in the WebLogic-specific deployment descriptor, `weblogic.xml`, which maps roles to principals in the security realm. For more information, see "security-role-assignment" on page 2-38.

Example

See Listing 2-13 for an example of how to use the `security-role` element in a `web.xml` file.

security-role-ref

The `security-role-ref` element links a security role name defined by `<security-role>` to an alternative role name that is hard-coded in the servlet logic. This extra layer of abstraction allows the servlet to be configured at deployment without changing servlet code.

The following table describes the elements you can define within a `security-role-ref` element.

Element	Required/ Optional	Description
<description>	Optional	Text description of the role.
<role-name>	Required	Defines the name of the security role or principal that is used in the servlet code.
<role-link>	Required	Defines the name of the security role that is defined in a `<security-role>` element later in the deployment descriptor.

Example

See Listing 2-16 for an example of how to use the `security-role-ref` element in a `web.xml` file.

user-data-constraint

The `user-data-constraint` element defines how data communicated between the client and the server should be protected.

The following table describes the elements you may define within a `user-data-constraint` element.

Element	Required/ Optional	Description
<description>	Optional	A text description.
<transport-guarantee>	Required	Specifies data security requirements for communications between the client and the server. Range of values: `NONE`—The application does not require any transport guarantees. `INTEGRAL`—The application requires that the data be sent between the client and server in such a way that it cannot be changed in transit. `CONFIDENTIAL`—The application requires that data be transmitted so as to prevent other entities from observing the contents of the transmission. WebLogic Server establishes a Secure Sockets Layer (SSL) connection when the user is authenticated using the `INTEGRAL` or `CONFIDENTIAL` transport guarantee.

Used Within

The `user-data-constraint` element is used within the `security-constraint` element.

Example

See Listing 2-10 for an example of how to use the `user-data-constraint` element in a `web.xml` file.

web-resource-collection

The `web-resource-collection` element is used to identify a subset of the resources and `HTTP` methods on those resources within a Web application to which a security constraint applies. If no `HTTP` methods are specified, then the security constraint applies to all `HTTP` methods.

The following table describes the elements you can define within a `web-resource-collection` element.

Element	Required/ Optional	Description
<web-resource-name>	Required	The name of this web resource collection.
<description>	Optional	Text description of the Web resource.
<url-pattern>	Required	The mapping, or location, of the Web resource collection.
<http-method>	Optional	The `HTTP` methods to which the security constraint applies when clients attempt to access the Web resource collection. If no `HTTP` methods are specified, then the security constraint applies to all `HTTP` methods.

Used Within

The `web-resource-collection` element is used within the `security-constraint` element.

Example

See Listing 2-10 for an example of how to use the `web-resource-collection` element in a `web.xml` file.

Weblogic.xml Deployment Descriptors

The following `weblogic.xml` security-related deployment descriptor elements are supported by WebLogic Server:

- "externally-defined" on page 2-33
- "run-as-principal-name" on page 2-35
- "run-as-role-assignment" on page 2-36
- "security-permission" on page 2-37
- "security-permission-spec" on page 2-37
- "security-role-assignment" on page 2-38

For additional information on `weblogic.xml` deployment descriptions, see the Document Type Descriptor (DTD) for `weblogic.xml` at http://www.bea.com/servers/wls810/dtd/weblogic810-web-jar.dtd.

externally-defined

In WebLogic Server 8.1 and later, the `externally-defined` element is supported for use in the `weblogic.xml` deployment descriptors. You use this element, instead of the `<principal-name>` tag, to explicitly indicate that you want the security roles defined by the `role-name` element in the `web.xml` deployment descriptors to use the mappings that you specify in the Administration Console.

Note: The `externally-defined` element replaces the `global-role` element that was used in WebLogic Server 7.0 SP1. The `externally-defined` element has the same functionality as the `global-role` element. The `global-role` element was deprecated in WebLogic Server 8.1.

The `externally-defined` element gives you the flexibility of not having to specify a specific security role mapping for each security role defined in the deployment descriptors for a particular Web application. Rather, you can use the Administration Console to specify and modify a specific role mapping for each defined role at anytime. Additionally, because you may elect to use this element on some applications and not others, it is not necessary to select the **ignore roles and polices from DD** option for the security realm. You select this option in the **On Future Redeploys:** field on the **General** tab of the **Security->Realms->myrealm** control panel on the Administration Console. Therefore, within the same security realm, deployment descriptors can be used to specify and modify security for some applications while the Administration Console can be used to specify and modify security for others.

Note: When specifying security role names, observe the following conventions and restrictions:

- The proper syntax for a security role name is as defined for an `Nmtoken` in the Extensible Markup Language (XML) recommendation available on the Web at: `http://www.w3.org/TR/REC-xml#NT-Nmtoken`.
- Do not use blank spaces, commas, hyphens, or any characters in this comma-separated list: \t, < >, #, |, &, ~, ?, ( ), { }.
- Security role names are case sensitive.
- The BEA suggested convention for security role names is that they be singular.

Used Within

The `externally-defined` element is used within the `security-role-assignment` element.

Example

Listing 2-11 and Listing 2-12 show by comparison how to use the `externally-defined` `element` in the `weblogic.xml` file. In Listing 2-12, the specification of the "webuser" `externally-defined` element in the `weblogic.xml` means that for security to be correctly configured on the `getReceipts` method, the principals for `webuser` will have to be created in the Administration Console.

Listing 2-11 Using the web.xml and weblogic.xml Files to Map Security Roles and Principals to a Security Realm

```
web.xml entries:

<web-app>
           ...
           <security-role>
               <role-name>webuser</role-name>
           </security-role>
           ...
</web-app>

<weblogic.xml entries:

<weblogic-web-app>

     <security-role-assignment>
         <role-name>webuser</role-name>
         <principal-name>myGroup</principal-name>
         <principal-name>Bill</principal-name>
         <principal-name>Mary</principal-name>
     </security-role-assignment>

</weblogic-web-app>
```

Listing 2-12 Using the externally-defined tag in Web Application Deployment Descriptors

```
web.xml entries:

<web-app>
           ...
           <security-role>
               <role-name>webuser</role-name>
           </security-role>
           ...
</web-app>

<weblogic.xml entries:

<weblogic-web-app>

     <security-role-assignment>
         <role-name>webuser</role-name>
         <externally-defined/>
     </security-role-assignment>
```

For information about how to use the Administration Console to configure security for Web applications, see *Securing WebLogic Resources*.

run-as-principal-name

The `run-as-principal-name` element specifies the name of a principal to used for a security role defined by a `run-as` element in the companion `web.xml` file.

Used Within

The `run-as-principal-name` element is used within a `run-as-role-assignment` element.

Example

For an example of how to use the `run-as-principal-name` element, see Listing 2-13.

run-as-role-assignment

The `run-as-role-assignment` element maps a given role name, defined by a `role-name` element in the companion `web.xml` file, to a valid user name in the system. The value can be overridden for a given servlet by the `run-as-principal-name` element in the servlet-descriptor. If the `run-as-role-assignment` element is absent for a given role name, the Web application container chooses the first principal-name defined in the `security-role-assignment` element.

The following table describes the elements you can define within a `run-as-role-assignment` element.

Element	Required Optional	Description
<role-name>	Required	Specifies the name of a security role name specified in a `run-as` element in the companion `web.xml` file.
<run-as-principal-name>	Required	Specifies a principal for the security role name defined in a `run-as` element in the companion `web.xml` file.

Example:

Listing 2-13 shows how to use the `run-as-role-assignment` element to have the `SnoopServlet` always execute as a user `joe`.

Listing 2-13 run-as-role-assignment Element Example

```
web.xml:

  <servlet>
    <servlet-name>SnoopServlet</servlet-name>
    <servlet-class>extra.SnoopServlet</servlet-class>
    <run-as>
      <role-name>runasrole</role-name>
    </run-as>
  </servlet>
  <security-role>
    <role-name>runasrole</role-name>
  </security-role>

weblogic.xml:
```

```
<weblogic-web-app>
   <run-as-role-assignment>
     <role-name>runasrole</role-name>
     <run-as-principal-name>joe</run-as-principal-name>
   </run-as-role-assignment>
</weblogic-web-app>
```

security-permission

The `security-permission` element specifies a security permission that is associated with a J2EE Sandbox.

Example

For an example of how to used the `security-permission` element, see Listing 2-14.

security-permission-spec

The `security-permission-spec` element specifies a single security permission based on the Security policy file syntax. Refer to the following URL for Sun's implementation of the security permission specification:

http://java.sun.com/j2se/1.3/docs/guide/security/PolicyFiles.html#FileSyntax

Note: Disregard the optional `codebase` and `signedBy` clauses.

Used Within

The security-permission-`spec` element is used within the `security-permission` element.

Example

Listing 2-14 shows how to use the security-permission-spec element to grant permission to the `java.net.SocketPermission` class.

Listing 2-14 security-permission-spec Element Example

```
<weblogic-web-app>
   <security-permission>
```

```
     <description>Optional explanation goes here</description>
     <security-permission-spec>
<!—
A single grant statement following the syntax of
http://java.sun.com/j2se/1.3/docs/guide/security/PolicyFiles.html#FileSynt
ax, without the “codebase” and “signedBy” clauses, goes here. For example:
-->
      grant {
      permission java.net.SocketPermission “*”, “resolve”;
      };
     </security-permission-spec>
   </security-permission>
</weblogic-web-app>
```

In Listing 2-14, `permission java.net.SocketPermission` is the permission class name, `"*"` represents the target name, and `resolve` (resolve host/ip name service lookups) indicates the action.

security-role-assignment

The `security-role-assignment` element declares a mapping between a security role and one or more principals in the WebLogic Server security realm,

Example

Listing 2-15 shows how to use the `security-role-assignment` element to assign principals to the `PayrollAdmin` role.

Listing 2-15 security-role-assignment Element Example

```
<weblogic-web-app>
  <security-role-assignment>
    <role-name>PayrollAdmin</role-name>
    <principal-name>Tanya</principal-name>
    <principal-name>Fred</principal-name>
    <principal-name>system</principal-name>
  </security-role-assignment>
</weblogic-web-app>
```

Using Programmatic Security With Web Applications

You can write your servlets to access users and security roles programmatically in your servlet code. To do this, use the following methods in your servlet code: `javax.servlet.http.HttpServletRequest.getUserPrincipal` and `javax.servlet.http.HttpServletRequest.isUserInRole(String role)` methods.

getUserPrincipal

You use the `getUserPrincipal()` method to determine the current user of the Web application. This method returns a `WLSUser Principal` if one exists in the current user. In the case of multiple `WLSUser Principals`, the method returns the first in the ordering defined by the `Subject.getPrincipals().iterator()` method. If there are no `WLSUser Principals`, then the `getUserPrincipal()` method returns the first non-`WLSGroup Principal`. If there are no `Principals` or all `Principals` are of type `WLSGroup`, this method returns `null`. This behavior is identical to the semantics of the `weblogic.security.SubjectUtils.getUserPrincipal()` method.

For more information about how to use the `getUserPrincipal()` method, see `http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Security5.html#80556`.

isUserInRole

The `javax.servlet.http.HttpServletRequest.isUserInRole(String role)` method returns a boolean indicating whether the authenticated user is granted the specified logical security "role." If the user has not been authenticated, this method returns false.

The `isUserInRole()` method maps security roles to the group names in the security realm. Listing 2-16 shows the elements that are used with the `<servlet>` element to define the security role in the `web.xml` file.

Listing 2-16 IsUserInRole web.xml and weblogic.xml Elements

```
Begin web.xml entries:

...
<servlet>
        <security-role-ref>
             <role-name>user-rolename</role-name>
             <role-link>rolename-link</role-link>
        </security-role-ref>
</servlet>

<security-role>
        <role-name>rolename-link</role-name>
</security-role>
...
```

Begin weblogic.xml entries:

```
...
<security-role-assignment>
        <role-name>rolename-link</role-name>
        <principal-name>groupname</principal>
        <principal-name>username</principal>
</security-role-assignment>
...
```

The string `role` is mapped to the name supplied in the `<role-name>` element, which is nested inside the `<security-role-ref>` element of a `<servlet>` declaration in the `web.xml` deployment descriptor. The `<role-name>` element defines the name of the security role or `principal` (the user or group) that is used in the servlet code. The `<role-link>` element maps to a `<role-name>` defined in the `<security-role-assignment>` element in the `weblogic.xml` deployment descriptor.

Note: When specifying security role names, observe the following conventions and restrictions:

- The proper syntax for a security role name is as defined for an `Nmtoken` in the Extensible Markup Language (XML) recommendation available on the Web at: `http://www.w3.org/TR/REC-xml#NT-Nmtoken`.
- Do not use blank spaces, commas, hyphens, or any characters in this comma-separated list: \t, < >, #, |, &, ~, ?, ( ), { }.
- Security role names are case sensitive.
- The BEA suggested convention for security role names is that they be singular.

For example, if the client has successfully logged in as user `Bill` with the security role of `manager`, the following method would return true:

```
request.isUserInRole("manager")
```

Listing 2-17 provides an example.

Listing 2-17 Example of Security Role Mapping

```
Servlet code:
out.println("Is the user a Manager? " +
                         request.isUserInRole("manager"));

web.xml entries:

<servlet>
. . .
   <role-name>manager</role-name>
   <role-link>mgr</role-link>
. . .
</servlet>

<security-role>
   <role-name>mgr</role-name>
</security-role>

weblogic.xml entries:

<security-role-assignment>
   <role-name>mgr</role-name>
   <principal-name>bostonManagers</principal-name>
   <principal-name>Bill</principal-name>
   <principal-name>Ralph</principal-name>
</security-role-ref>
```

Using the Programmatic Authentication API

There are some applications where programmatic authentication is appropriate.

WebLogic Server provides a server-side API that supports programmatic authentication from within a servlet application:

```
weblogic.servlet.security.ServletAuthentication
```

Using this API, you can write servlet code that authenticates the user, logs in the user, and associates the user with the current session so that the user is registered in the default (active) security realm. Once the login is completed, it appears as if the user logged in using the standard mechanism.

You have the option of using either of two WebLogic-supplied classes with the `ServletAuthentication` API, the `weblogic.security.SimpleCallbackHandler` class or the `weblogic.security.URLCallbackHandler` class. For more information on these classes, see Javadocs for WebLogic Classes.

Listing 2-18 shows an example that uses `SimpleCallbackHandler`. Listing 2-19 shows an example that uses `URLCallbackHandler`.

Listing 2-18 Programmatic Authentication Code Fragment Using the SimpleCallbackHandler Class

```
CallbackHandler handler = new SimpleCallbackHandler(username, password);
Subject mySubject = weblogic.security.services.Authentication.login(handler);
weblogic.servlet.security.ServletAuthentication.runAs(mySubject, request);
```

Where `request` is the `httpservletrequest` object.

Listing 2-19 Programmatic Authentication Code Fragment Using the URLCallbackHandler Class

```
CallbackHandler handler = new URLCallbackHandler(username, password);
Subject mySubject = weblogic.security.services.Authentication.login(handler);
weblogic.servlet.security.ServletAuthentication.runAs(mySubject, request);
```

Where `request` is the `httpservletrequest` object.

Using JAAS Authentication in Java Clients

The following topics are covered in this section:

JAAS and WebLogic Server

The Java Authentication and Authorization Service (JAAS) is a standard extension to the security in the Java Software Development Kit version 1.4.1. JAAS provides the ability to enforce access controls based on user identity. JAAS is provided in WebLogic Server as an alternative to the JNDI authentication mechanism.

WebLogic Server clients use the authentication portion of the standard JAAS only. The JAAS LoginContext provides support for the ordered execution of all configured authentication provider LoginModule instances and is responsible for the management of the completion status of each configured provider.

Note the following considerations when using JAAS authentication for Java clients:

- WebLogic Server clients can either use the JNDI login or JAAS login for authentication, however JAAS login is the preferred method.

- While JAAS is the preferred method of authentication, the WebLogic-supplied LoginModule (`weblogic.security.auth.login.UsernamePasswordLoginModule`) only supports username and password authentication. Thus, for client certificate authentication (also referred to as two-way SSL authentication), you should use JNDI. To use JAAS for client certificate authentication, you must write a custom LoginModule that does certificate authentication.

  Note: If you are going to write your own LoginModule for use with WebLogic Server clients, you must have it call `weblogic.security.auth.Authenticate.authenticate()` to perform the login.

- To perform a JAAS login from a remote Java client (that is, the Java client is not a WebLogic Server client), you may use the WebLogic-supplied LoginModule to perform the login. However, if you elect not to use the WebLogic-supplied LoginModule but decide to write your own instead, you must have it call the `weblogic.security.auth.Authenticate.authenticate()` method to perform the login.

- If you are using a remote, or perimeter, login system such as Security Assertion Markup Language (SAML), you do not need to call `weblogic.security.auth.Authenticate.authenticate()`. You only need to call the `authenticate()` method if you are using WebLogic Server to perform the logon.

  Note: WebLogic Server provides full container support for JAAS authentication and supports full use of JAAS authentication and authorization in application code.

- Within WebLogic Server, JAAS is called to perform the login. Each Authentication provider includes a LoginModule. This is true for servlet logins as well as Java client logins via JNDI or JAAS. The method WebLogic Server calls internally to perform the JAAS logon is `weblogic.security.services.Authentication.authenticate()`. When using the Authenticate class, `weblogic.security.SimpleCallbackHandler` may be a useful helper class.

- WebLogic Server supports the full JAAS 1.0 Reference Implementation with respect to authentication and authorization. While WebLogic Server does not protect any resources using JAAS authorization (it uses WebLogic security), you can use JAAS authorization in application code to protect the application's own resources.

For more information about JAAS, see the *Java Authentication and Authorization Service Developer's Guide* on the Web at `http://java.sun.com/security/jaas/doc/api.html`.

JAAS Authentication Development Environment

Whether the client is an application, applet, Enterprise JavaBean (EJB), or servlet that requires authentication, WebLogic Server uses the Java Authentication and Authorization Service (JAAS) classes to reliably and securely authenticate to the server. JAAS implements a Java version of the Pluggable Authentication Module (PAM) framework, which permits applications to remain independent from underlying authentication technologies. Therefore, the PAM framework allows the use of new or updated authentication technologies without requiring modifications to your Java application.

WebLogic Server uses JAAS for remote Java client authentication, and internally for authentication. Therefore, only developers of custom Authentication providers and developers of remote Java client applications need to be involved with JAAS directly. Users of Web browser clients or developers of within-container Java client applications (for example, those calling an Enterprise JavaBean (EJB) from a servlet) do not require the direct use or knowledge of JAAS.

Note: Both the Java Authentication and Authorization Service (JAAS) and the Java Naming And Directory Interface (JNDI) can be used by Java clients running on WebLogic Server to login to an instance of WebLogic Server in a secure manner, however, JAAS is preferred.

Note: In order to implement security in a WebLogic client you must install the WebLogic Server software distribution kit on the Java client.

The following topics are covered in this section:

- "JAAS Authentication APIs" on page 3-3
- "JAAS Client Application Components" on page 3-7
- "WebLogic LoginModule Implementation" on page 3-9

JAAS Authentication APIs

To implement Java clients that use JAAS authentication on WebLogic Server, you use a combination of Java SDK 1.4.1 application programming interfaces (APIs) and WebLogic APIs.

Table 3-1 lists and describes the Java SDK APIs packages used to implement JAAS authentication. The information in Table 3-1 is taken from the Java SDK API documentation and annotated to add WebLogic Server specific information. For more information on the Java SDK APIs, see the Javadocs at `http://java.sun.com/j2se/1.4.1/docs/api/index.html` and `http://java.sun.com/j2ee/1.4/docs/api/index.html`.

Table 3-2 lists and describes the WebLogic APIs used to implement JAAS authentication. For more information, see Javadocs for WebLogic Classes.

Table 3-1 Java SDK JAAS APIs

Java SDK JAAS API	Description
`javax.security.auth.Subject`	The `Subject` class represents the source of the request and can be an individual user or a group. The `Subject` object is created only after the user is successfully logged in.
`javax.security.auth.login. LoginContext`	The `LoginContext` class describes the basic methods used to authenticate `Subjects` and provides a way to develop an application independent of the underlying authentication technology. A `Configuration` specifies the authentication technology, or LoginModule, to be used with a particular application. Therefore, different LoginModules can be plugged in under an application without requiring any modifications to the application itself. Once the caller has instantiated a `LoginContext`, it invokes the `login` method to authenticate a `Subject`. This `login` method invokes the `login` method from each of the LoginModules configured for the name specified by the caller. If the `login` method returns without throwing an exception, then the overall authentication succeeded. The caller can then retrieve the newly authenticated `Subject` by invoking the `getSubject` method. Principals and credentials associated with the `Subject` may be retrieved by invoking the `Subject's` respective `getPrincipals`, `getPublicCredentials`, and `getPrivateCredentials` methods. To logout the `Subject`, the caller simply needs to invoke the `logout` method. As with the `login` method, this `logout` method invokes the `logout` method for each LoginModule configured for this `LoginContext`. For a sample implementation of this class, see Listing 3-4.

Table 3-1 Java SDK JAAS APIs

Java SDK JAAS API	Description
`javax.security.auth.login.Configuration`	This is an abstract class for representing the configuration of LoginModules under an application. The `Configuration` specifies which LoginModules should be used for a particular application, and in what order the LoginModules should be invoked. This abstract class needs to be subclassed to provide an implementation which reads and loads the actual configuration. In WebLogic Server, use a login configuration file instead of this class. For a sample configuration file, see Listing 3-3. By default, WebLogic Server uses the Sun Microsystems, Inc. configuration class, which reads from a configuration file.
`javax.security.auth.spi.LoginModule`	`LoginModule` describes the interface implemented by authentication technology providers. `LoginModules` are plugged in under applications to provide a particular type of authentication. While application developers write to the `LoginContext` API, authentication technology providers implement the LoginModule interface. A configuration specifies the LoginModule(s) to be used with a particular login application. Therefore, different LoginModules can be plugged in under the application without requiring any modifications to the application itself. **Note:** WebLogic Server provides an implementation of the LoginModule (`weblogic.security.auth.login.UsernamePasswordLoginModule`). BEA recommends that you use this implementation for JAAS authentication in WebLogic Server Java clients, however, you can develop your own LoginModule. Listing 3-3 shows how to call the WebLogic Server LoginModule.

Table 3-1 Java SDK JAAS APIs

Java SDK JAAS API	Description
`javax.security.auth.callback.Callback`	Implementations of this interface are passed to a `CallbackHandler`, allowing underlying security services the ability to interact with a calling application to retrieve specific authentication data, such as usernames and passwords, or to display certain information, such as error and warning messages. `Callback` implementations do not retrieve or display the information requested by underlying security services. `Callback` implementations simply provide the means to pass such requests to applications, and for applications, if appropriate, to return requested information back to the underlying security services. For a sample implementation of this interface, see Listing 3-2.
`javax.security.auth.callback.CallbackHandler`	An application implements a `CallbackHandler` and passes it to underlying security services so that they may interact with the application to retrieve specific authentication data, such as usernames and passwords, or to display certain information, such as error and warning messages. `CallbackHandlers` are implemented in an application-dependent fashion. Underlying security services make requests for different types of information by passing individual `Callbacks` to the `CallbackHandler`. The `CallbackHandler` implementation decides how to retrieve and display information depending on the `Callbacks` passed to it. For example, if the underlying service needs a username and password to authenticate a user, it uses a `NameCallback` and `PasswordCallback`. The `CallbackHandler` can then choose to prompt for a username and password serially, or to prompt for both in a single window. For a sample implementation of this interface, see Listing 3-2.

Table 3-2 WebLogic JAAS APIs

WebLogic JAAS API	Description
weblogic.security.auth.Authenticate	An authentication class that is used to authenticate user credentials. The WebLogic implementation of the LoginModule (`weblogic.security.auth.login.UsernamePasswordLoginModule`) uses this class to authenticate a user and add `Principals` to the `Subject`. User-written LoginModules must also use this class for the same purpose.
`weblogic.security.auth.Callback.URLCallback`	Underlying security services use this class to instantiate and pass a `URLCallback` to the `invokeCallback` method of a `CallbackHandler` to retrieve URL information. The WebLogic implementation of the `LoginModule` (`weblogic.security.auth.login.UsernamePasswordLoginModule`) uses this class. **Note:** Application developers should not use this class to retrieve URL information. Instead, they should use the `weblogic.security.URLCallbackHandler`.
`weblogic.security.Security`	This class implements the WebLogic Server client `runAs` methods. Client applications use the `runAs` methods to associate their `Subject` identity with the `PrivilegedAction` or `PrivilegedExceptionAction` that they execute. For a sample implementation, see Listing 3-6.
`weblogic.security.URLCallbackHandler`	The class used by application developers for returning a `username`, `password` and `URL`. Application developers should use this class to handle the `URLCallback` to retrieve URL information.

JAAS Client Application Components

At a minimum, a JAAS authentication client application comprises the following components:

- Java client

  The Java client instantiates a `LoginContext` object and invokes the login by calling the object's `login()` method. The `login()` method calls methods in each LoginModule to perform the login and authentication.

The LoginContext also instantiates a new empty `javax.security.auth.Subject` object (which represents the user or service being authenticated), constructs the configured LoginModule, and initializes it with this new `Subject` and `CallbackHandler`.

The LoginContext subsequently retrieves the authenticated Subject by calling the LoginContext's `getSubject` method. The LoginContext uses the `weblogic.security.Security` class `runAs()` method to associate the `Subject` identity with the `PrivilegedAction` or `PrivilegedExceptionAction` to be executed on behalf of the user identity.

- LoginModule

  The LoginModule utilizes the `CallbackHandler` to obtain the user name and password and checks that the name and password are the ones it expects.

  If authentication is successful, the LoginModule populates the Subject with a Principal representing the user. The Principal the LoginModule places in the Subject is an instance of `Principal`, which is a class implementing the `java.security.Principal` interface.

  LoginModule files can be written to perform different types of authentication, including username/password authentication and certificate authentication. A client application can include one LoginModule (the minimum requirement) or several LoginModules.

  Note: Use of the JAAS `javax.security.auth.Subject.doAs` methods in WebLogic Server applications do not associate the Subject with the client actions. You may use the `doAs` methods to implement J2SE security in WebLogic Server applications, but such usage is independent of the need to use the `Security.runAs()` method.

- Callbackhandler

  The `CallbackHandler` implements the `javax.security.auth.callback.CallbackHandler` interface. The LoginModule uses the `CallbackHandler` to communicate with the user and obtain the requested information, such as the username and password.

- Configuration file

  This file configures the LoginModule(s) to be used in the application. It specifies the location of the LoginModule(s) and, if there are multiple LoginModules, the order in which they are to be executed. Use of this file enables Java applications to remain independent from the authentication technologies, which are defined and implemented using the LoginModule.

- Action file

  This file defines the operations that the client application will perform.

- `ant` build script (`build.xml`)

  This script compiles all the files required for the application and deploys them to the WebLogic Server applications directories.

For a complete working JAAS authentication client that implements the components described here, see the JAAS sample application in the *`SAMPLES_HOME`*`\server\examples\src\examples\security\jaas` directory provided with WebLogic Server.

For more information on the basics of JAAS authentication, see Sun's *JAAS Authentication Tutorial* available at `http://java.sun.com/j2se/1.4/docs/guide/security/jaas/tutorials/GeneralAcnOnly.html`.

WebLogic LoginModule Implementation

The WebLogic implementation of the `LoginModule` class is provided in the WebLogic Server distribution in the `weblogic.jar` file, located in the `WL_HOME\server\lib` directory.

Note: WebLogic Server supports all callback types defined by JAAS as well as all callback types that extend the JAAS specification.

The `UsernamePasswordLoginModule` that is part of the WebLogic Server product checks for existing system user authentication definitions prior to execution and does nothing if they are already defined.

For more information about implementing JAAS LoginModules, see the *Java Authentication and Authorization Service Developer's Guide*.

JVM-Wide Default User and the runAs() Method

The first time you use the implementation of the LoginModule provided by WebLogic Server (`weblogic.security.auth.login.UsernamePasswordLoginModule`) to logon, the specified user becomes the machine-wide default user for the JVM (Java virtual machine). When you execute the `weblogic.security.Security.runAs()` method, it associates the specified `Subject` with the current thread's access permissions and then executes the action. If a specified `Subject` represents a non-privileged user (users that are not assigned to any groups are considered non-privileged), the JVM-wide default user is used. Therefore, it is important make sure that the `runAs()` method specifies the desired `Subject`. You can do this using one of the following options:

- ***Option 1***: If the client has control of `main()`, implement the wrapper code shown in Listing 3-1 in the client code.

Listing 3-1 runAs() Method Wrapper Code

```
import java.security.PrivilegedAction;
import javax.security.auth.Subject;
import weblogic.security.Security;

public class client
{
  public static void main(String[] args)
  {
  Security.runAs(new Subject(),
    new PrivilegedAction() {
     public Object run() {
      //
      //If implementing in client code, main() goes here.
      //
     return null;
     }
   });
  }
}
```

- ***Option 2:*** If the client does not have control of `main()`, implement the wrapper code shown in Listing 3-1 on each thread's `run()` method.

Writing a Client Application Using JAAS Authentication

To use JAAS in a WebLogic Server Java client to authenticate a subject, perform the following procedure:

1. Implement `LoginModule` classes for the authentication mechanisms you want to use with WebLogic Server. You will need a LoginModule class for each type of authentication mechanism. You can have multiple LoginModule classes for a single WebLogic Server deployment. For information on how to implement the `LoginModule` class, see the *Java*

Authentication and Authorization Service (JAAS) 1.0 Developer's Guide available at `http://java.sun.com/security/jaas/doc/api.html`

Note: BEA recommends that you use the implementation of the LoginModule provided by WebLogic Server (`weblogic.security.auth.login.UsernamePasswordLoginModule`) for username/password authentication. If you so desire, you can write your own `LoginModule` for username/password authentication, however, *do not* attempt to modify the WebLogic Server `LoginModule` and reuse it. If you are going to write your own LoginModule, you must have it call the `weblogic.security.auth.Authenticate.authenticate()` method to perform the login. If you are using a remote login system such as SAML you do not need to call the `authenticate()` method. You only need to call `authenticate()` if you are using WebLogic Server to perform the logon.

The `weblogic.security.auth.Authenticate` class uses a JNDI Environment object for initial context as described in Table 3-3.

2. Implement the `CallbackHandler` class that the LoginModule will use to communicate with the user and obtain the requested information, such as the username, password, and URL. The URL can be the URL of a WebLogic cluster, providing the client with the benefits of server failover. See Listing 3-2 for the sample `CallbackHandler` used in the JAAS client sample provided in the WebLogic Server distribution.

Note: Instead of implementing your own `CallbackHandler` class, you can use either of two WebLogic-supplied `CallbackHandler` classes, `weblogic.security.SimpleCallbackHandler` or `weblogic.security.URLCallbackHandler`. For more information on these classes, see Javadocs for WebLogic Classes.

Listing 3-2 Implementation of the CallbackHandler Interface

```
package examples.security.jaas;

import java.io.*;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.callback.TextOutputCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.TextInputCallback;
import javax.security.auth.callback.NameCallback;
import weblogic.security.auth.callback.URLCallback;
import examples.utils.common.ExampleUtils;

/**
 * SampleCallbackHandler.java
 * Implementation of the CallbackHandler Interface
 *
 * @author Copyright (c) 2000-2002 by BEA Systems, Inc. All Rights
 * Reserved.
 */
class SampleCallbackHandler implements CallbackHandler
{
  private String username = null;
  private String password = null;
  private String url = null;

  public SampleCallbackHandler() { }

  public SampleCallbackHandler(String pUsername, String pPassword,
                               String pUrl)
  {
    username = pUsername;
    password = pPassword;
    url = pUrl;
  }

  public void handle(Callback[] callbacks) throws IOException,
                                     UnsupportedCallbackException
  {
    for(int i = 0; i < callbacks.length; i++)
    {
      if(callbacks[i] instanceof TextOutputCallback)
      {
        // Display the message according to the specified type
        TextOutputCallback toc = (TextOutputCallback)callbacks[i];
        switch(toc.getMessageType())
        {
        case TextOutputCallback.INFORMATION:
          ExampleUtils.log(toc.getMessage());
          break;
        case TextOutputCallback.ERROR:
          ExampleUtils.log("ERROR: " + toc.getMessage());
          break;
        case TextOutputCallback.WARNING:
          ExampleUtils.log("WARNING: " + toc.getMessage());
          break;
        default:
          throw new IOException("Unsupported message type: " +
                                   toc.getMessageType());
        }
      }
      else if(callbacks[i] instanceof NameCallback)
      {
        // If username not supplied on cmd line, prompt the user
        // for the username.
        NameCallback nc = (NameCallback)callbacks[i];
        if (ExampleUtils.isEmpty(username)) {
          System.err.print(nc.getPrompt());
          System.err.flush();
          nc.setName((new BufferedReader(new
                          InputStreamReader(System.in))).readLine());
        }
        else {
          ExampleUtils.log("username: "+username);
          nc.setName(username);
        }
      }
      else if(callbacks[i] instanceof URLCallback)
      {
        // If url not supplied on cmd line, prompt the user for the
        // url.
        // This example requires the url.
        URLCallback uc = (URLCallback)callbacks[i];
        if (ExampleUtils.isEmpty(url)) {
          System.err.print(uc.getPrompt());\
          System.err.flush();
          uc.setURL((new BufferedReader(new
                         InputStreamReader(System.in))).readLine());\
        }
        else {
          ExampleUtils.log("URL: "+url);
          uc.setURL(url);
        }
      }
      else if(callbacks[i] instanceof PasswordCallback)
      {
        PasswordCallback pc = (PasswordCallback)callbacks[i];

        // If password not supplied on cmd line, prompt the user
        // for the password.
        if (ExampleUtils.isEmpty(password)) {
          System.err.print(pc.getPrompt());
          System.err.flush();

          // Note: JAAS specifies that the password is a char[]
          // rather than a String.
          String tmpPassword = (new BufferedReader(new
                           InputStreamReader(System.in))).readLine();
          int passLen = tmpPassword.length();
          char[] passwordArray = new char[passLen];
          for(int passIdx = 0; passIdx < passLen; passIdx++)
            passwordArray[passIdx] = tmpPassword.charAt(passIdx);
          pc.setPassword(passwordArray);
        }
        else {
          String tPass = new String();
          for(int p = 0; p < password.length(); p++)
            tPass += "*";
          ExampleUtils.log("password: "+tPass);
          pc.setPassword(password.toCharArray());
        }
      }
      else if(callbacks[i] instanceof TextInputCallback)
      {
        // Prompt the user for the username
        TextInputCallback callback =
                              (TextInputCallback)callbacks[i];
        System.err.print(callback.getPrompt());
        System.err.flush();
        callback.setText((new BufferedReader(new
                           InputStreamReader(System.in))).readLine());
      }
      else
      {
        throw new UnsupportedCallbackException(callbacks[i],
                                           "Unrecognized Callback");
      }
    }
  }
}
```

3. Write a configuration file that specifies which LoginModule classes should be used for your WebLogic Server and in which order the LoginModule classes should be invoked. See Listing 3-3 for the sample configuration file used in the JAAS client sample provided in the WebLogic Server distribution.

Listing 3-3 sample_jaas.config Code Example

```
/** Login Configuration for the JAAS Sample Application **/

Sample {
   weblogic.security.auth.login.UsernamePasswordLoginModule
         required debug=false;
};
```

4. In the Java client, write code to instantiate a `LoginContext`. The `LoginContext` consults the configuration file, `sample_jaas.config`, to load the default LoginModule configured for WebLogic Server. See Listing 3-4 for an example `LoginContext` instantiation.

Listing 3-4 LoginContext Code Fragment

```
...
import javax.security.auth.login.LoginContext;
...
    LoginContext loginContext = null;

    try
    {
      // Create LoginContext; specify username/password login module
      loginContext = new LoginContext("Sample",
             new SampleCallbackHandler(username, password, url));
    }
```

> **Note:** If you use another means to authenticate the user such as an Identity Assertion provider or a remote instance of WebLogic Server, the default LoginModule is determined by the remote resource.

5. Invoke the `login()` method of the `LoginContext` instance. The `login()` method invokes all the loaded LoginModules. Each LoginModule attempts to authenticate the subject. The `LoginContext` throws a `LoginException` if the configured login conditions are not met. See Listing 3-5 for an example of the `login()` method.

Listing 3-5 Login() Method Code Fragment

```
...
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;
import javax.security.auth.login.FailedLoginException;
import javax.security.auth.login.AccountExpiredException;
import javax.security.auth.login.CredentialExpiredException;
...
 /**
     * Attempt authentication
     */
    try
    {
      // If we return without an exception, authentication succeeded
      loginContext.login();
    }
    catch(FailedLoginException fle)
    {
      System.out.println("Authentication Failed, " +
                              fle.getMessage());
      System.exit(-1);
    }
    catch(AccountExpiredException aee)
    {
      System.out.println("Authentication Failed: Account Expired");
      System.exit(-1);
    }
    catch(CredentialExpiredException cee)
    {
      System.out.println("Authentication Failed: Credentials
                               Expired");
      System.exit(-1);
    }
    catch(Exception e)
    {
      System.out.println("Authentication Failed: Unexpected
                             Exception, " + e.getMessage());
      e.printStackTrace();
      System.exit(-1);
    }
```

6. Write code in the Java client to retrieve the authenticated Subject from the `LoginContext` instance using the `javax.security.auth.Subject.getSubject() method` and call the action as the Subject. Upon successful authentication of a Subject, access controls can be placed upon that Subject by invoking the `weblogic.security.Security.runAs()` method. The `runAs()` method associates the specified Subject with the current thread's access permissions and then executes the action. See Listing 3-6 for an example implementation of the `getSubject()` and `runAs()` methods.

Note: Use of the JAAS `javax.security.auth.Subject.doAs` methods in WebLogic Server applications do not associate the Subject with the client actions. You may use the `doAs` methods to implement J2SE security in WebLogic Server applications, but such usage is independent of the need to use the `Security.runAs()` method.

Listing 3-6 getSubject() and runAs() Methods Code Fragment

```
...
/**
 * Retrieve authenticated subject, perform SampleAction as Subject
 */
   Subject subject = loginContext.getSubject();
   SampleAction sampleAction = new SampleAction(url);
   Security.runAs(subject, sampleAction);
   System.exit(0);
...
```

7. Write code to execute an action if the Subject has the required privileges. See Listing 3-7 for a sample implementation of the `javax.security.PrivilegedAction` class that executes an EJB to trade stocks.

Listing 3-7 Example of a PrivilegedAction Implementation

```
package examples.security.jaas;

import java.security.PrivilegedAction;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;
import javax.ejb.CreateException;
import javax.ejb.EJBException;
import javax.ejb.FinderException;
import javax.ejb.ObjectNotFoundException;
import javax.ejb.RemoveException;
import java.rmi.RemoteException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import examples.ejb20.basic.statelessSession.TraderHome;
import examples.ejb20.basic.statelessSession.Trader;
import examples.utils.common.ExampleUtils;

/**
 * SampleAction.java
 *
 * JAAS sample PrivilegedAction Implementation
 *
 * @author Copyright (c) 2000-2002 by BEA Systems, Inc. All Rights
 * Reserved.
 */
public class SampleAction implements PrivilegedAction
{
  private static final String JNDI_NAME =
            "ejb20-statelessSession-TraderHome";
  private String url;

  public SampleAction(String url)
  {
    this.url = url;
  }

  public Object run()
  {
    Object obj = null;

    try {
      callTraderEJB();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
    return obj;
  }
```

```
/**
 * Call Trader EJB.
 */
public void callTraderEJB()
  throws NamingException, CreateException, RemoteException,
             RemoveException
{
  TraderHome home = lookupTraderHome();

  // create a Trader
  ExampleUtils.log("Creating a trader");
  Trader trader = (Trader)ExampleUtils.narrow(home.create(),
                   Trader.class);

  String [] stocks = {"BEAS", "MSFT", "AMZN", "HWP" };

    // execute some buys
  for (int i=0; i<stocks.length; i++) {
    int shares = (i+1) * 100;
    ExampleUtils.log("Buying "+shares+" shares of
                   "+stocks[i]+".");
    trader.buy(stocks[i], shares);
  }

  // execute some sells
  for (int i=0; i<stocks.length; i++) {
    int shares = (i+1) * 100;
    ExampleUtils.log("Selling "+shares+" shares of
                     "+stocks[i]+".");
    trader.sell(stocks[i], shares);
  }

  // remove the Trader
  ExampleUtils.log("Removing the trader");
  trader.remove();
}

/**
 * Look up the bean's home interface using JNDI.
 */
private TraderHome lookupTraderHome()
```

```
      throws NamingException
    {
      Context ctx = ExampleUtils.getInitialContext(url);
      Object home = (TraderHome)ctx.lookup(JNDI_NAME);
      return (TraderHome)ExampleUtils.narrow(home, TraderHome.class);
    }
}
```

8. Invoke the `logout()` method of the `LoginContext` instance. The `logout()` method closes the user's session and clear the `Subject`. See Listing 3-8 for an example of the `login()` method.

Listing 3-8 logout() Method Code Example

```
...
import javax.security.auth.login.LoginContext;
...
try
    {
      System.out.println("logging out...");
      loginContext.logout();
    }
```

Note: The `LoginModule.logout()` method is never called for a WebLogic Authentication provider or a custom Authentication provider. This is simply because once the `Principals` are created and placed into a `Subject`, the WebLogic Security Framework no longer controls the lifecycle of the `Subject`. Therefore, the developer-written, user code that creates the JAAS `LoginContext` to login and obtain the `Subject` should also call the `LoginContext` to logout. Calling `LoginContext.logout()` results in the clearing of the `Principals` from the `Subject`.

Using JNDI Authentication

Java clients use the Java Naming and Directory Interface (JNDI) to pass credentials to WebLogic Server. A Java client establishes a connection with WebLogic Server by getting a JNDI `InitialContext`. The Java client then uses the `InitialContext` to look up the resources it needs in the WebLogic Server JNDI tree.

Note: JAAS is the preferred method of authentication, however, the WebLogic Authentication provider's LoginModule only supports user name and password authentication. Thus, for client certificate authentication (also referred to as two-way SSL authentication), you should use JNDI. To use JAAS for client certificate authentication, you must write a custom Authentication provider whose LoginModule does certificate authentication. For information on how to write LoginModules, see `http://java.sun.com/j2se/1.4.1/docs/guide/security/jaas/JAASLMDevGuide.html`.

To specify a user and the user's credentials, set the JNDI properties listed in Table 3-3.

Table 3-3 JNDI Properties Used for Authentication

Property	Meaning
`INITIAL_CONTEXT_FACTORY`	Provides an entry point into the WebLogic Server environment. The class `weblogic.jndi.WLInitialContextFactory` is the JNDI SPI for WebLogic Server.
`PROVIDER_URL`	Specifies the host and port of the WebLogic Server that provides the name service. For example: `t3://weblogic:7001`.
`SECURITY_PRINCIPAL`	Specifies the identity of the user when that user authenticates to the default (active) security realm.
`SECURITY_CREDENTIALS`	Specifies the credentials of the user when that user authenticates to the default (active) security realm.

These properties are stored in a hash table that is passed to the `InitialContext` constructor. Listing 3-9 illustrates how to use JNDI authentication in a Java client running on WebLogic Server.

Listing 3-9 Example of Authentication

```
...
Hashtable env = new Hashtable();
     env.put(Context.INITIAL_CONTEXT_FACTORY,
              "weblogic.jndi.WLInitialContextFactory");
     env.put(Context.PROVIDER_URL, “t3://weblogic:7001”);
     env.put(Context.SECURITY_PRINCIPAL, “javaclient”);
     env.put(Context.SECURITY_CREDENTIALS, “javaclientpassword”);
     ctx = new InitialContext(env);
```

Note: For information on JNDI contexts and threads and how to avoid potential JNDI context problems, see “JNDI Contexts and Threads” and “How to Avoid JNDI Context Problems” in the *Programming WebLogic JNDI*.

Java Client JAAS Authentication Code Examples

A complete working JAAS authentication sample is provided with the WebLogic Server product. The sample is located in the `*SAMPLES_HOME*\server\examples\src\examples\security\jaas` directory. For a description of the sample and instructions on how to build, configure, and run this sample, see the `package.html` file in the sample directory. You can modify this code example and reuse it.

Using SSL Authentication in Java Clients

The following topics are covered in this section:

JSSE and WebLogic Server

JSSE is a set of packages that support and implement the SSL and TLS v1 protocols, making those capabilities programmatically available. BEA WebLogic Server provides Secure Sockets Layer (SSL) support for encrypting data transmitted between WebLogic Server clients and servers, Java clients, Web browsers, and other servers.

WebLogic Server's Java Secure Socket Extension (JSSE) implementation can be used by WebLogic clients, but is not required. Other JSSE implementations can be used for their client-side code outside the server as well.

The following restrictions apply when using SSL in WebLogic server-side applications:

- The use of other (third-party) JSSE implementations to develop WebLogic server applications is not supported. The SSL implementation that WebLogic Server uses is static to the server configuration and is not replaceable by customer applications. You cannot plug different JSSE implementations into WebLogic Server to have it use those implementations for SSL.

- The WebLogic implementation of JSSE does support JCE Cryptographic Service Providers (CSPs), however, due to the inconsistent provider support for JCE, BEA cannot guarantee that untested providers will work out of the box. BEA has tested WebLogic Server with the following providers:
  - The default JCE provider (SunJCE provider) that is included with JDK 1.4.1.
  - The nCipher JCE provider.

  Other providers may work with WebLogic Server, but an untested provider is not likely to work out of the box. For more information on using the JCE providers supported by WebLogic Server, see Configuring SSL in *Managing WebLogic Security.*

WebLogic Server uses the HTTPS port for SSL. Only SSL can be used on that port. SSL encrypts the data transmitted between the client and WebLogic Server so that the username and password do not flow in clear text.

Note: In order to implement security in a WebLogic client, you must install the WebLogic Server software distribution kit on the Java client.

Using JNDI Authentication

Java clients use the Java Naming and Directory Interface (JNDI) to pass credentials to WebLogic Server. A Java client establishes a connection with WebLogic Server by getting a JNDI `InitialContext`. The Java client then uses the `InitialContext` to look up the resources it needs in the WebLogic Server JNDI tree.

Note: JAAS is the preferred method of authentication, however, the Authentication provider's LoginModule only supports username and password authentication. Thus, for client certificate authentication (also referred to as two-way SSL authentication), you should use JNDI. To use JAAS for client certificate authentication, you must write a custom Authentication provider whose LoginModule does certificate authentication.

To specify a user and the user's credentials, set the JNDI properties listed in the Table 4-1.

Table 4-1 JNDI Properties Used for Authentication

Property	Meaning
`INITIAL_CONTEXT_FACTORY`	Provides an entry point into the WebLogic Server environment. The class `weblogic.jndi.WLInitialContextFactory` is the JNDI SPI for WebLogic Server.
`PROVIDER_URL`	Specifies the host and port of the WebLogic Server that provides the name service. For example: `t3s://weblogic:7002`.
`SECURITY_PRINCIPAL`	Specifies the identity of the user when that user authenticates to the default (active) security realm.

These properties are stored in a hash table that is passed to the `InitialContext` constructor. Notice the use of t3s, which is a WebLogic Server proprietary version of SSL. t3s uses encryption to protect the connection and communication between WebLogic Server and the Java client.

Listing 4-1 demonstrates how to use one-way SSL certificate authentication in a Java client. For a two-SSL authentication code example, see Listing 4-5, "Example of a Two-Way SSL Authentication Client That Uses JNDI," on page 4-20.

Listing 4-1 Example of One-Way SSL Authentication Using JNDI

```
...
Hashtable env = new Hashtable();
      env.put(Context.INITIAL_CONTEXT_FACTORY,
              "weblogic.jndi.WLInitialContextFactory");
      env.put(Context.PROVIDER_URL, "t3s://weblogic:7002");
      env.put(Context.SECURITY_PRINCIPAL, "javaclient");
      env.put(Context.SECURITY_CREDENTIALS, "javaclientpassword");
      ctx = new InitialContext(env);
```

Note: For information on JNDI contexts and threads and how to avoid potential JNDI context problems, see "JNDI Contexts and Threads" and "How to Avoid JNDI Context Problems" in the *Programming WebLogic JNDI*.

SSL Certificate Authentication Development Environment

The following topics are covered in this section:

- "SSL Authentication APIs" on page 4-4
- "SSL Client Application Components" on page 4-8

SSL Authentication APIs

To implement Java clients that use SSL authentication on WebLogic Server, you use a combination of Java SDK 1.3 application programming interfaces (APIs) and WebLogic APIs.

Table 4-2 lists and describes the Java SDK APIs packages used to implement certificate authentication. The information in Table 4-2 is taken from the Java SDK API documentation and annotated to add WebLogic Server specific information. For more information on the Java SDK APIs, see the Javadocs at `http://java.sun.com/j2se/1.4.1/docs/api/index.html` and `http://java.sun.com/j2ee/1.4/docs/api/index.html`.

Table 4-3 lists and describes the WebLogic APIs used to implement certificate authentication. For more information, see Javadocs for WebLogic Classes.

Table 4-2 Java SDK Certificate APIs

Java SDK Certificate APIs	Description
`javax.crypto`	This package provides the classes and interfaces for cryptographic operations. The cryptographic operations defined in this package include encryption, key generation and key agreement, and Message Authentication Code (MAC) generation. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. This package also supports secure streams and sealed objects. Many classes provided in this package are provider-based (see the `java.security.Provider` class). The class itself defines a programming interface to which applications may be written. The implementations themselves may then be written by independent third-party vendors and plugged in seamlessly as needed. Therefore, application developers may take advantage of any number of provider-based implementations without having to add or rewrite code.
`javax.net`	This package provides classes for networking applications. These classes include factories for creating sockets. Using socket factories you can encapsulate socket creation and configuration behavior.
`javax.net.SSL`	While the classes and interfaces in this package are supported by WebLogic Server, BEA recommends that you use the `weblogic.security.SSL` package when you use SSL with WebLogic Server.
`java.security.cert`	This package provides classes and interfaces for parsing and managing certificates, certificate revocation lists (CRLs), and certification paths. It contains support for X.509 v3 certificates and X.509 v2 CRLs.

Table 4-2 Java SDK Certificate APIs (Continued)

Java SDK Certificate APIs	Description
`java.security.KeyStore`	This class represents an in-memory collection of keys and certificates. It is used to manage two types of keystore entries: • Key Entry This type of keystore entry holds very sensitive cryptographic key information, which is stored in a protected format to prevent unauthorized access. Typically, a key stored in this type of entry is a secret key, or a private key accompanied by the certificate chain for the corresponding public key. Private keys and certificate chains are used by a given entity for self-authentication. Applications for this authentication include software distribution organizations which sign JAR files as part of releasing and/or licensing software. • Trusted Certificate Entry This type of entry contains a single public key certificate belonging to another party. It is called a trusted certificate because the keystore owner trusts that the public key in the certificate indeed belongs to the identity identified by the subject (owner) of the certificate. This type of entry can be used to authenticate other parties.
`java.security.PrivateKey`	A private key. This interface contains no methods or constants. It merely serves to group (and provide type safety for) all private key interfaces. **Note:** The specialized private key interfaces extend this interface. For example, see the `DSAPrivateKey` interface in `java.security.interfaces`.

Table 4-2 Java SDK Certificate APIs (Continued)

Java SDK Certificate APIs	Description
`java.security.Provider`	This class represents a "Cryptographic Service Provider" for the Java Security API, where a provider implements some or all parts of Java Security, including: • Algorithms (such as DSA, RSA, MD5 or SHA-1). • Key generation, conversion, and management facilities (such as for algorithm-specific keys). Each provider has a name and a version number, and is configured in each runtime it is installed in. To supply implementations of cryptographic services, a team of developers or a third-party vendor writes the implementation code and creates a subclass of the `Provider` class.
`javax.servlet.http.HttpServletRequest`	This interface extends the `ServletRequest` interface to provide request information for HTTP servlets. The servlet container creates an `HttpServletRequest` object and passes it as an argument to the servlet's service methods (`doGet`, `doPost`, and so on.).
`javax.servlet.http.HttpServletResponse`	This interface extends the `ServletResponse` interface to provide HTTP-specific functionality in sending a response. For example, it has methods to access HTTP headers and cookies. The servlet container creates an `HttpServletRequest` object and passes it as an argument to the servlet's service methods (`doGet`, `doPost`, and so on.).
`javax.servlet.ServletOutputStream`	This class provides an output stream for sending binary data to the client. A `ServletOutputStream` object is normally retrieved via the `ServletResponse.getOutputStream()` method. This is an abstract class that the servlet container implements. Subclasses of this class must implement the `java.io.OutputStream.write(int)` method.
`javax.servlet.ServletResponse`	This class defines an object to assist a servlet in sending a response to the client. The servlet container creates a `ServletResponse` object and passes it as an argument to the servlet's service methods (`doGet`, `doPost`, and so on.).

Table 4-3 WebLogic Certificate APIs

WebLogic Certificate APIs	Description
weblogic.net.http. HttpsURLConnection	This class is used to represent a Hyper-Text Transfer Protocol with SSL (HTTPS) connection to a remote object. This class is used to make an outbound SSL connection from a WebLogic Server acting as a client to another WebLogic Server.
`weblogic.security.SSL. HostnameVerifierJSSE`	This interface provides a callback mechanism so that implementers of this interface can supply a policy for handling the case where the host that's being connected to and the server name from the certificate `SubjectDN` must match. To specify an instance of this interface to be used by the server, set the class for the custom host name verifier in the Client Attributes fields that are located on the Advanced Options pane under the Keystore & SSL tab for the server (for example, `myserver`).
`weblogic.security.SSL. TrustManagerJSSE`	This interface permits the user to override certain validation errors in the peer's certificate chain and allow the handshake to continue. This interface also permits the user to perform additional validation on the peer certificate chain and interrupt the handshake if need be.
`weblogic.security.SSL. SSLContext`	This class holds all of the state information shared across all sockets created under that context.
`weblogic.security.SSL. SSLSocketFactory`	This class delegates requests to create SSL sockets to the `SSLSocketFactory`.

SSL Client Application Components

At a minimum, an SSL client application comprises the following components:

- Java client

  Typically, a Java client performs these functions:

  – Initializes an `SSLContext` with client identity, a `HostnameVerifierJSSE`, a `TrustManagerJSSE`, and a `HandshakeCompletedListener`.

  – Creates a keystore and retrieves the private key and certificate chain

  – Uses an SSLSocketFactory

  – Uses HTTPS to connect to a JSP served by an instance of WebLogic Server

- HostnameVerifier

  The HostnameVerifier implements the `weblogic.security.SSL.HostnameVerifierJSSE interface`. It provides a callback mechanism so that implementers of this interface can supply a policy for handling the case where the host that is being connected to and the server name from the certificate Subject Distinguished Name (SubjectDN) must match.

- HandshakeCompletedListener

  The HandshakeCompletedListener implements the `javax.net.ssl.HandshakeCompletedListener` interface. It defines how the SSL client receives notifications about the completion of an SSL handshake on a given SSL connection. It also defines the number of times an SSL handshake takes place on a given SSL connection.

- TrustManager

  The TrustManager implements the `weblogic.security.SSL.TrustManagerJSSE` interface. It builds a certificate path to a trusted root and returns true if it can be validated and is trusted for client SSL authentication.

- `ant` build script (`build.xml`)

  This script compiles all the files required for the application and deploys them to the WebLogic Server applications directories.

For a complete working SSL authentication client that implements the components described here, see the SSLClient sample application in the *`SAMPLES_HOME`*`\server\examples\src\examples\security\sslclient` directory provided with WebLogic Server.

For more information on JSSE authentication, see Sun's *Java Secure Socket Extension (JSSE) 1.0.3 API User's Guide* available at `http://java.sun.com/products/jsse/doc/guide/API_users_guide.html`.

Writing Applications that Use SSL

This section covers the following topics:

- "Communicating Securely From WebLogic Server to Other WebLogic Servers" on page 4-10

- "Writing SSL Clients" on page 4-10
- "Using Two-Way SSL Authentication" on page 4-19
- "Two-Way SSL Authentication with JNDI" on page 4-20
- "Using a Custom Host Name Verifier" on page 4-26
- "Using a Trust Manager" on page 4-28
- "Using an SSLContext" on page 4-31
- "Using an SSL Server Socket Factory" on page 4-32
- "Using URLs to Make Outbound SSL Connections" on page 4-33

Communicating Securely From WebLogic Server to Other WebLogic Servers

You can use a URL object to make an outbound SSL connection from a WebLogic Server instance acting as a client to another WebLogic Server instance. The `weblogic.net.http.HttpsURLConnection` class provides a way to specify the security context information for a client, including the digital certificate and private key of the client.

The `weblogic.net.http.HttpsURLConnection` class provides methods for determining the negotiated cipher suite, getting/setting a host name verifier, getting the server's certificate chain, and getting/setting an `SSLSocketFactory` in order to create new SSL sockets.

The SSLClient code example uses the `weblogic.net.http.HttpsURLConnection` class to make an outbound SSL connection. The SSLClient code example is available in the `examples.security.sslclient` package in the `SAMPLES_HOME\server\examples\src\examples\security\sslclient` directory.

Writing SSL Clients

This section describes, by way of example, how to write various types of SSL clients. Examples of the following types of SSL clients are provided:

- "SSLClient Sample" on page 4-11
- "SSLSocketClient Sample" on page 4-15
- "SSLClientServlet Sample" on page 4-18

SSL Client License Requirement: Any stand-alone Java client that uses WebLogic SSL classes (`weblogic.security.SSL`) to invoke an Enterprise JavaBean (EJB) must use the BEA license file. When you run your client application, set the following system properties on the command line:

- *`bea.home=license_file_directory`*
- `java.protocol.handler.pkgs=com.certicom.net.ssl`

Where *`license_file_directory`* refers to the directory that contains the BEA license file (`license.bea`). Here is an example of a run command that uses the default location of the license file (`c:\bea`):

```
java -Dbea.home=c:\bea \
  -Djava.protocol.handler.pkgs=com.certicom.net.ssl  my_app
```

SSLClient Sample

The SSLClient sample demonstrates how to use the WebLogic SSL library to make outgoing SSL connections using `URL` and `URLConnection` objects. It shows both how to do this from a stand-alone application as well as from a servlet in WebLogic Server.

Note: When making an outgoing SSL connection, a WebLogic Server instance uses the server's certificate. When communicating to either the same or another WebLogic Server instance with two-way SSL, the originating server's certificate will be verified against the client root CA list in the receiving WebLogic Server instance.

Listing 4-2 shows a sample SSLClient. This code is taken from the `SSLClient.java` file located at *`SAMPLES_HOME`*`\server\examples\src\examples\security\sslclient`.

Listing 4-2 SSL Client Sample Code

```
package examples.security.sslclient;

import java.io.File;
import java.net.URL;
import java.io.IOException;
import java.io.InputStream;
import java.io.FileInputStream;
import java.io.OutputStream;
import java.io.PrintStream;
import java.util.Hashtable;
import java.security.Provider;
import javax.naming.NamingException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.servlet.ServletOutputStream;
import weblogic.net.http.*;
import weblogic.jndi.Environment;

/** SSLClient is a short example of how to use the SSL library of
 * WebLogic to make outgoing SSL connections. It shows both how to
 * do this from a stand-alone application as well as from within
 * WebLogic (in a Servlet).
 *
 *    Be careful to notice that the WebLogic Server, when making an
 *    outgoing SSL connection, will use that instance of the server's
 *    certificate. When communicating to either the same or another
 *    WebLogic Server with two-way SSL, the originating server's
 *    certificate will be verified against the client root CA list in
 *    the receiving WebLogic Server.
 *
 * @author Copyright (c) 1999-2002 by BEA Systems, Inc. All Rights Reserved.
 */

public class SSLClient {
  public void SSLClient() {}
  public static void main (String [] argv)
    throws IOException {
    if ((!((argv.length == 4) || (argv.length == 5))) ||
        (!(argv[0].equals("wls")))
        ) {
      System.out.println("example: java SSLClient wls
                server2.weblogic.com 80 443 /examplesWebApp/SnoopServlet.jsp");
      System.exit(-1);
    }

    try {
      System.out.println("----");
      if (argv.length == 5) {
        if (argv[0].equals("wls"))
          wlsURLConnect(argv[1], argv[2], argv[3], argv[4], System.out);
      } else {  // for null query, default page returned...
        if (argv[0].equals("wls"))
          wlsURLConnect(argv[1], argv[2], argv[3], null, System.out);
      }
      System.out.println("----");
    } catch (Exception e) {
      e.printStackTrace();
      printSecurityProviders(System.out);
      System.out.println("----");
    }
  }
```

```
private static void printOut(String outstr, OutputStream stream) {
  if (stream instanceof PrintStream) {
    ((PrintStream)stream).print(outstr);
    return;
  } else if (stream instanceof ServletOutputStream) {
    try {
      ((ServletOutputStream)stream).print(outstr);
      return;
    } catch (IOException ioe) {
      System.out.println(" IOException: "+ioe.getMessage());
    }
  }
  System.out.print(outstr);
}
private static void printSecurityProviders(OutputStream stream) {
  StringBuffer outstr = new StringBuffer();
  outstr.append(" JDK Protocol Handlers and Security Providers:\n");
  outstr.append("   java.protocol.handler.pkgs - ");
  outstr.append(System.getProperties().getProperty(
                                    "java.protocol.handler.pkgs"));
  outstr.append("\n");
  Provider[] provs = java.security.Security.getProviders();
  for (int i=0; i<provs.length; i++)
      outstr.append("   provider[" + i + "] - " + provs[i].getName() +
                    " - " + provs[i].getInfo() + "\n");
  outstr.append("\n");
  printOut(outstr.toString(), stream);
}
private static void tryConnection(java.net.HttpURLConnection connection,
                                  OutputStream stream)
  throws IOException {
    connection.connect();
    String responseStr = "\t\t" +
                         connection.getResponseCode() + " -- " +
                         connection.getResponseMessage() + "\n\t\t" +
                         connection.getContent().getClass().getName() + "\n";
    connection.disconnect();
    printOut(responseStr, stream);
}
/*
 * This method contains an example of how to use the URL and
 *  URLConnection objects to create a new SSL connection, using
 *  WebLogic SSL client classes.
 */
public static void wlsURLConnect(String host, String port,
                                 String sport, String query,
                                 OutputStream out) {
  try {
    if (query == null)
```

```
      query = "/examplesWebApp/index.jsp";
    // The following protocol registration is taken care of in the
    //  normal startup sequence of WebLogic. It can be turned off
    //  using the console SSL panel.
    //
    // We duplicate it here as a proof of concept in a stand alone
    //  java application. Using the URL object for a new connection
    //  inside of WebLogic would work as expected.
    java.util.Properties p = System.getProperties();
    String s = p.getProperty("java.protocol.handler.pkgs");
    if (s == null) {
      s = "weblogic.net";
    } else if (s.indexOf("weblogic.net") == -1) {
      s += "|weblogic.net";
    }
    p.put("java.protocol.handler.pkgs", s);
    System.setProperties(p);
    printSecurityProviders(out);
    // end of protocol registration
    printOut(" Trying a new HTTP connection using WLS client classes -
              \n\thttp://" + host + ":" + port + query + "\n", out);
    URL wlsUrl = null;
    try {
     wlsUrl = new URL("http", host, Integer.valueOf(port).intValue(), query);
      weblogic.net.http.HttpURLConnection connection =
        new weblogic.net.http.HttpURLConnection(wlsUrl);
      tryConnection(connection, out);
    } catch (Exception e) {
      printOut(e.getMessage(), out);
      e.printStackTrace();
      printSecurityProviders(System.out);
      System.out.println("----");
    }
    printOut(" Trying a new HTTPS connection using WLS client classes -
               \n\thttps://" + host + ":" + sport + query + "\n", out);
   wlsUrl = new URL("https", host, Integer.valueOf(sport).intValue(), query);
    weblogic.net.http.HttpsURLConnection sconnection =
      new weblogic.net.http.HttpsURLConnection(wlsUrl);

   // Only when you have configured a two-way SSL connection, i.e.
   // Client Certs Requested and Enforced is selected in Two Way Client Cert
   // Behavior field in the Server Attributes
   // that are located on the Advanced Options pane under Keystore & SSL
   // tab on the server, the following private key and the client cert chain
   // is used.

    File ClientKeyFile  = new File ("clientkey.pem");
    File ClientCertsFile  = new File ("client2certs.pem");
    if (!ClientKeyFile.exists() || !ClientCertsFile.exists())
```

```
{
   System.out.println("Error : clientkey.pem/client2certs.pem
                           is not present in this directory.");
   System.out.println("To create it run - ant createmycerts.");
   System.exit(0);
}
      InputStream [] ins = new InputStream[2];
      ins[0] = new FileInputStream("client2certs.pem");
      ins[1] = new FileInputStream("clientkey.pem");
      String pwd = "clientkey";
      sconnection.loadLocalIdentity(ins[0], ins[1], pwd.toCharArray());
      tryConnection(sconnection, out);
    } catch (Exception ioe) {
      printOut(ioe.getMessage(), out);
      ioe.printStackTrace();
    }
  }
}
```

SSLSocketClient Sample

The SSLSocketClient sample demonstrates how to use SSL sockets to go directly to the secure port to connect to a JSP served by an instance of WebLogic Server and display the results of that connection. It shows how to implement the following functions:

- Initializing an `SSLContext` with client identity, a `HostnameVerifierJSSE`, and a `TrustManagerJSSE`
- Creating a keystore and retrieving the private key and certificate chain
- Using an `SSLSocketFactory`
- Using HTTPS to connect to a JSP served by WebLogic Server
- Implementing the `javax.net.ssl.HandshakeCompletedListener` interface
- Creating a dummy implementation of the `weblogic.security.SSL.HostnameVerifierJSSE` class to verify that the server the example connects to is running on the desired host

Listing 4-3 shows a sample SSLSocketClient. This code taken from the `SSLSocketClient.java` file located at `SAMPLES_HOME\server\examples\src\examples\security\sslclient`.

Listing 4-3 SSLSocketClient Sample Code

```
package examples.security.sslclient;

import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.FileInputStream;
import java.util.Hashtable;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.cert.Certificate;
import weblogic.security.SSL.HostnameVerifierJSSE;
import weblogic.security.SSL.SSLContext;
import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSession;
import weblogic.security.SSL.SSLSocketFactory;
import weblogic.security.SSL.TrustManagerJSSE;

  /**
  * A Java client demonstrates connecting to a JSP served by WebLogic Server
  * using the secure port and displays the results of the connection.
  *
  * @author Copyright (c) 1999-2002 by BEA Systems, Inc. All Rights Reserved.
  */

public class SSLSocketClient {
  public void SSLSocketClient() {}
  public static void main (String [] argv)
    throws IOException {
    if ((argv.length < 2) || (argv.length > 3)) {
      System.out.println("usage:   java SSLSocketClient host sslport
                                 <HostnameVerifierJSSE>");
     System.out.println("example: java SSLSocketClient server2.weblogic.com 443
                                 MyHVClassName");
      System.exit(-1);
    }
    try {
      System.out.println("\nhttps://" + argv[0] + ":" + argv[1]);
      System.out.println(" Creating the SSLContext");
      SSLContext sslCtx = SSLContext.getInstance("https");

      File KeyStoreFile  = new File ("mykeystore");
      if (!KeyStoreFile.exists())
{
   System.out.println("Keystore Error : mykeystore is not present in this
                          directory.");
   System.out.println("To create it run - ant createmykeystore.");
   System.exit(0);
}

      System.out.println(" Initializing the SSLContext with client\n" +
                         "  identity (certificates and private key),\n" +
                         "  HostnameVerifierJSSE, AND NulledTrustManager");

     // Open the keystore, retrieve the private key, and certificate chain
      KeyStore ks = KeyStore.getInstance("jks");
      ks.load(new FileInputStream("mykeystore"), null);
      PrivateKey key = (PrivateKey)ks.getKey("mykey", "testkey".toCharArray());
      Certificate [] certChain = ks.getCertificateChain("mykey");
      sslCtx.loadLocalIdentity(certChain, key);

      HostnameVerifierJSSE hVerifier = null;
      if (argv.length < 3)
        hVerifier = new NulledHostnameVerifier();
      else
       hVerifier = (HostnameVerifierJSSE) Class.forName(argv[2]).newInstance();
      sslCtx.setHostnameVerifierJSSE(hVerifier);
      TrustManagerJSSE tManager = new NulledTrustManager();
      sslCtx.setTrustManagerJSSE(tManager);
      System.out.println(" Creating new SSLSocketFactory with SSLContext");
     SSLSocketFactory sslSF = (SSLSocketFactory) sslCtx.getSocketFactoryJSSE();
      System.out.println(" Creating and opening new SSLSocket with
                              SSLSocketFactory");

      // using createSocket(String hostname, int port)
      SSLSocket sslSock = (SSLSocket) sslSF.createSocket(argv[0],
                                              new Integer(argv[1]).intValue());
      System.out.println(" SSLSocket created");
      sslSock.addHandshakeCompletedListener(new MyListener());
      OutputStream out = sslSock.getOutputStream();

      // Send a simple HTTP request
      String req = "GET /examplesWebApp/ShowDate.jsp HTTP/1.0\r\n\r\n";
      out.write(req.getBytes());

      // Retrieve the InputStream and read the HTTP result, displaying
      // it on the console
      InputStream    in = sslSock.getInputStream();
      byte buf[] = new byte[1024];
      try
      {
        while (true)
        {
          int amt = in.read(buf);
          if (amt == -1) break;
          System.out.write(buf, 0, amt);
        }
      }
      catch (IOException e)
      {
        return;
      }
      sslSock.close();
      System.out.println(" SSLSocket closed");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

SSLClientServlet Sample

The SSLClientServlet sample is a simple servlet wrapper of the SSLClient sample.

Listing 4-4 shows a sample SSLClientServlet. This code taken from the `SSLSClientServlet.java` file located at `SAMPLES_HOME\server\examples\src\examples\security\sslclient`.

Listing 4-4 SSLClientServlet Sample Code

```
package examples.security.sslclient;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.Enumeration;
import javax.servlet.*;
import javax.servlet.http.*;

/**
 * SSLClientServlet is a simple servlet wrapper of
 * examples.security.sslclient.SSLClient.
 *
 * @see SSLClient
 * @author Copyright (c) 1999-2002 by BEA Systems, Inc. All Rights Reserved.
 */

public class SSLClientServlet extends HttpServlet {
  public void service(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    response.setHeader("Pragma", "no-cache"); // HTTP 1.0
    response.setHeader("Cache-Control", "no-cache"); // HTTP 1.1
    ServletOutputStream out = response.getOutputStream();

    out.println("<br><h2>ssl client test</h2><br><hr>");
    String[] target = request.getParameterValues("url");
    try {

      out.println("<h3>wls ssl client classes</h3><br>");
      out.println("java SSLClient wls localhost 7001 7002
                       /examplesWebApp/SnoopServlet.jsp<br>");
      out.println("<pre>");
      SSLClient.wlsURLConnect("localhost", "7001", "7002",
                                  "/examplesWebApp/SnoopServlet.jsp", out);
      out.println("</pre><br><hr><br>");

    } catch (IOException ioe) {
      out.println("<br><pre> "+ioe.getMessage()+"</pre>");
      ioe.printStackTrace();
    }
  }
}
```

Using Two-Way SSL Authentication

When using certificate authentication, WebLogic Server sends a digital certificate to the requesting client. The client examines the digital certificate to ensure that it is authentic, has not expired, and matches the WebLogic Server instance that presented it.

With two-way SSL authentication (a form of mutual authentication), the requesting client also presents a digital certificate to WebLogic Server. When the instance of WebLogic Server is configured for two-way SSL authentication, requesting clients are required to present digital certificates from a specified set of certificate authorities. WebLogic Server accepts only digital certificates that are signed by root certificates from the specified trusted certificate authorities.

For information on how to configure WebLogic Server for two-way SSL authentication, see the Configuring SSL in *Managing WebLogic Security*.

The following sections describe the different ways two-way SSL authentication can be implemented in WebLogic Server.

- “Two-Way SSL Authentication with JNDI” on page 4-20
- “Using Two-Way SSL Authentication Between WebLogic Server Instances” on page 4-24

- "Using Two-Way SSL Authentication with Servlets" on page 4-25

Two-Way SSL Authentication with JNDI

When using JNDI for two-way SSL authentication in a Java client, use the `setSSLClientCertificate()` method of the WebLogic JNDI `Environment` class. This method sets a private key and chain of X.509 digital certificates for client authentication.

To pass digital certificates to JNDI, create an array of `InputStreams` opened on files containing DER-encoded digital certificates and set the array in the JNDI hash table. The first element in the array must contain an `InputStream` opened on the Java client's private key file. The second element must contain an `InputStream` opened on the Java client's digital certificate file. (This file contains the public key for the Java client.) Additional elements may contain the digital certificates of the root certificate authority and the signer of any digital certificates in a certificate chain. A certificate chain allows WebLogic Server to authenticate the digital certificate of the Java client if that digital certificate was not directly issued by a certificate authority registered for the Java client in the WebLogic Server keystore file.

You can use the `weblogic.security.PEMInputStream` class to read digital certificates stored in Privacy Enhanced Mail (PEM) files. This class provides a filter that decodes the base 64-encoded DER certificate into a PEM file.

Listing 4-5 demonstrates how to use two-way SSL authentication in a Java client.

Listing 4-5 Example of a Two-Way SSL Authentication Client That Uses JNDI

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import weblogic.jndi.Environment;
import weblogic.security.PEMInputStream;
import java.io.InputStream;
import java.io.FileInputStream;

public class SSLJNDIClient
{
  public static void main(String[] args) throws Exception
  {
    Context context = null;
    try {
      Environment env = new Environment();

      // set connection parameters
      env.setProviderUrl("t3s://localhost:7002");
      // The next two set methodes are optional if you are using
      // a UserNameMapper interface.
      env.setSecurityPrincipal("system");
      env.setSecurityCredentials("weblogic");

      InputStream key = new FileInputStream("certs/demokey.pem");
      InputStream cert = new FileInputStream("certs/democert.pem");

      // wrap input streams if key/cert are in pem files
      key = new PEMInputStream(key);
      cert = new PEMInputStream(cert);

      env.setSSLClientCertificate(new InputStream[] { key, cert});

      env.setInitialContextFactory(Environment.
                  DEFAULT_INITIAL_CONTEXT_FACTORY);
      context = env.getInitialContext();

      Object myEJB = (Object) context.lookup("myEJB");
    }
    finally {
      if (context != null) context.close();
    }
  }
}
```

When the JNDI `getInitialContext()` method is called, the Java client and WebLogic Server execute mutual authentication in the same way that a Web browser performs mutual authentication to get a secure Web server connection. An exception is thrown if the digital certificates cannot be validated or if the Java client's digital certificate cannot be authenticated in the default (active) security realm. The authenticated user object is stored on the Java client's server thread and is used for checking the permissions governing the Java client's access to any protected WebLogic resources.

When you use the WebLogic JNDI `Environment` class, you must create a new `Environment` object for each call to the `getInitialContext()` method. Once you specify a `User` object and security credentials, both the user and their associated credentials remain set in the `Environment` object. If you try to reset them and then call the JNDI `getInitialContext()` method, the original user and credentials are used.

When you use two-way SSL authentication from a Java client, WebLogic Server gets a unique Java Virtual Machine (JVM) ID for each client JVM so that the connection between the Java client and WebLogic Server is constant. Unless the connection times out from lack of activity, it persists as long as the JVM for the Java client continues to execute. The only way a Java client can negotiate a new SSL connection reliably is by stopping its JVM and running another instance of the JVM.

The code in Listing 4-5, “Example of a Two-Way SSL Authentication Client That Uses JNDI,” on page 4-20 generates a call to the WebLogic Identity Assertion provider that implements the `weblogic.security.providers.authentication.UserNameMapper` interface. The class that implements the `UserNameMapper` interface returns a user object if the digital certificate is valid. WebLogic Server stores this authenticated user object on the Java client’s thread in WebLogic Server and uses it for subsequent authorization requests when the thread attempts to use WebLogic resources protected by the default (active) security realm.

Note: The implementation of the `weblogic.security.providers.authentication.UserNameMapper` interface must be specified in your `CLASSPATH`.

If you have not configured an Identity Assertion provider that performs certificate-based authentication, a Java client running in a JVM with an SSL connection can change the WebLogic Server user identity by creating a new JNDI `InitialContext` and supplying a new user name and password in the JNDI `SECURITY_PRINCIPAL` and `SECURITY_CREDENTIALS` properties. Any digital certificates passed by the Java client after the SSL connection is made are not used. The new WebLogic Server user continues to use the SSL connection negotiated with the initial user’s digital certificate.

If you have configured an Identity Assertion provider that performs certificate-based authentication, WebLogic Server passes the digital certificate from the Java client to the class that implements the `UserNameMapper` interface and the `UserNameMapper` class maps the digital certificate to a WebLogic Server user name. Therefore, if you want to set a new user identity when you use the certificate-based identity assertion, you cannot change the identity. This is because the digital certificate is processed only at the time of the first connection request from the JVM for each `Environment`.

Caution: **Restriction:** Multiple, concurrent, user logins to WebLogic Server from a single client JVM when using two-way SSL and JNDI is not supported. If multiple logins are executed on different threads, the results are undeterminable and might result in one user’s requests being executed on another user’s login, thereby allowing one user to access another user’s data. WebLogic Server does not support multiple, concurrent, certificate-based logins from a single client JVM. For information on JNDI contexts and threads and how to avoid potential JNDI context problems, see “JNDI Contexts and Threads” and “How to Avoid JNDI Context Problems” in *Programming WebLogic JNDI*.

Writing a User Name Mapper

When using 2-way SSL, WebLogic Server verifies the digital certificate of the Web browser or Java client when establishing an SSL connection. However, the digital certificate does not identify the Web browser or Java client as a user in the WebLogic Server security realm. If the Web browser or Java client requests a WebLogic Server resource protected by a security policy, WebLogic Server requires the Web browser or Java client to have an identity. To handle this requirement, the WebLogic Identity Assertion provider allows you to enable a user name mapper that maps the digital certificate of a Web browser or Java client to a user in a WebLogic Server security realm. The user name mapper must be an implementation the `weblogic.security.providers.authentication.UserNameMapper` interface.

In this release of WebLogic Server, you have the option of the using the default implementation of the `weblogic.security.providers.authentication.UserNameMapper` interface, `DefaultUserNameMapperImpl`, or developing your own implementation.

The WebLogic Identity Assertion provider can call the implementation of the `UserNameMapper` interface for the following types of identity assertion token types:

- X.509 digital certificates passed via the SSL handshake
- X.509 digital certificates passed via CSIv2
- X.501 distinguished names passed via CSIv2

If you need to map different types of certificates, you need write your own implementation of the `UserNameMapper` interface.

To implement a `UserNameMapper` interface that maps a digital certificate to a user name, write a `UserNameMapper` class that performs the following operations:

1. Instantiates the `UserNameMapper` implementation class.
2. Creates the `UserNameMapper` interface implementation.
3. Uses the `mapCertificateToUserName()` method to map a certificate to a user name based on a certificate chain presented by the client.
4. Maps a string attribute type to the corresponding `Attribute Value Assertion` field type.

Using Two-Way SSL Authentication Between WebLogic Server Instances

You can use two-way SSL authentication in server-to-server communication in which one WebLogic Server instance is acting as the client of another WebLogic Server instance. Using two-way SSL authentication in server-to-server communication enables you to have dependable, highly-secure connections, even without the more common client/server environment.

Listing 4-6 shows an example of how to establish a secure connection from a servlet running in one instance of WebLogic Server to a second WebLogic Server instance called `server2.weblogic.com`.

Listing 4-6 Establishing a Secure Connection to Another WebLogic Server Instance

```
FileInputStream [] f = new FileInputStream[3];
   f[0]= new FileInputStream(“demokey.pem”);
   f[1]= new FileInputStream(“democert.pem”);
   f[2]= new FileInputStream(“ca.pem”);
Environment e = new Environment ();
e.setProviderURL(“t3s://server2.weblogic.com:443”);
e.setSSLClientCertificate(f);
e.setSSLServerName(“server2.weblogic.com”);
e.setSSLRootCAFingerprints(“ac45e2d1ce492252acc27ee5c345ef26”);


e.setInitialContextFactory
       (“weblogic.jndi.WLInitialContextFactory”);
Context ctx = new InitialContext(e.getProperties())
```

In Listing 4-6, the WebLogic JNDI `Environment` class creates a hash table to store the following parameters:

- `setProviderURL`—specifies the URL of the WebLogic Server instance acting as the SSL server. The WebLogic Server instance acting as SSL client calls this method. The URL specifies the t3s protocol which is a WebLogic Server proprietary protocol built on the SSL protocol. The SSL protocol protects the connection and communication between the two WebLogic Servers instances.

- `setSSLClientCertificate`—specifies a certificate chain to use for the SSL connection. You use this method to specify an input stream array that consists of a private key (which is the first input stream in the array) and a chain of X.509 certificates (which make up the remaining input streams in the array). Each certificate in the chain of certificates is the issuer of the certificate preceding it in the chain.

- `setSSLServerName`—specifies the name of the WebLogic Server instance acting as the SSL server. When the SSL server presents its digital certificate to the server acting as the SSL client, the name specified using the `setSSLServerName` method is compared to the common name field in the digital certificate. In order for hostname verification to succeed, the names must match. This parameter is used to prevent man-in-the-middle attacks.

- `setSSLRootCAFingerprint`—specifies digital codes that represent a set of trusted certificate authorities. The root certificate in the certificate chain received from the WebLogic Server instance acting as the SSL server has to match one of the fingerprints specified with this method to be trusted. This parameter is used to prevent man-in-the-middle attacks.

Note: For information on JNDI contexts and threads and how to avoid potential JNDI context problems, see "JNDI Contexts and Threads" and "How to Avoid JNDI Context Problems" in the *Programming WebLogic JNDI*.

Using Two-Way SSL Authentication with Servlets

To authenticate Java clients in a servlet (or any other server-side Java class), you must check whether the client presented a digital certificate and if so, whether the certificate was issued by a trusted certificate authority. The servlet developer is responsible for asking whether the Java client has a valid digital certificate. When developing servlets with the WebLogic Servlet API, you must access information about the SSL connection through the `getAttribute()` method of the `HTTPServletRequest` object.

The following attributes are supported in WebLogic Server servlets:

- `javax.servlet.request.X509Certificate`
  `java.security.cert.X509Certificate []`—returns an array of the X.509 certificate.

- `javax.servlet.request.cipher_suite`—returns a string representing the cipher suite used by HTTPS.

- `javax.servlet.request.key_size`— returns an integer (0, 40, 56, 128, 168) representing the bit size of the symmetric (bulk encryption) key algorithm.

- `weblogic.servlet.request.SSLSession`
  `javax.net.ssl.SSLSession`—returns the SSL session object that contains the cipher suite and the dates on which the object was created and last used.

You have access to the user information defined in the digital certificates. When you get the `javax.servlet.request.X509Certificate` attribute, it is an array of the `java.security.cert` X.509 certificate. You simply cast the array to that and examine the certificates.

A digital certificate includes information, such as the following:

- The name of the subject (holder, owner) and other identification information required to verify the unique identity of the subject, such as the uniform resource locator (URL) of the Web server using the digital certificate, or an individual user's e-mail address
- The subject's public key
- The name of the certificate authority that issued the digital certificate
- A serial number
- The validity period (or lifetime) of the digital certificate (as defined by a start date and an end date)

Using a Custom Host Name Verifier

A host name verifier validates that the host to which an SSL connection is made is the intended or authorized party. A host name verifier is useful when a WebLogic client or a WebLogic Server instance is acting as an SSL client to another application server. It helps prevent man-in-the-middle attacks.

Note: In this release of WebLogic Server, the demonstration digital certificates are generated during installation so they do contain the hostname of the system on which the WebLogic Server software installed. Therefore, you should leave host name verification on when using the demonstration certificates for development or testing purposes.

By default, WebLogic Server, as a function of the SSL handshake, compares the common name in the Subject's Distinguished Name (SubjectDN) of the SSL server's digital certificate with the host name of the SSL server used to initiate the SSL connection. If these names do not match, the SSL connection is dropped.

The dropping of the SSL connection is caused by the SSL client, which validates the host name of the server against the digital certificate of the server. If anything but the default behavior is desired, you can either turn off host name verification or register a custom host name verifier.

Turning off host name verification leaves the SSL connections vulnerable to man-in-the-middle attacks.

You can turn off host name verification in the following ways:

- In the Administration Console, specify None in the Hostname Verification field that is located on the Advanced Options pane under the Keystore & SSL tab for the server (for example, `myserver`).
- On the command line of the SSL client, enter the following argument:

  ```
  -Dweblogic.security.SSL.ignoreHostnameVerification=true
  ```

You can write a custom host name verifier. The `weblogic.security.SSL.HostnameVerifierJSSE` interface provides a callback mechanism so that you can define a policy for handling the case where the server name that is being connected to does not match the server name in the SubjectDN of the server's digital certificate.

To use a custom host name verifier, create a class that implements the `weblogic.security.SSL.HostnameVerifierJSSE` interface and define the methods that capture information about the server's security identity.

Note: This interface takes new style certificates and replaces the `weblogic.security.SSL.HostnameVerifier` interface, which is deprecated in this release of WebLogic Server.

Before you can use a custom host name verifier, you need to specify the class for your implementation in the following ways:

- In the Administration Console, specify the class for the custom host name verifier in the Client Attributes fields that are located on the Advanced Options pane under the Keystore & SSL tab for the server (for example, `myserver`).
- On the command line, enter the following argument:

  ```
  -Dweblogic.security.SSL.HostnameVerifierJSSE=hostnameverifier
  ```

  where *`hostnameverifier`* is the name of the class that implements the custom host name verifier.

An example of a custom host name verifier (see Listing 4-7) is available in the SSLclient code located in the *`SAMPLES_HOME`*`\server\examples\src\examples\security\sslclient` directory. This code example contains a `NulledHostnameVerifier` class which always returns true for the comparison. This sample allows the WebLogic SSL client to connect to any SSL server regardless of the server's host name and digital certificate SubjectDN comparison.

Listing 4-7 Host Name Verifier Sample Code

```
package examples.security.sslclient;
/**
 * HostnameVerifierJSSE provides a callback mechanism so that
 * implementers of this interface can supply a policy for handling
 * the case where the host that's being connected to and the server
 * name from the certificate SubjectDN must match.
 *
 * This is a null version of that class to show the WebLogic SSL
 * client classes without major problems. For example, in this case,
 * the client code connects to a server at 'localhost' but the
 * democertificate's SubjectDN CommonName is 'bea.com' and the
 * default WebLogic HostnameVerifierJSSE does a String.equals() on
 * those two hostnames.
 *
 * @see HostnameVerifier#verify
 * @author Copyright (c) 1999-2002 by BEA Systems, Inc. All Rights
 * Reserved.
 */
public class NulledHostnameVerifier implements
                     weblogic.security.SSL.HostnameVerifierJSSE {
  public boolean verify(String urlHostname, String certHostname) {
    return true;
  }
}
```

Using a Trust Manager

The `weblogic.security.SSL.TrustManagerJSSE` interface allows you to override validation errors in a peer's digital certificate and continue the SSL handshake. You can also use the interface to discontinue an SSL handshake by performing additional validation on a server's digital certificate chain.

Note: This interface takes new style certificates and replaces the `weblogic.security.SSL.TrustManager` interface, which is deprecated in this release of WebLogic Server.

When an SSL client connects to an instance of WebLogic Server, the server presents its digital certificate chain to the client for authentication. That chain could contain an invalid digital certificate. The SSL specification says that the client should drop the SSL connection upon discovery of an invalid certificate. Web browsers, however, ask the user whether to ignore the invalid certificate and continue up the chain to determine if it is possible to authenticate the SSL server with any of the remaining certificates in the certificate chain. You can use the `TrustManagerJSSE` interface to eliminate this inconsistent practice by controlling when to continue or discontinue an SSL connection. Using a trust manager, you can perform custom checks before continuing an SSL connection. For example, you can use a trust manager to specify that only users from specific localities, such as towns, states, or countries, or users with other special attributes, are allowed to gain access via the SSL connection.

Use the `weblogic.security.SSL.TrustManagerJSSE` interface to create a trust manager. The interface contains a set of error codes for certificate verification. You can also perform additional validation on the peer certificate and interrupt the SSL handshake if need be. After a digital certificate has been verified, the `weblogic.security.SSL.TrustManagerJSSE` interface uses a callback function to override the result of verifying the digital certificate. You can associate an instance of a trust manager with an SSL context through the `setTrustManagerJSSE()` method.

The `weblogic.security.SSL.TrustManagerJSSE` interface conforms to the JSSE specification. You can only set up a trust manger programmatically; its use cannot be defined through the Administration Console or on the command-line.

Note: Depending on the checks performed, use of a trust manager may potentially impact performance.

An example of a trust manager (see Listing 4-8) is available in the SSLClient code example located in the `SAMPLES_HOME\server\examples\src\examples\security\sslclient` directory.

Listing 4-8 TrustManager Code Example

```
package examples.security.sslclient;

import weblogic.security.SSL.TrustManagerJSSE;
import javax.security.cert.X509Certificate;

public class NulledTrustManagerJSSE implements TrustManagerJSSE{
  public boolean certificateCallback(X509Certificate[] o, int validateErr) {
   System.out.println(" --- Do Not Use In Production ---\n" + "  By using this " +
                    "NulledTrustManager, the trust in the server's identity "+
                    "is completely lost.\n --------------------------------");
    for (int i=0; i<o.length; i++)
      System.out.println(" certificate " + i + " -- " + o[i].toString());
    return true;
  }
}
```

The SSLSocketClient example uses the custom trust manager shown above. The SSLSocketClient shows how to set up a new SSL connection by using an SSL context with the trust manager. This example is also located in the `*SAMPLES_HOME*\server\examples\src\examples\security\sslclient` directory.

Using a Handshake Completed Listener

The `javax.net.ssl.HandshakeCompletedListener` interface defines how the SSL client receives notifications about the completion of an SSL protocol handshake on a given SSL connection. It also defines the number of times an SSL handshake takes place on a given SSL connection. Listing 4-9 shows a HandshakeCompletedListener interface code example. This example is also located in the `*SAMPLES_HOME*\server\examples\src\examples\security\sslclient` directory.

Listing 4-9 HandshakeCompletedListener Code Example

```
package examples.security.sslclient;

import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.FileInputStream;
import java.util.Hashtable;
import javax.net.ssl.HandshakeCompletedListener;
import javax.net.ssl.SSLSession;

  public class MyListener implements HandshakeCompletedListener
  {
    public void handshakeCompleted(javax.net.ssl.
                                 HandshakeCompletedEvent event)
    {
      SSLSession session = event.getSession();
      System.out.println("Handshake Completed with peer " +
                         session.getPeerHost());
      System.out.println("   cipher: " + session.getCipherSuite());
      javax.security.cert.X509Certificate[] certs = null;
      try
      {
        certs = session.getPeerCertificateChain();
      }
      catch (javax.net.ssl.SSLPeerUnverifiedException puv)
      {
        certs = null;
      }
      if  (certs != null)
      {
        System.out.println("   peer certificates:");
        for (int z=0; z<certs.length; z++) System.out.
                 println("certs["+z+"]: " + certs[z]);
      }
      else
      {
        System.out.println("No peer certificates presented");
      }
    }
  }
```

Using an SSLContext

The `SSLContext` class is used to programmatically configure SSL and retain SSL session information. For example, all sockets that are created by socket factories provided by the `SSLContext` class can agree on session state by using the handshake protocol associated with the SSL context. Each instance can be configured with the keys, certificate chains, and trusted root certificate authorities that it needs to perform authentication. These sessions are cached so that other sockets created under the same SSL context can potentially reuse them later. For more information on session caching see SSL Session Behavior in *Managing WebLogic Security*. To associate an instance of a trust manager class with its SSL context, use the `weblogic.security.SSL.SSLContext.setTrustManagerJSSE()` method.

You can only set up an SSL context programmatically; not by using the Administration Console or the command line. A Java `new` expression or the `getInstance()` method of the `SSLContext` class can create an `SSLContext` object. The `getInstance()` method is static and it generates a new `SSLContext` object that implements the specified secure socket protocol. An example of using the `SSLContext` class is provided in the SSLSocketClient sample in the `SAMPLES_HOME\server\examples\src\examples\security\sslclient` directory. This example (see Listing 4-3, “SSLSocketClient Sample Code,” on page 4-16) shows how to create a new SSL socket factory that will create a new SSL socket using `SSLContext`.

Listing 4-10 shows a sample instantiation using the `getInstance()` method.

Listing 4-10 SSL Context Code Example

```
import weblogic.security.SSL.SSLContext;

SSLcontext sslctx = SSLContext.getInstance (“https”)
```

Using an SSL Server Socket Factory

Instances of the `SSLServerSocketFactory` class create and return SSL sockets. This class extends `javax.net.SocketFactory`.

Listing 4-11 shows a sample instantiation of this class. Listing 4-3, “SSLSocketClient Sample Code,” on page 4-16 shows a code example that uses the `SSLServerSocketFactory` class. This code example is taken from the `SSLSocketClient.java` file located at `SAMPLES_HOME\server\examples\src\examples\security\sslclient`.

Listing 4-11 SSLServerSocketFactory Code Example

```
import weblogic.security.SSL.SSLSocketFactory;

  SSLSocketFactory sslSF = (SSLSocketFactory) sslCtx.getSocketFactoryJSSE();
```

Using URLs to Make Outbound SSL Connections

You can use a `URL` object to make an outbound SSL connection from a WebLogic Server instance acting as a client to another WebLogic Server instance. WebLogic Server supports both one-way and two-way SSL authentication for outbound SSL connections.

For one-way SSL authentication, you use the `java.net.URL`, `java.net.URLConnection`, and `java.net.HTTPURLConnection` classes to make outbound SSL connections using `URL` objects. Listing 4-12 shows a `simpleURL` class that supports both HTTP and HTTPS URLs and that only uses these Java classes (that is, no WebLogic classes are required). To use the `simpleURL` class for one-way SSL authentication (HTTPS) on WebLogic Server, all that is required is that "`weblogic.net`" be defined in the system property for `java.protocols.handler.pkgs`.

Note: Because the `simpleURL` sample shown in Listing 4-12 defaults trust and hostname checking, this sample requires that you connect to a real Web server that is trusted and that passes hostname checking by default. Otherwise, you must override trust and hostname checking on the command line.

Listing 4-12 One-Way SSL Authentication URL Outbound SSL Connection Class That Uses Java Classes Only

```
import java.net.URL;
import java.net.URLConnection;
import java.net.HttpURLConnection;
import java.io.IOException;

public class simpleURL
{
   public static void main (String [] argv)
   {
     if (argv.length != 1)
     {
       System.out.println("Please provide a URL to connect to");
       System.exit(-1);
     }
     setupHandler();
     connectToURL(argv[0]);
   }

   private static void setupHandler()
   {
     java.util.Properties p = System.getProperties();
     String s = p.getProperty("java.protocol.handler.pkgs");
     if (s == null)
       s = "weblogic.net";
     else if (s.indexOf("weblogic.net") == -1)
       s += "|weblogic.net";
     p.put("java.protocol.handler.pkgs", s);
     System.setProperties(p);
   }

   private static void connectToURL(String theURLSpec)
   {
     try
     {
       URL theURL = new URL(theURLSpec);
       URLConnection urlConnection = theURL.openConnection();
       HttpURLConnection connection = null;
       if (!(urlConnection instanceof HttpURLConnection))
       {
         System.out.println("The URL is not using HTTP/HTTPS: " +
                              theURLSpec);
         return;
       }
       connection = (HttpURLConnection) urlConnection;
       connection.connect();
       String responseStr = "\t\t" +
              connection.getResponseCode() + " -- " +
              connection.getResponseMessage() + "\n\t\t" +
                   connection.getContent().getClass().getName() + "\n";
       connection.disconnect();
       System.out.println(responseStr);
     }
     catch (IOException ioe)
     {
       System.out.println("Failure processing URL: " + theURLSpec);
       ioe.printStackTrace();
     }
    }
}
```

For two-way SSL authentication, the `weblogic.net.http.HttpsURLConnection` class provides a way to specify the security context information for a client, including the digital certificate and private key of the client. Instances of this class represent an HTTPS connection to a remote object.

The SSLClient code example demonstrates using the WebLogic `URL` object to make an outbound SSL connection (see Listing 4-13). The code example shown in Listing 4-13 is excerpted from the `SSLClient.java` file in the *`SAMPLES_HOME`*`\server\examples\src\examples\security\sslclient` directory.

Listing 4-13 WebLogic Two-Way SSL Authentication URL Outbound SSL Connection Code Example

```
wlsUrl = new URL("https", host, Integer.valueOf(sport).intValue(),
                 query);
weblogic.net.http.HttpsURLConnection sconnection =
         new weblogic.net.http.HttpsURLConnection(wlsUrl);

InputStream [] ins = new InputStream[2];
      ins[0] = new FileInputStream("client2certs.pem");
      ins[1] = new FileInputStream("clientkey.pem");
      String pwd = "clientkey";
      sconnection.loadLocalIdentity(ins[0], ins[1], pwd.toCharArray());
```

SSL Client Code Examples

A complete working SSL authentication sample is provided with the WebLogic Server product. The sample is located in the *`SAMPLES_HOME`*`\server\examples\src\examples\security\sslclient` directory. For a description of the sample and instructions on how to build, configure, and run this sample, see the `package.html` file in the sample directory. You can modify this code example and reuse it.

Securing Enterprise JavaBeans (EJBs)

WebLogic Server supports the J2EE architecture security model for securing Enterprise JavaBeans (EJBs), which includes support for declarative authorization (also referred to in this document as declarative security) and programmatic authorization (also referred to in this document as programmatic security).

The following topics are covered in this section:

Note: You can use deployment descriptor files and the Administration Console to secure EJBs. For information on using the Administration Console to secure EJBs, see *Securing WebLogic Resources*.

J2EE Architecture Security Model

The document *Designing Enterprise Applications with the J2EE Platform, Second Edition*, published by Sun Microsystems, Inc., states in Section 9.3 Authorization:

> "In the J2EE architecture, a container serves as an authorization boundary between the components it hosts and their callers. The authorization boundary exists inside the container's authentication boundary so that authorization is considered in the context of successful authentication. For inbound calls, the container compares security attributes

> from the caller's credential with the access control rules for the target component. If the rules are satisfied, the call is allowed. Otherwise, the call is rejected."
>
> "There are two fundamental approaches to defining access control rules: capabilities and permissions. Capabilities focus on what a caller can do. Permissions focus on who can do something. The J2EE application programming model focuses on permissions. In the J2EE architecture, the job of the deployer is to map the permission model of the application to the capabilities of users in the operational environment."

The same document then discusses two ways to control access to application resources using the J2EE architecture, declarative authorization and programmatic authorization.

The document *Designing Enterprise Applications with the J2EE Platform, Second Edition*, published by Sun Microsystems, Inc., in available online at `http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/security/security4.html`.

Declarative Authorization

The document *Designing Enterprise Applications with the J2EE Platform, Second Edition*, published by Sun Microsystems, Inc., states in Section 9.3.1 Authorization:

> "The deployer establishes the container-enforced access control rules associated with a J2EE application. The deployer uses a deployment tool to map an application permission model, which is typically supplied by the application assembler, to policy and mechanisms specific to the operational environment. The application permission model is defined in a deployment descriptor."

WebLogic Server supports the use of deployment descriptors to implement declarative authorization in EJBs.

Note: Declarative authorization is also referred in this document as declarative security.

The document *Designing Enterprise Applications with the J2EE Platform, Second Edition*, published by Sun Microsystems, Inc., in available online at `http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/security/security4.html`.

Programmatic Authorization

The document *Designing Enterprise Applications with the J2EE Platform, Second Edition*, published by Sun Microsystems, Inc., states in Section 9.3.2 Programmatic Authorization:

> "A J2EE container makes access control decisions before dispatching method calls to a component. The logic or state of the component doesn't factor in these access decisions. However, a component can use two methods, `EJBContext.isCallerInRole` (for use by enterprise bean code) and `HttpServletRequest.isUserInRole` (for use by Web components), to perform finer-grained access control. A component uses these methods to determine whether a caller has been granted a privilege selected by the component based on the parameters of the call, the internal state of the component, or other factors such as the time of the call."
>
> "The application component provider of a component that calls one of these functions must declare the complete set of distinct roleName values to be used in all calls. These declarations appear in the deployment descriptor as `security-role-ref` elements. Each `security-role-ref` element links a privilege name embedded in the application as a roleName to a security role. Ultimately, the deployer establishes the link between the privilege names embedded in the application and the security roles defined in the deployment descriptor. The link between privilege names and security roles may differ for components in the same application."
>
> "In addition to testing for specific privileges, an application component can compare the identity of its caller, acquired using `EJBContext.getCallerPrincipal` or `HttpServletRequest.getUserPrincipal`, to the distinguished caller identities embedded in the state of the component when it was created. If the identity of the caller is equivalent to a distinguished caller, the component can allow the caller to proceed. If not, the component can prevent the caller from further interaction. The caller principal returned by a container depends on the authentication mechanism used by the caller. Also, containers from different vendors may return different principals for the same user authenticating by the same mechanism. To account for variability in principal forms, an application developer who chooses to apply distinguished caller state in component access decisions should allow multiple distinguished caller identities, representing the same user, to be associated with components. This is recommended especially where application flexibility or portability is a priority."

WebLogic Server supports the use of the `EJBContext.isCallerInRole` and `EJBContext.getCallerPrincipal` methods and the use of the `security-role-ref` element in deployment descriptors to implement programmatic authorization in EJBs.

Note: Programmatic authorization is also referred in this document as programmatic security.

The document *Designing Enterprise Applications with the J2EE Platform, Second Edition*, published by Sun Microsystems, Inc., in available online at `http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/security/security4.html`.

Declarative Versus Programmatic Authorization

The document *Designing Enterprise Applications with the J2EE Platform, Second Edition*, published by Sun Microsystems, Inc., states in Section 9.3.3 Declarative Versus Programmatic Authorization:

> "There is a trade-off between the external access control policy configured by the deployer and the internal policy embedded in the application by the component provider. The external policy is more flexible after the application has been written. The internal policy provides more flexible functionality while the application is being written. In addition, the external policy is transparent and completely comprehensible to the deployer, while internal policy is buried in the application and may only be completely understood by the application developer. These trade-offs should be considered in choosing the authorization model for particular components and methods."

The document *Designing Enterprise Applications with the J2EE Platform, Second Edition*, published by Sun Microsystems, Inc., in available online at `http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/security/security4.html`.

Using Declarative Security With EJBs

To implement declarative security in EJBs you use deployment descriptors (`ejb-jar.xml` and `weblogic-ejb-jar.xml`) to define the security requirements. Listing 5-1 shows examples of how to use the `ejb-jar.xml` and `weblogic-ejb-jar.xml` deployment descriptors to map security role names to a security realm. The deployment descriptors map the application's logical security requirements to its runtime definitions. And at runtime, the EJB container uses the security definitions to enforce the requirements.

To configure security in the EJB deployment descriptors, perform the following steps (see Listing 5-1):

1. Use a text editor to create `ejb-jar.xml` and `weblogic-ejb-jar.xml` deployment descriptor files.
2. In the `ejb-jar.xml` file, define the security role name, the EJB name, and the method name (see bold text).

   Note: When specifying security role names, observe the following conventions and restrictions:

- The proper syntax for a security role name is as defined for an `Nmtoken` in the Extensible Markup Language (XML) recommendation available on the Web at: `http://www.w3.org/TR/REC-xml#NT-Nmtoken`.
- Do not use blank spaces, commas, hyphens, or any characters in this comma-separated list: \t, < >, #, |, &, ~, ?, ( ), { }.
- Security role names are case sensitive.
- The BEA suggested convention for security role names is that they be singular.

For more information on configuring security in the `ejb-jar.xml` file, see the *Sun Microsystems Enterprise JavaBeans Specification, Version 2.0* which is at this location on the Internet: `http://java.sun.com/products/ejb/docs.html`.

3. In the WebLogic-specific EJB deployment descriptor file, `weblogic-ejb-jar.xml`, define the security role name and link it to one or more principals (users or groups) in a security realm.

   For more information on configuring security in the `weblogic-ejb-jar.xml` file, see `weblogic-ejb-jar.xml` Deployment Descriptor Reference in *Programming WebLogic Enterprise JavaBeans*.

Listing 5-1 Using the ejb-jar.xml and weblogic-ejb-jar.xml Files to Map Security Role Names to a Security Realm

```
ejb-jar.xml entries:

         ...
<assembly-descriptor>
  <security-role>
        <role-name>manger</role-name>
  </security-role>
  <security-role>
        <role-name>east</role-name>
  </security-role>
  <method-permission>
        <role-name>manager</role-name>
        <role-name>east</role-name>
        <method>
          <ejb-name>accountsPayable</ejb-name>
          <method-name>getReceipts</method-name>
        </method>
  </method-permission>
         ...
</assembly-descriptor>
          ...
```

weblogic-ejb-jar.xml entries:

```
    <security-role-assignment>
        <role-name>manager</role-name>
        <principal-name>al</principal-name>
        <principal-name>george</principal-name>
        <principal-name>ralph</principal-name>
    </security-role-assignment>
       ...
```

EJB Security-Related Deployment Descriptors

The following topics describe the deployment descriptor elements that are used in the `ejb-jar.xml` and `weblogic-ejb-jar.xml` files to define security requirements in EJBs:

- "ejb-jar.xml Deployment Descriptors" on page 5-6
- "weblogic-ejb-jar.xml Deployment Descriptors" on page 5-13

ejb-jar.xml Deployment Descriptors

The following `ejb-jar.xml` deployment descriptor elements are used to define security requirements in WebLogic Server:

- "method" on page 5-7
- "method-permission" on page 5-8
- "role-name" on page 5-9
- "run-as" on page 5-9
- "security-identity" on page 5-9
- "security-role" on page 5-10

- "security-role-ref" on page 5-10
- "unchecked" on page 5-12
- "use-caller-identity" on page 5-12

The information in this section is based on the Document Type Descriptor (DTD) for `ejb-jar.xml` provided by Sun Microsystems, Inc. The DTD for `ejb-jar.xml` is available on the Web at `http://java.sun.com/dtd/ejb-jar_2_0.dtd`.

method

The `method` element is used to denote a method of an enterprise bean's home or component interface, or, in the case of a message-driven bean, the bean's onMessage method, or a set of methods.

The following table describes the elements you can define within an `method` element.

Element	Required/ Optional	Description
<description>	Optional	A text description of the method.
<ejb-name>	Required	Specifies the name of one of the enterprise beans declared in the `ejb-jar.xml` file.
<method-intf>	Optional	Allows you to distinguish between a method with the same signature that is multiply defined across both the home and component interfaces of the enterprise bean.
<method-name>	Required	Specifies a name of an enterprise bean method or the asterisk (*) character. The asterisk is used when the element denotes all the methods of an enterprise bean's component and home interfaces.
<method-params>	Optional	Contains a list of the fully-qualified Java type names of the method parameters.

Used Within

The `method` element is used within the `method-permission` element.

Example

For an example of how to use the `method` element, see Listing 5-1, "Using the ejb-jar.xml and weblogic-ejb-jar.xml Files to Map Security Role Names to a Security Realm," on page 5-5.

method-permission

The `method-permission` element specifies that one or more security roles are allowed to invoke one or more enterprise bean methods. The `method-permission` element consists of an optional description, a list of security role names or an indicator to state that the method is unchecked for authorization, and a list of method elements.

The security roles used in the method-permission element must be defined in the security-role elements of the deployment descriptor, and the methods must be methods defined in the enterprise bean's component and/or home interfaces.

The following table describes the elements you can define within an `method-permission` element.

Element	Required/ Optional	Description
<description>	Optional	A text description of this security constraint.
<role-name> or <unchecked>	Required	The `role-name` element or the `unchecked` element must be specified. The `role-name` element contains the name of a security role. The name must conform to the lexical rules for an `NMTOKEN`. The `unchecked` element specifies that a method is not checked for authorization by the container prior to invocation of the method.
<method>	Required	Specifies a method of an enterprise bean's home or component interface, or, in the case of a message-driven bean, the bean's `onMessage` method, or a set of methods.

Used Within

The `method-permission` element is used within the `assembly-descriptor` element.

Example

For an example of how to use the `method-permission` element, see Listing 5-1, "Using the ejb-jar.xml and weblogic-ejb-jar.xml Files to Map Security Role Names to a Security Realm," on page 5-5.

role-name

The role-name element contains the name of a security role. The name must conform to the lexical rules for an `NMTOKEN`.

Used Within

The role-name element is used within the `method-permission`, `run-as`, `security-role, and security-role-ref elements.`

Example

For an example of how to use the `role-name` element, see Listing 5-1, "Using the ejb-jar.xml and weblogic-ejb-jar.xml Files to Map Security Role Names to a Security Realm," on page 5-5.

run-as

The `run-as` element specifies the run-as identity to be used for the execution of the enterprise bean. It contains an optional description, and the name of a security role.

Used Within

The run-as element is used within the `security-identity` element.

Example

For an example of how to use the `run-as` element, see Listing 5-8, "run-as-role-assignment Element Example," on page 5-24.

security-identity

The `security-identity` element specifies whether the caller's security identity is to be used for the execution of the methods of the enterprise bean or whether a specific run-as identity is to be used. It contains an optional description and a specification of the security identity to be used.

The following table describes the elements you can define within an `security-identity` element.

Element	**Required/ Optional**	**Description**
<description>	Optional	A text description of the security identity.

Element	Required/ Optional	Description
<use-caller-identity> or <run-as>	Required	The `use-caller-identity` element or the `run-as` element must be specified. The `use-caller-identity` element specifies that the caller's security identity be used as the security identity for the execution of the enterprise bean's methods. The `run-as` element specifies the run-as identity to be used for the execution of the enterprise bean. It contains an optional description, and the name of a security role.

Used Within

The `security-identity` element is used within the `entity`, `message-driven`, and `session` elements.

Example

For an example of how to use the `security-identity` element, see Listing 5-3, "use-caller-identity Element Example," on page 5-12 and Listing 5-8, "run-as-role-assignment Element Example," on page 5-24.

security-role

The `security-role` element contains the definition of a security role. The definition consists of an optional description of the security role, and the security role name.

Used Within

The `security-role` element is used within the `assembly-descriptor` element.

Example

For an example of how to use the `assembly-descriptor` element, see Listing 5-1, "Using the ejb-jar.xml and weblogic-ejb-jar.xml Files to Map Security Role Names to a Security Realm," on page 5-5.

security-role-ref

The `security-role-ref` element contains the declaration of a security role reference in the enterprise bean's code. The declaration consists of an optional description, the security role name used in the code, and an optional link to a security role. If the security role is not specified, the Deployer must choose an appropriate security role.

The value of the `role-name` element must be the String used as the parameter to the `EJBContext.isCallerInRole(String roleName)` method or the `HttpServletRequest.isUserInRole(String role)` method.

Used Within

The `security-role-ref` element is used within the `entity` and `session` elements.

Example

For an example of how to use the security-role-ref element, see Listing 5-2.

Listing 5-2 Security-role-ref Element Example

```
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN' 'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>

<ejb-jar>
 <enterprise-beans>
   ...
   <session>
     <ejb-name>SecuritySLEJB</ejb-name>
     <home>weblogic.ejb20.security.SecuritySLHome</home>
     <remote>weblogic.ejb20.security.SecuritySL</remote>
      <ejb-class>weblogic.ejb20.security.SecuritySLBean</ejb-class>
     <session-type>Stateless</session-type>
     <transaction-type>Container</transaction-type>
     <security-role-ref>
        <role-name>rolenamedifffromlink</role-name>
<role-link>role121SL</role-link>
     </security-role-ref>
     <security-role-ref>
        <role-name>roleForRemotes</role-name>
        <role-link>roleForRemotes</role-link>
     </security-role-ref>
     <security-role-ref>
        <role-name>roleForLocalAndRemote</role-name>
        <role-link>roleForLocalAndRemote</role-link>
     </security-role-ref>
   </session>
    ...
 </enterprise-beans>
</ejb-jar>
```

unchecked

The `unchecked` element specifies that a method is not checked for authorization by the container prior to invocation of the method.

Used Within

The `unchecked` element is used within the `method-permission` element.

Example

For an example of how to use the `unchecked` element, see Listing 5-1, "Using the ejb-jar.xml and weblogic-ejb-jar.xml Files to Map Security Role Names to a Security Realm," on page 5-5.

use-caller-identity

The `use-caller-identity` element specifies that the caller's security identity be used as the security identity for the execution of the enterprise bean's methods.

Used Within

The `use-caller-identity` element is used within the `security-identity` element.

Example

For an example of how to use the `use-caller-identity` element, see Listing 5-3.

Listing 5-3 use-caller-identity Element Example

```
<ejb-jar>
  <enterprise-beans>
```

```
    <session>
      <ejb-name>SecurityEJB</ejb-name>
      <home>weblogic.ejb20.SecuritySLHome</home>
      <remote>weblogic.ejb20.SecuritySL</remote>
      <local-home>
          weblogic.ejb20.SecurityLocalSLHome
      </local-home>
      <local>weblogic.ejb20.SecurityLocalSL</local>
      <ejb-class>weblogic.ejb20.SecuritySLBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>

    <message-driven>
      <ejb-name>SecurityEJB</ejb-name>
      <ejb-class>weblogic.ejb20.SecuritySLBean</ejb-class>
      <transaction-type>Container</transaction-type>
      <security-identity>
         <use-caller-identity/>
      </security-identity>
    </message-driven>

  </enterprise-beans>
</ejb-jar>
```

weblogic-ejb-jar.xml Deployment Descriptors

The following `weblogic-ejb-jar.xml` deployment descriptor elements are used to define security requirements in WebLogic Server:

- "client-authentication" on page 5-14
- "client-cert-authentication" on page 5-15
- "confidentiality" on page 5-15
- "externally-defined" on page 5-16
- "identity-assertion" on page 5-18

- "iiop-security-descriptor" on page 5-19
- "integrity" on page 5-20
- "principal-name" on page 5-20
- "role-name" on page 5-21
- "run-as-identity-principal" on page 5-21
- "run-as-principal-name" on page 5-23
- "run-as-role-assignment" on page 5-23
- "security-permission" on page 5-26
- "security-permission-spec" on page 5-26
- "security-role-assignment" on page 5-27
- "transport-requirements" on page 5-27

client-authentication

The `client-authentication` element specifies whether the EJB supports or requires client authentication.

The following table defines the possible settings.

Setting	Definition
None	Client authentication is not supported.
Supported	Client authentication is supported, but not required.
Required	Client authentication is required.

Example

For an example of how to use the `client-authentication` element, see Listing 5-6, "iiop-security-descriptor Element Example," on page 5-19.

client-cert-authentication

The `client-cert-authentication` element specifies whether the EJB supports or requires client certificate authentication at the transport level.

The following table defines the possible settings.

Setting	Definition
None	Client certificate authentication is not supported.
Supported	Client certificate authentication is supported, but not required.
Required	Client certificate authentication is required.

Example

For an example of how to use the `client-cert-authentication` element, see Listing 5-10, "transport-requirements Element Example," on page 5-28.

confidentiality

The `confidentiality` element specifies the transport confidentiality requirements for the EJB. Using the `confidentiality` element ensures that the data is sent between the client and server in such a way as to prevent other entities from observing the contents.

The following table defines the possible settings.

Setting	**Definition**
None	Confidentiality is not supported.
`Supported`	Confidentiality is supported, but not required.
`Required`	Confidentiality is required.

Example

For an example of how to use the `confidentiality` element, see Listing 5-10, "transport-requirements Element Example," on page 5-28.

externally-defined

In WebLogic Server 8.1 and later, the `externally-defined` tag is supported for use in the `weblogic-ejb-jar.xml` deployment descriptors. You can use this tag, instead of the `<principal-name>` tag, to explicitly indicate that you want the security roles defined in the deployment descriptors by the `<role-name>` tag in the `weblogic-ejb-jar.xml` file to use the mappings that you specify in the Administration Console.

Note: The `externally-defined element` replaces the `global-role element` that was used in WebLogic Server 7.0 SP1. The `externally-defined` tag has the same functionality as the `global-role` element. The `global-role` element was deprecated in WebLogic Server 8.1.

The `externally-defined element` gives you the flexibility of not having to specify a specific role mapping for each security role defined in the deployment descriptors for a particular EJB. Rather, you can use the Administration Console to specify and modify a specific role mapping for each defined role at anytime. Additionally, because you may elect to use this tag on some EJBs and not others, it is not necessary to select the **ignore roles and polices from DD** option for the security realm. You select this option in the **On Future Redeploys:** field on the **General** tab of the **Security->Realms->myrealm** control panel on the Administration Console. Thus, within the same security realm, deployment descriptors can be used to specify and modify security for some EJBs, while the Administration Console can be used to specify and modify security for others.

Note: When specifying security role names, observe the following conventions and restrictions:

- The proper syntax for a security role name is as defined for an `Nmtoken` in the Extensible Markup Language (XML) recommendation available on the Web at: `http://www.w3.org/TR/REC-xml#NT-Nmtoken`.
- Do not use blank spaces, commas, hyphens, or any characters in this comma-separated list: \t, < >, #, |, &, ~, ?, ( ), { }.
- Security role names are case sensitive.
- The BEA suggested convention for security role names is that they be singular.

Listing 5-4 and Listing 5-5 show by comparison how to use the `externally-defined` element in the `weblogic-ejb-jar.xml` file. In Listing 5-5, the specification of the "manager" `externally-defined` element in the `weblogic-ejb-jar.xml` means that for security to be correctly configured on the `getReceipts` method, the principals for `manager` will have to be created in the Administration Console.

Listing 5-4 Using the ejb-jar.xml and weblogic-ejb-jar.xml Deployment Descriptors to Map Security Roles in EJBs

```
ejb-jar.xml entries:
         ...
<assembly-descriptor>
  <security-role>
        <role-name>manger</role-name>
  </security-role>
  <security-role>
        <role-name>east</role-name>
  </security-role>
  <method-permission>
        <role-name>manager</role-name>
        <role-name>east</role-name>
        <method>
          <ejb-name>accountsPayable</ejb-name>
          <method-name>getReceipts</method-name>
        </method>
  </method-permission>
         ...
</assembly-descriptor>
          ...
weblogic-ejb-jar.xml entries:

  <security-role-assignment>
      <role-name>manager</role-name>
      <principal-name>joe</principal-name>
      <principal-name>Bill</principal-name>
      <principal-name>Mary</principal-name>
      ...
</security-role-assignment>
   ...
```

Listing 5-5 Using the externally-defined Element in EJB Deployment Descriptors for Role Mapping

```
ejb-jar.xml entries:
         ...
<assembly-descriptor>
  <security-role>
        <role-name>manger</role-name>
  </security-role>
  <security-role>
        <role-name>east</role-name>
  </security-role>
  <method-permission>
        <role-name>manager</role-name>
        <role-name>east</role-name>
        <method>
          <ejb-name>accountsPayable</ejb-name>
          <method-name>getReceipts</method-name>
        </method>
  </method-permission>
         ...
</assembly-descriptor>
          ...
weblogic-ejb-jar.xml entries:
  <security-role-assignment>
      <role-name>manager</role-name>
      <externally-defined/>
      ...
  </security-role-assignment>
   ...
```

For more information on using the Administration Console to configure security for EJBs, see *Securing WebLogic Resources*.

identity-assertion

The `identity-assertion` element specifies whether the EJB supports identity assertion.

The following table defines the possible settings.

Setting	Definition
None	Identity assertion is not supported
Supported	Identity assertion is supported, but not required.
Required	Identity assertion is required.

Used Within

The `identity-assertion` element is used with the `iiop-security-descriptor` element.

Example

For an example of how to the `identity-assertion` element, see Listing 5-6, "iiop-security-descriptor Element Example," on page 5-19.

iiop-security-descriptor

The `iiop-security-descriptor` element specifies security configuration parameters at the bean-level. These parameters determine the IIOP security information contained in the interoperable object reference (IOR).

Example

For an example of how to use the `iiop-security-descriptor` element, see Listing 5-6.

Listing 5-6 iiop-security-descriptor Element Example

```
<weblogic-enterprise-bean>
  <iiop-security-descriptor>
       <transport-requirements>
              <confidentiality>supported</confidentiality>
              <integrity>supported</integrity>
              <client-cert-authorization>
                  supported
</client-cert-authentication>
       </transport-requirements>
```

```
            <client-authentication>supported<client-authentication>
            <identity-assertion>supported</identity-assertion>
      </iiop-security-descriptor>
</weblogic-enterprise-bean>
```

integrity

The `integrity` element specifies the transport integrity requirements for the EJB. Using the integrity element ensures that the data is sent between the client and server in such a way that it cannot be changed in transit.

The following table defines the possible settings.

Setting	Definition
None	Integrity is not supported.
Supported	Integrity is supported, but not required.
Required	Integrity is required.

Used Within

The `integrity` element is used within the `transport-requirements` element.

Example

For an example of how to use the `integrity` element, see Listing 5-10, "transport-requirements Element Example," on page 5-28.

principal-name

`The principal-name` element specifies the name of the principal in the WebLogic Server security realm that applies to role name specified in the security-role-assignment element. At least one `principal` is required in the `security-role-assignment` element. You may define more than one `principal-name` for each role name.

Used Within

The `principal-name` element is used within the `security-role-assignment` element.

Example

For an example of how to use the `principal-name` element, see Listing 5-1, "Using the ejb-jar.xml and weblogic-ejb-jar.xml Files to Map Security Role Names to a Security Realm," on page 5-5.

role-name

The `role-name` element identifies an application role name that the EJB provider placed in the companion `ejb-jar.xml` file. Subsequent principal-name elements in the stanza map WebLogic Server principals to the specified `role-name`.

Used Within

The `role-name` element is used within the `security-role-assignment` element.

Example

For an example of how to use the `role-name` element, see Listing 5-1, "Using the ejb-jar.xml and weblogic-ejb-jar.xml Files to Map Security Role Names to a Security Realm," on page 5-5.

run-as-identity-principal

The `run-as-identity-principal` element specifies which security principal name is to be used as the run-as principal for a bean that has specified a security-identity run-as role-name in its ejb-jar deployment descriptor. For an explanation of how of run-as role-names to are mapped to run-as-identity-principals (or run-as-principal-names, see "run-as-role-assignment" on page 5-23.

Note: **Deprecated:** The `run-as-identity-principal` element is deprecated in the WebLogic Server 8.1. Use the `run-as-principal-name` element instead.

Used Within

The `run-as-identity-principal` element is used within the `run-as-role-assignment` element.

Example

For an example of how to use the `run-as-identity-principal` element, see Listing 5-7.

Listing 5-7 run-as-identity-principal Element Example

```
ebj-jar.xml:

<ejb-jar>
 <enterprise-beans>
 <session>
   <ejb-name>Caller2EJB</ejb-name>
   <home>weblogic.ejb11.security.CallerBeanHome</home>
   <remote>weblogic.ejb11.security.CallerBeanRemote</remote>
   <ejb-class>weblogic.ejb11.security.CallerBean</ejb-class>
   <session-type>Stateful</session-type>
   <transaction-type>Container</transaction-type>
   <ejb-ref><ejb-ref-name>Callee2Bean</ejb-ref-name>
     <ejb-ref-type>Session</ejb-ref-type>
     <home>weblogic.ejb11.security.CalleeBeanHome</home>
     <remote>weblogic.ejb11.security.CalleeBeanRemote</remote>
   </ejb-ref>
   <security-role-ref>
     <role-name>users1</role-name>
     <role-link>users1</role-link>
   </security-role-ref>

   <security-identity>
    <run-as>
      <role-name>users2</role-name>
    </run-as>
   </security-identity>
  </session>
 </enterprise-beans>
</ejb-jar>

woblogic-ejb-jar.xml:

<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>Caller2EJB</ejb-name>
    <reference-descriptor>
     <ejb-reference-description>
       <ejb-ref-name>Callee2Bean</ejb-ref-name>
        <jndi-name>security.Callee2Bean</jndi-name>
     </ejb-reference-description>
    </reference-descriptor>
    <run-as-identity-principal>wsUser3</run-as-identity-principal>
  </weblogic-enterprise-bean>
  <security-role-assignment>
    <role-name>user</role-name>
    <principal-name>wsUser2</principal-name>
    <principal-name>wsUser3</principal-name>
    <principal-name>wsUser4</principal-name>
  </security-role-assignment>
</weblogic-ejb-jar>
```

run-as-principal-name

The `run-as-principal-name` element specifies which security principal name is to be used as the run-as principal for a bean that has specified a security-identity run-as role-name in its ejb-jar deployment descriptor. For an explanation of how the run-as role-names map to run-as-principal-names, see "run-as-role-assignment" on page 5-23.

Used Within

The `run-as-principal-name` element is used within the `run-as-role-assignment` element.

Example

For an example of how to use the `run-as-principal-name` element, see Listing 5-8, "run-as-role-assignment Element Example," on page 5-24.

run-as-role-assignment

The `run-as-role-assignment` element is used to map a given security-identity run-as role-name that is specified in the `ejb-jar.xml` file to a `run-as-principal-name` specified in the `weblogic-ejb-jar.xml` file. The value of the `run-as-principal-name` element for a given role-name is scoped to all beans in the `ejb-jar.xml` file that use the specified role-name as their security-identity. The value of the `run-as-principal-name` element specified in `weblogic-ejb-jar.xml` file can be overridden at the individual bean level by specifying a `run-as-principal-name` element under that bean's `weblogic-enterprise-bean` element.

Note: For a given bean, if there is no `run-as-principal-name` element specified in either a `run-as-role-assignment` element or in a bean specific `run-as-principal-name` element, then the EJB container will choose the first `principal-name` of a security user in the weblogic-enterprise-bean `security-role-assignment` element for the `role-name` and use that `principal-name` as the `run-as-principal-name`.

Example

For an example of how to use the run-as-role-assignment element, see Listing 5-8.

Listing 5-8 run-as-role-assignment Element Example

```
In the ejb-jar.xml file:

// Beans "A_EJB_with_runAs_role_X" and "B_EJB_with_runAs_role_X"
// specify a security-identity run-as role-name "runAs_role_X".
// Bean "C_EJB_with_runAs_role_Y" specifies a security-identity
// run-as role-name "runAs_role_Y".

<ejb-jar>
  <enterprise-beans>

    <session>
      <ejb-name>SecurityEJB</ejb-name>
      <home>weblogic.ejb20.SecuritySLHome</home>
      <remote>weblogic.ejb20.SecuritySL</remote>
      <local-home>
          weblogic.ejb20.SecurityLocalSLHome
      </local-home>
      <local>weblogic.ejb20.SecurityLocalSL</local>
<ejb-class>weblogic.ejb20.SecuritySLBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>

    <message-driven>
      <ejb-name>SecurityEJB</ejb-name>
      <ejb-class>weblogic.ejb20.SecuritySLBean</ejb-class>
      <transaction-type>Container</transaction-type>
      <security-identity>
           <run-as>
                 <role-name>runAs_role_X</role-name>
           </run-as>
      </security-identity>
      <security-identity>
            <run-as>
                 <role-name>runAs_role_Y</role-name>
            </run-as>
      </security-identity>
    </message-driven>
  </enterprise-beans>
</ejb-jar>
```

weblogic-ejb-jar file:

```
<weblogic-ejb-jar>
   <weblogic-enterprise-bean>
     <ejb-name>A_EJB_with_runAs_role_X</ejb-name>
   </weblogic-enterprise-bean>

   <weblogic-enterprise-bean>
     <ejb-name>B_EJB_with_runAs_role_X</ejb-name>
     <run-as-principal-name>Joe</run-as-principal-name>
   </weblogic-enterprise-bean>

   <weblogic-enterprise-bean>
     <ejb-name>C_EJB_with_runAs_role_Y</ejb-name>
   </weblogic-enterprise-bean>

   <security-role-assignment>
     <role-name>runAs_role_Y</role-name>
     <principal-name>Harry</principal-name>
     <principal-name>John</principal-name>
   </security-role-assignment>

   <run-as-role-assignment>
     <role-name>runAs_role_X</role-name>
     <run-as-principal-name>Fred</run-as-principal-name>
   </run-as-role-assignment>
</weblogic-ejb-jar>
```

Each of the three beans shown in Listing 5-8 will choose a different principal name to run as.

- A_EJB_with_runAs_role_X

  This bean's run-as role-name is "runAs_role_X". The jar scoped <run-as-role-assignment> mapping will be used to look up the name of the principal to use. The <run-as-role-assignment> mapping specifies that for <role-name> "runAs_role_X" we are to use <run-as-principal-name> "Fred". Therefore, 'Fred' is the principal name that will be used.

- B_EJB_with_runAs_role_X

  This bean's run-as role-name is also "runAs_role_X". This bean will not use the jar scoped <run-as-role-assignment> to look up the name of the principal to use because that value is overridden by this bean's <weblogic-enterprise-bean> <run-as-principal-name> value "Joe". Therefore "Joe" is the principal name that will be used.

- C_EJB_with_runAs_role_Y

  This bean's run-as role-name is "runAs_role_Y". There is no explicit mapping of "runAs_role_Y to a run-as principal name, that is, there is no jar scoped <run-as-role-assignment> for "runAs_role_Y" nor is there a bean scoped <run-as-principal-name> specified in this bean's <weblogic-enterprise-bean>. To determine the principal name to use, the <security-role-assignment> for <role-name> "runAs_role_Y" is examined. The first <principal-name> corresponding to a user that is not a Group is chosen. Therefore, "Harry" is the principal name that will be used.

security-permission

The `security-permission` element specifies a security permission that is associated with a J2EE Sandbox.

Example

For an example of how to use the `security-permission` element, see Listing 5-9, "security-permission-spec Element Example," on page 5-27.

security-permission-spec

The `security-permission-spec` element specifies a single security permission based on the Security policy file syntax.

For more information, see Sun's implementation of the security permission specification:

http://java.sun.com/j2se/1.3/docs/guide/security/PolicyFiles.html#FileSyntax

Note: Disregard the optional `codebase` and `signedBy` clauses.

Used Within

The `security-permission-spec` element is used within the `security-permission` element.

Example

For an example of how to use the `security-permission-spec` element, see Listing 5-9.

Listing 5-9 security-permission-spec Element Example

```
<weblogic-ejb-jar>
  <security-permission>
     <description>Optional explanation goes here</description>
     <security-permission-spec>
<!
A single grant statement following the syntax of
http://java.sun.com/j2se/1.3/docs/guide/security/PolicyFiles.html#FileSynt
ax, without the codebase and signedBy clauses, goes here. For example:
-->
      grant {
      permission java.net.SocketPermission *, resolve;
      };
     </security-permission-spec>
   </security-permission>
</weblogic-ejb-jar>
```

In Listing 5-9, `permission java.net.SocketPermission` is the permission class name, `"*"` represents the target name, and `resolve` (resolve host/IP name service lookups) indicates the action.

security-role-assignment

The `security-role-assignment` element maps application roles in the `ejb-jar.xml` file to the names of security principals available in WebLogic Server.

Example

For an example of how to use the `security-role-assignment` element, see Listing 5-1, "Using the ejb-jar.xml and weblogic-ejb-jar.xml Files to Map Security Role Names to a Security Realm," on page 5-5.

transport-requirements

The `transport-requirements` element defines the transport requirements for the EJB.

Used Within

The `transport-requirements` element is used within the `iiop-security-descriptor` element.

Example

For an example of how to use the `transport-requirements` element, see Listing 5-10.

Listing 5-10 transport-requirements Element Example

```
<weblogic-enterprise-bean>
  <iiop-security-descriptor>
       <transport-requirements>
              <confidentiality>supported</confidentiality>
              <integrity>supported</integrity>
              <client-cert-authorization>
                   supported
              </client-cert-authentication>
</transport-requirements>
  </iiop-security-descriptor>
<weblogic-enterprise-bean>
```

Using Programmatic Security With EJBs

To implement programmatic security in EJBs you use the `javax.ejb.EJBContext.getCallerPrincipal()` and the `javax.ejb.EJBContext.isCallerInRole()` methods.

getCallerPrincipal

You use the `getCallerPrincipal()` method to determine the caller of the EJB. The `javax.ejb.EJBContext.getCallerPrincipal()` method returns a `WLSUser Principal` if one exists in the `Subject` of the calling user. In the case of multiple `WLSUser Principals`, the method returns the first in the ordering defined by the `Subject.getPrincipals().iterator()` method. If there are no `WLSUser Principals`, then the `getCallerPrincipal()` method returns the first non-`WLSGroup Principal`. If there are no `Principals` or all `Principals` are of type `WLSGroup`, this method returns `weblogic.security.WLSPrincipals.getAnonymousUserPrincipal()`. This behavior is similar to the semantics of `weblogic.security.SubjectUtils.getUserPrincipal()` except that `SubjectUtils.getUserPrincipal()` returns a `null` whereas `EJBContext.getCallerPrincipal()` returns `WLSPrincipals.getAnonmyousUserPrincipal()`.

For more information about how to use the `getCallerPrincipal()` method, see http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Security7.html#79857.

isCallerInRole

The `isCallerInRole()` method is used to determine if the caller (the current user) has been assigned a security role that is authorized to perform actions on the WebLogic resources in that thread of execution. For example, the method `javax.ejb.EJBContext.isCallerInRole("admin")` will return `true` if the current user has `admin` privileges.

For more information about how to use the `isCallerInRole()` method, see http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Security7.html#79857.

For Javadoc for the `isCallerInRole()` method, see http://java.sun.com/products/ejb/javadoc-1.1/javax/ejb/EJBContext.html#isCallerInRole(java.lang.String).

Using Network Connection Filters

This section covers the following topics:

The Benefits of Using Network Connection Filters

Security roles and security policies allow you to secure WebLogic resources at the domain level, the application level, and the application-component level, however, connection filters allow you to deny access at the network level. Thus, network connection filters are used to add an additional layer of security. Connection filters can be used to protect server resources on individual servers, server clusters, or an entire internal network, or Intranet.

Connection filters are particularly useful when using the Administration port. Depending on your network firewall configuration, you may be able to use a connection filter to further restrict administration access. A typical use might be to restrict access to the Administration port to only the servers and machines in the domain. Even if an attacker gets access to a machine inside the firewall, they will not be able to perform administration operations unless they are on one of the permitted machines.

Network connection filters are a type of firewall in that they can be configured to filter on protocols, IP addresses, and DNS node names. For example, you can deny any non-SSL connections originating outside of your corporate network. This would ensure that all access from systems on the Internet would be secure.

Network Connection Filter API

This section describes the `weblogic.security.net` package. This API provides interfaces and classes for developing network connection filters. It also includes a class, `ConnectionFilterImpl`, which is a ready-to-use implementation of a network connection filter. For more information, see Javadocs for WebLogic Classes for this release of WebLogic Server.

This section covers the following topics:

- "Connection Filter Interfaces" on page 6-2
- "Connection Filter Classes" on page 6-3

Connection Filter Interfaces

To implement connection filtering, write a class that implements the connection filter interfaces. The following `weblogic.security.net` interfaces are provided for implementing connection filters:

- "ConnectionFilter Interface" on page 6-2
- "ConnectionFilterRulesListener Interface" on page 6-3

ConnectionFilter Interface

This interface defines the `accept()` method, which is used to implement connection filtering. To program the server to perform connection filtering, instantiate a class that implements this interface and then configure that class in the Administration Console. This interface is the minimum implementation requirement for connection filtering.

Note: Implementing this interface alone does not permit the use of the Administration Console to enter and modify filtering rules to restrict client connections, rather, some other form (such as a flat file, which is defined in the Administration Console) must be used for that purpose. To use the Administration Console to enter and modify filtering rules, you must also implement the `ConnectionFilterRulesListener` interface. For a description of the `ConnectionFilterRulesListener` interface, see "ConnectionFilterRulesListener Interface" on page 6-3.

For code examples that demonstrate of how to use this interface, see "Connection Filter Examples" on page 6-8.

ConnectionFilterRulesListener Interface

The server uses this interface to determine if the rules specified in the Administration Console in the `ConnectionFilterRules` field are valid during startup and at runtime based on the `setRules()` and `checkRules()` methods.

Note: You can optionally implement this interface or just use the WebLogic connection filter implementation, `weblogic.security.net.ConnectionFilterImpl`, which is provided as part of the WebLogic Server product.

This interface defines two methods that are used to implement connection filtering: `setRules()` and `checkRules()`. Implementing this interface in addition to the `ConnectionFilter` interface allows the use of the Administration Console to enter filtering rules to restrict client connections.

Note: To be able to enter and edit connection filtering rules on the Administration Console, you must implement the `ConnectionFilterRulesListener` interface; otherwise some other means must be used. For example, you could use a flat file.

For a code example of how to use this interface, see "Connection Filter Examples" on page 6-8.

Connection Filter Classes

Two `weblogic.security.net` classes are provided for implementing connection filters:

- "ConnectionFilterImpl Class" on page 6-3
- "ConnectionEvent Class" on page 6-4

ConnectionFilterImpl Class

This class is the WebLogic connection filter implementation. It implements the `ConnectionFilter` and `ConnectionFilterRulesListener` interfaces. Once configured using the Administration Console, this connection filter accepts all incoming connections by default, and also provides static factory methods that allow the server to obtain the current connection filter. To use this connection to deny access, simply enter connection filter rules using the Administration Console.

This class is provided as part of the WebLogic Server product. To configure this class for use, see "Configuring the WebLogic Connection Filter" on page 6-7.

ConnectionEvent Class

This is the class from which all event state objects are derived. All events are constructed with a reference to the object, that is, the source that is logically deemed to be the object upon which a specific event initially occurred. Applications use the methods provided by this class (`getLocalAddress()`, `getLocalPort()`, `getRemoteAddress`(), `getRemotePort()`, and `hashcode()`) to create a new `ConnectionEvent` instance.

For a code example of how to use this class, see Listing 6-1, "Example of Filtering Network Connections," on page 6-9.

Guidelines for Writing Connection Filter Rules

This section describes how connection filter rules are written and evaluated. If no connection rules are specified, all connections are accepted.

Connection filter rules can be written in a flat file or input directly on the Administration Console, depending on how you implement connection filtering. The Network Connection Filter code examples demonstrate both methods. The Network Connection Filter code examples are `http://dev2dev.bea.com/code/wls.jsp` available at on the BEA dev2dev Web site.

The following sections provide information and guidelines for writing connection filter rules:

- "Connection Filter Rules Syntax" on page 6-4
- "Types of Connection Filter Rules" on page 6-5
- "How Connection Filter Rules are Evaluated" on page 6-6

Connection Filter Rules Syntax

The syntax of connection filter rules is as follows:

- Each rule must be written on a single line.
- Tokens in a rule are separated by white space.
- A pound sign (`#`) is the comment character. Everything after a pound sign on a line is ignored.
- Whitespace before or after a rule is ignored.

- Lines consisting only of whitespace or comments are skipped.

The format of filter rules differ depending on whether you are using a filter file to enter the filter rules or you enter the filter rules on the Administration Console.

- When entering the filter rules on the Administration Console, enter them in the following format:

  ```
  targetAddress localAddress localPort action protocols
  ```

- When specifying rules in the filter file, enter them in the following format:

  ```
  targetAddress action protocols
  ```

where

- `target` specifies one or more systems to filter.
- `localAddress` defines the host address of the WebLogic Server instance. (If you specify an asterisk (`*`), the match returns all local IP addresses.)
- `localPort` defines the port on which the WebLogic Server instance is listening. (If you specify an asterisk, the match returns all available ports on the server).
- `action` specifies the action to perform. This value must be `allow` or `deny`.
- `protocols` is the list of protocol names to match. The following protocols may be specified: `http, https`, `t3`, `t3s`, `giop`, `giops`, `dcom`, `ftp`, `ldap`. If no protocol is defined, all protocols will match a rule.

Types of Connection Filter Rules

Two types of filter rules are recognized:

- Fast rules

  A fast rule applies to a hostname or IP address with an optional netmask. If a hostname corresponds to multiple IP addresses, multiple rules are generated (in no particular order). Netmasks can be specified either in numeric or dotted-quad form. For example:

```
dialup-555-1212.pa.example.net  127.0.0.1 7001 deny t3 t3s  # http(s) OK

192.168.81.0/255.255.254.0   127.0.0.1 8001 allow   # 23-bit netmask

192.168.0.0/16   127.0.0.1 8002 deny   # like /255.255.0.0
```

  Hostnames for fast rules are looked up once at startup of the WebLogic Server instance. While this design greatly reduces overhead at connect time, it can result in the filter obtaining out of date information about what addresses correspond to a host name. BEA recommends using numeric IP addresses instead.

- Slow rules

  A slow rule applies to part of a domain name. Since a rule requires a connect-time DNS lookup on the client-side in order to perform a match, a slow rule may be much slower than the fast rule. Slow rules are also subject to DNS spoofing. Slow rules are specified as follows:

  ```
  *.script-kiddiez.org 127.0.0.1 7001 deny
  ```

  An asterisk only matches at the head of a pattern. If you specify an asterisk anywhere else in a rule, it is treated as part of the pattern. Note that the pattern will never match a domain name since an asterisk is not a legal part of a domain name.

How Connection Filter Rules are Evaluated

When a client connects to WebLogic Server, the rules are evaluated in the order in which they were written. The first rule to match determines how the connection is treated. If no rules match, the connection is permitted.

If you want to further protect your server and only allow connections from certain addresses, specify the last rule as:

```
0.0.0.0/0   *   *   deny
```

With this as the last rule, only connections that are allowed by preceding rules are allowed, all others are denied. For example, if you specify the following rules:

```
Remote IP Address * * allow https
0.0.0.0/0   *   *   deny
```

Only machines with the Remote IP Address are allowed to access the instance of WebLogic Server running connection filter. All other systems are denied access.

Note: The default connection filter implementation interprets a target address of 0 (0.0.0.0/0) as meaning "the rule should apply to all IP addresses." By design, the default filter does not evaluate the port, just the action. To clearly specify restrictions when using the default filter, modify the rules.

Another option is to implement a custom connection filter. A sample filter is available at http://dev.bea.com/code/security.jsp; the `FastFilterEntry.java` example contains `match()` methods, which determine how to process the rules.

Configuring the WebLogic Connection Filter

WebLogic Server provides a network connection filter as part of the product. It is ready to use out of the box. To use it, simply configure it using the Administration Console. For information on how to configure connection filters, see *Managing WebLogic Security*.

Developing Custom Connection Filters

If you decide not to use the WebLogic connection filter and want to develop you own, your can use the application programming interface (API) provided in the `weblogic.security.net` package to do so. For a description of this API, see "Network Connection Filter API" on page 6-2.

To develop custom connection filters with WebLogic Server, perform the following steps:

1. Write a class that implements the `ConnectionFilter` interface (minimum requirement).

   Or, optionally, if you want to use the Administration Console to enter and modify the connection filtering rules directly, write a class that implements both the `ConnectionFilter` interface and the `ConnectionFilterRulesListener` interface.

2. If you choose the minimum requirement in step 1 (only implementing the `ConnectionFilter` interface), enter the connection filtering rules in a flat file and define the location of the flat file in the class that implements the `ConnectionFilter` interface. Then use the Administration Console to configure the class in WebLogic Server. For instructions for configuring the class in the Administration Console, see the "Configuring Connection Filtering" section in *Managing WebLogic Security*. For an example of how to use a flat file to implement connection filtering, see the `SimpleConnectionFilter.java` file in the Network Connection Filter code examples, which are available at `http://dev2dev.bea.com/code/wls.jsp` on the BEA dev2dev Web site.

3. If you choose to implement both interfaces in step 1, use the Administration Console to configure the class and to enter the connection filtering rules. For instructions on configuring the class in the Administration Console, see "Configuring Connection Filtering" in *Managing WebLogic Security*. For an example of how to use the `ConnectionFilterRulesListener` interface to implement connection filtering, see the `SimpleConnectionFilter2.java` file in the Network Connection Filter code examples, which are available at `http://dev2dev.bea.com/code/wls.jsp` on the BEA dev2dev Web site.

Notes: If connection filtering is implemented when a Java or Web browser client tries to connect to a WebLogic Server instance, The WebLogic Server instance constructs a `ConnectionEvent` object and passes it to the `accept()` method of your connection filter class. The connection filter class examines the `ConnectionEvent` object and accepts the connection by returning, or denies the connection by throwing a `FilterException`.

Both implemented classes (the class that implements only the `ConnectionFilter` interface and the class that implements both the `ConnectionFilter` interface and the `ConnectionFilterRulesListener` interface) must call the `accept()` method after gathering information about the client connection. However, if you only implement the `ConnectionFilter` interface, the information gathered includes the remote IP address and the connection protocol (HTTP, HTTPS, T3, T3S, GIOP, GIOPS, DCOM, FTP, or LDAP). If you implement both interfaces, the information gathered includes the remote IP address, remote port number, local IP address, local port number and the connection protocol.

Connection Filter Examples

Two connection filter examples are available for use with WebLogic Server. Both examples provide an efficient, generalized connection filter. To download the connection filter examples, go to `http://dev2dev.bea.com/code/wls.jsp`. Both examples parse the rules and set up a rule-matching algorithm so that connection filtering adds minimal overhead to a WebLogic Server connection. If necessary, you can modify this sample code and reuse it. You may, for example, want to accommodate the local or remote port number in your filter or a more site-specific algorithm that will reduce filtering overhead. For instructions on how to build, configure, and run these samples, see the `package.html` file included with in the Network Connection Filter code examples.

This section covers the following topics:

- "SimpleConnectionFilter Example" on page 6-8
- "SimpleConnectionFilter2 Example" on page 6-9
- "Example of the accept Method Used in Filtering Network Connections" on page 6-9

SimpleConnectionFilter Example

The `examples.security.net.SimpleConnectionFilter` example is included in the Network Connection Filter code examples, which are available at `http://dev2dev.bea.com/code/wls.jsp` on the BEA dev2dev Web site. This example implements the `ConnectionFilter` interface and filters connections using rules that you define in the filter file.

SimpleConnectionFilter2 Example

The `examples.security.net.SimpleConnectionFilter2` example is included in the Network Connection Filter code examples, which are available at `http://dev2dev.bea.com/code/wls.jsp` on the BEA dev2dev Web site. This example implements the `ConnectionFilter` and `ConnectionFilterRulesListener` interfaces and filters connections using the rules that you define using the Administration Console.

Example of the accept Method Used in Filtering Network Connections

In Listing 6-1, WebLogic Server calls the `SimpleConnectionFilter.accept()` method with a `ConnectionEvent`. The `SimpleConnectionFilter.accept()` method gets the remote address and protocol and converts the protocol to a bitmask to avoid string comparisons in rule-matching. Then the `SimpleConnectionFilter.accept()` method compares the remote address and protocol against each rule until it finds a match.

This code fragment is taken from the `SimpleConnectionFilter.java` file in the Network Connection Filter code examples, which are available at `http://dev2dev.bea.com/code/wls.jsp` on the BEA dev2dev Web site.

Listing 6-1 Example of Filtering Network Connections

```
public void accept(ConnectionEvent evt)
  throws FilterException
{
  InetAddress remoteAddress = evt.getRemoteAddress();
  String protocol = evt.getProtocol().toLowerCase();
  int bit = protocolToMaskBit(protocol);
  // this special bitmask indicates that the
  // connection does not use one of the recognized
  // protocols
  if (bit == 0xdeadbeef)
  {
    bit = 0;
  }
  // Check rules in the order in which they were written.
  for (int i = 0; i < rules.length; i++)
```

```
    {
      switch (rules[i].check(remoteAddress, bit))
      {
      case FilterEntry.ALLOW:
    return;
      case FilterEntry.DENY:
    throw new FilterException("rule " + (i + 1));
      case FilterEntry.IGNORE:
    break;
      default:
    throw new RuntimeException("connection filter internal error!");
      }
    }
    // If no rule matched, we allow the connection to succeed.
    return;
  }
```

Using Java Security to Protect WebLogic Resources

This section discusses the following topics:

- "Using J2EE Security to Protect WebLogic Resources" on page 7-1
- "Using the Java Security Manager to Protect WebLogic Resources" on page 7-2

Using J2EE Security to Protect WebLogic Resources

WebLogic Server supports the use of J2EE security to protect URL (Web), Enterprise JavaBeans (EJBs), and Connector components. In addition, WebLogic Server extends the connector model of specifying additional security policies in the deployment descriptor to the URL and EJB components.

Note: J2EE has requirements for Java 2 security default permissions for different application types (see the J2EE 1.3 specification, section 6.2.2) as does the Connector 1.0 spec (see section 11.2). These specifications are available at `http://java.sun.com/j2ee/download.html#platformspec`.

Furthermore, the J2EE specification suggests that the deployer be able to add to these security policies. For URL and EJB components, this is done through comments in the deployment descriptor, but the specification states: "A future version of this specification will allow these security requirements to be specified in the deployment descriptor for the application components." The connector specification already provides for deployment descriptors to specify additional security policies using the `<security-permission>` tag (see Listing 7-1):

Listing 7-1 Security-Permission Tag Sample

```
<security-permission>
<description> Optional explanation goes here </description>
<security-permission-spec>
<!—
A single grant statement following the syntax of
http://java.sun.com/j2se/1.3/docs/guide/security/PolicyFiles.html#FileSynt
ax without the "codebase" and "signedBy" clauses goes here. For example:
-->
grant {
permission java.net.SocketPermission "*", "resolve";
};
</security-permission-spec>
</security-permission>
```

Besides support of the `<security-permission>` tag in the `rar.xml` file, WebLogic Server adds the `<security-permission>` tag to the `weblogic.xml` and `weblogic-ejb-jar.xml` files. This extends the connector model to the two other application types, Web applications and EJBs, provides a uniform interface to security policies across all component types, and anticipates future J2EE specification changes.

Using the Java Security Manager to Protect WebLogic Resources

The Java Security Manager can be used with WebLogic Server to provide additional protection for resources running in a Java Virtual Machine (JVM). Using a Java Security Manager is an optional security step. The following sections describe how to use the Java Security Manager with WebLogic Server:

- "Setting Up the Java Security Manager" on page 7-3
- "Using the Recording Security Manager Utility" on page 7-6

For more information on Java Security Manager, see the Java Security Web page at `http://java.sun.com/j2se/1.4/docs/guide/security/index.html`.

Setting Up the Java Security Manager

When you run WebLogic Server under Java 2 (SDK 1.2 or later), WebLogic Server can use the Java Security Manager in Java 2, which prevents untrusted code from performing actions that are restricted by the Java security policy file.

The JVM has security mechanisms built into it that allow you to define restrictions to code through a Java security policy file. The Java Security Manager uses the Java security policy file to enforce a set of permissions granted to classes. The permissions allow specified classes running in that instance of the JVM to permit or not permit certain runtime operations. In many cases, where the threat model does not include malicious code being run in the JVM, the Java Security Manager is unnecessary. However, when untrusted third-parties use WebLogic Server and untrusted classes are being run, the Java Security Manager may be useful.

To use the Java Security Manager with WebLogic Server, specify the `-Djava.security.policy` and `-Djava.security.manager` arguments when starting WebLogic Server. The `-Djava.security.policy` argument specifies a filename (using a relative or fully-qualified pathname) that contains Java 2 security policies.

WebLogic Server provides a sample Java security policy file, which you can edit and use. The file is located at *`WL_HOME`*`\server\lib\weblogic.policy`.

If you enable the Java Security Manager but do not specify a security policy file, the Java Security Manager uses the default security policies defined in the `java.policy` file in the *$JAVA_HOME*`\jre\lib\security` directory.

Define security policies for the Java Security Manager in one of the following ways:

- "Modifying the weblogic.policy file for General Use" on page 7-3
- "Setting Application-Type Security Policies" on page 7-4
- "Setting Application-Specific Security Policies" on page 7-5

Modifying the weblogic.policy file for General Use

To use the Java Security Manager security policy file with your WebLogic Server deployment, you must specify the location of the `weblogic.policy` file to the Java Security Manager when you start WebLogic Server. To do this, you set the following arguments on the Java command line you use to start the server:

- `java.security.manager` tells the JVM to use a Java security policy file.

- `java.security.policy` tells the JVM the location of the Java security policy file to use. The argument is the fully qualified name of the Java security policy, which in this case is `weblogic.policy`.

For example:

```
java...-Djava.security.manager \
   -Djava.security.policy==c:\weblogic\weblogic.policy
```

Note: Be sure to use `==` instead of `=` when specifying the `java.security.policy` argument so that only the `weblogic.policy` file is used by the Java Security Manager. The `==` causes the `weblogic.policy` file to override any default security policy. A single equal sign (`=`) causes the `weblogic.policy` file to be appended to an existing security policy.

If you have extra directories in your `CLASSPATH` or if you are deploying applications in extra directories, add specific permissions for those directories to your `weblogic.policy` file.

BEA recommends taking the following precautions when using the `weblogic.policy` file:

– Make a backup copy of the `weblogic.policy` file and put the backup copy in a secure location.

– Set the permissions on the `weblogic.policy` file via the operating system such that the administrator of the WebLogic Server deployment has write and read privileges and no other users have access to the file.

Caution: The Java Security Manager is partially disabled during the booting of Administration and Managed Servers. During the boot sequence, the current Java Security Manager is disabled and replaced with a variation of the Java Security Manager that has the `checkRead()` method disabled. While disabling this method greatly improves the performance of the boot sequence, it also minimally diminishes security. The startup classes for WebLogic Server are run with this partially disabled Java Security Manager and therefore the classes need to be carefully scrutinized for security considerations involving the reading of files.

For more information about the Java Security Manager, see the Javadoc for the `java.lang.SecurityManager` class which is available on the Web `http://java.sun.com/j2se/1.4.1/docs/api/index.html`.

Setting Application-Type Security Policies

Set default security policies for Servlets, EJBs, and J2EE Connector Resource Adapters in the Java security policy file. The default security policies for Servlets, EJBs, and Resource Adapters are defined in the Java security policy file under the following codebases:

- Servlets—`"file:/weblogic/application/defaults/Web"`
- EJBs—`"file:/weblogic/application/defaults/EJB"`
- Resource Adapters—`"file:/weblogic/application/defaults/Connectors"`

Note: These security policies apply to all Servlets, EJBs, and Resource Adapters deployed in the particular instance of WebLogic Server.

Setting Application-Specific Security Policies

Set security policies for a specific Servlet, EJB, or Resource Adapter by adding security policies to their deployment descriptors. Deployment descriptors are defined in the following files:

- Servlets—`weblogic.xml`
- EJBs—`weblogic-ejb-jar.xml`
- Resource Adapters—`rar.xml`

Note: The security policies for Resource Adapters follow the J2EE standard while the security policies for Servlets and EJBs follow the WebLogic Server extension to the J2EE standard.

Listing 7-2 shows the syntax for adding a security policy to a deployment descriptor:

Listing 7-2 Security Policy Syntax

```
<security-permission>
 <description>
  Allow getting the J2EEJ2SETest4 property
 </description>
 <security-permission-spec>
  grant {
   permission java.util.PropertyPermission "welcome.J2EEJ2SETest4","read";
  };
 </security-permission-spec>
</security-permission>
```

Note: The `<security-permission-spec>` tag cannot currently be added to a `weblogic-application.xml` file, you are limited to using this tag within a `weblogic-ejb-jar.xml`, `rar.xml`, or `weblogic.xml` file. Also, variables are not supported in the `<security-permission-spec>` attribute.

Using the Recording Security Manager Utility

The Recording Security Manager utility can be used to detect permission problems that occur when starting and running WebLogic Server. The utility outputs permissions that can be added to your Java security policy file to resolve the permission problems that the utility finds. The Recording Security Manager is available at the BEA dev2dev Online.

Deprecated Security APIs

Some or all of the Security interfaces, classes, and exceptions in the following WebLogic security packages were deprecated in this release of WebLogic Server:

- `weblogic.security`
- `weblogic.security.acl`
- `weblogic.security.audit`
- `weblogic.security.SSL`

For specific information on the interfaces, classes, and exceptions deprecated in each package, see the Javadocs for WebLogic Classes.

Index