Sun GlassFish Enterprise Server v3 Application Development Guide

Part No: 820–7695–11
January 2010

ORACLE

100608@20205

Contents

Tables

Preface

This *Application Development Guide* describes how to create and run Java Platform, Enterprise Edition (Java EE platform) applications that follow the open Java standards model for Java EE components and APIs in the Sun GlassFish Enterprise Server environment. Topics include developer tools, security, and debugging. This book is intended for use by software developers who create, assemble, and deploy Java EE applications using Sun GlassFish servers and software.

This preface contains information about and conventions for the entire Sun GlassFish Enterprise Server (Enterprise Server) documentation set.

Enterprise Server v3 is developed through the GlassFish project open-source community at `https://glassfish.dev.java.net/`. The GlassFish project provides a structured process for developing the Enterprise Server platform that makes the new features of the Java EE platform available faster, while maintaining the most important feature of Java EE: compatibility. It enables Java developers to access the Enterprise Server source code and to contribute to the development of the Enterprise Server. The GlassFish project is designed to encourage communication between Sun engineers and the community.

The following topics are addressed here:

Enterprise Server Documentation Set

The Enterprise Server documentation set describes deployment planning and system installation. The Uniform Resource Locator (URL) for Enterprise Server documentation is `http://docs.sun.com/coll/1343.9`. For an introduction to Enterprise Server, refer to the books in the order in which they are listed in the following table.

TABLE P–1 Books in the Enterprise Server Documentation Set

Book Title	Description
Release Notes	Provides late-breaking information about the software and the documentation. Includes a comprehensive, table-based summary of the supported hardware, operating system, Java Development Kit (JDK), and database drivers.
Quick Start Guide	Explains how to get started with the Enterprise Server product.
Installation Guide	Explains how to install the software and its components.
Upgrade Guide	Explains how to upgrade to the latest version of Enterprise Server. This guide also describes differences between adjacent product releases and configuration options that can result in incompatibility with the product specifications.
Administration Guide	Explains how to configure, monitor, and manage Enterprise Server subsystems and components from the command line by using the `asadmin(1M)` utility. Instructions for performing these tasks from the Administration Console are provided in the Administration Console online help.
Application Deployment Guide	Explains how to assemble and deploy applications to the Enterprise Server and provides information about deployment descriptors.
Your First Cup: An Introduction to the Java EE Platform	Provides a short tutorial for beginning Java EE programmers that explains the entire process for developing a simple enterprise application. The sample application is a web application that consists of a component that is based on the Enterprise JavaBeans specification, a JAX-RS web service, and a JavaServer Faces component for the web front end.
Application Development Guide	Explains how to create and implement Java Platform, Enterprise Edition (Java EE platform) applications that are intended to run on the Enterprise Server. These applications follow the open Java standards model for Java EE components and APIs. This guide provides information about developer tools, security, and debugging.
Add-On Component Development Guide	Explains how to use published interfaces of Enterprise Server to develop add-on components for Enterprise Server. This document explains how to perform *only* those tasks that ensure that the add-on component is suitable for Enterprise Server.

TABLE P–1 Books in the Enterprise Server Documentation Set *(Continued)*

Book Title	Description
Embedded Server Guide	Explains how to run applications in embedded Enterprise Server and to develop applications in which Enterprise Server is embedded.
Scripting Framework Guide	Explains how to develop scripting applications in languages such as Ruby on Rails and Groovy on Grails for deployment to Enterprise Server.
Troubleshooting Guide	Describes common problems that you might encounter when using Enterprise Server and how to solve them.
Error Message Reference	Describes error messages that you might encounter when using Enterprise Server.
Reference Manual	Provides reference information in man page format for Enterprise Server administration commands, utility commands, and related concepts.
Domain File Format Reference	Describes the format of the Enterprise Server configuration file, `domain.xml`.
Java EE 6 Tutorial, Volume I	Explains how to use Java EE 6 platform technologies and APIs to develop Java EE applications.
Message Queue Release Notes	Describes new features, compatibility issues, and existing bugs for Sun GlassFish Message Queue.
Message Queue Administration Guide	Explains how to set up and manage a Sun GlassFish Message Queue messaging system.
Message Queue Developer's Guide for JMX Clients	Describes the application programming interface in Sun GlassFish Message Queue for programmatically configuring and monitoring Message Queue resources in conformance with the Java Management Extensions (JMX).
System Virtualization Support in Sun Java System Products	Summarizes Sun support for Sun Java System products when used in conjunction with system virtualization products and features.

Related Documentation

The Java EE 6 Tutorial, Volume II (`https://www.sun.com/offers/details/java_ee6_tutorial.xml`) contains all the topics in *Java EE 6 Tutorial, Volume I* and adds advanced topics, additional technologies, and case studies. The document is available to registered users of Enterprise Server.

Javadoc tool reference documentation for packages that are provided with Enterprise Server is available as follows:

- The API specification for version 6 of Java EE is located at `http://java.sun.com/javaee/6/docs/api/`.
- The API specification for Enterprise Server v3, including Java EE 6 platform packages and nonplatform packages that are specific to the Enterprise Server product, is located at: `https://glassfish.dev.java.net/nonav/docs/v3/api/`.

Additionally, the following resources might be useful:

- The Java EE Specifications (`http://java.sun.com/javaee/technologies/index.jsp`)
- The Java EE Blueprints (`http://java.sun.com/reference/blueprints/index.html`)

For information about creating enterprise applications in the NetBeans Integrated Development Environment (IDE), see `http://www.netbeans.org/kb/60/index.html`.

For information about the Java DB for use with the Enterprise Server, see `http://developers.sun.com/javadb/`.

The sample applications demonstrate a broad range of Java EE technologies. The samples are bundled with the Java EE Software Development Kit (SDK).

Typographic Conventions

The following table describes the typographic changes that are used in this book.

TABLE P–2 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	A placeholder to be replaced with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized (note that some emphasized items appear bold online)	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file.

Symbol Conventions

The following table explains symbols that might be used in this book.

TABLE P–3 Symbol Conventions

Symbol	Description	Example	Meaning
`[ ]`	Contains optional arguments and command options.	`ls [-l]`	The `-l` option is not required.
`{ | }`	Contains a set of choices for a required command option.	`-d {y|n}`	The `-d` option requires that you use either the `y` argument or the `n` argument.
`${ }`	Indicates a variable reference.	`${com.sun.javaRoot}`	References the value of the `com.sun.javaRoot` variable.
-	Joins simultaneous multiple keystrokes.	Control-A	Press the Control key while you press the A key.
+	Joins consecutive multiple keystrokes.	Ctrl+A+N	Press the Control key, release it, and then press the subsequent keys.
→	Indicates menu item selection in a graphical user interface.	File → New → Templates	From the File menu, choose New. From the New submenu, choose Templates.

Default Paths and File Names

The following table describes the default paths and file names that are used in this book.

TABLE P–4 Default Paths and File Names

Placeholder	Description	Default Value
as-install	Represents the base installation directory for Enterprise Server. In configuration files, *as-install* is represented as follows: `${com.sun.aas.installRoot}`	Installations on the Solaris operating system, Linux operating system, and Mac operating system: *user's-home-directory*`/glassfishv3/glassfish` Windows, all installations: *SystemDrive*`:\glassfishv3\glassfish`
as-install-parent	Represents the parent of the base installation directory for Enterprise Server.	Installations on the Solaris operating system, Linux operating system, and Mac operating system: *user's-home-directory*`/glassfishv3` Windows, all installations: *SystemDrive*`:\glassfishv3`
domain-root-dir	Represents the directory in which a domain is created by default.	*as-install*`/domains/`

TABLE P–4 Default Paths and File Names *(Continued)*

Placeholder	Description	Default Value
domain-dir	Represents the directory in which a domain's configuration is stored. In configuration files, *domain-dir* is represented as follows: `${com.sun.aas.instanceRoot}`	*domain-root-dir*/*domain-name*

Documentation, Support, and Training

The Sun web site provides information about the following additional resources:

- Documentation (`http://www.sun.com/documentation/`)
- Support (`http://www.sun.com/support/`)
- Training (`http://www.sun.com/training/`)

Searching Sun Product Documentation

Besides searching Sun product documentation from the docs.sun.com web site, you can use a search engine by typing the following syntax in the search field:

search-term `site:docs.sun.com`

For example, to search for "broker," type the following:

```
broker site:docs.sun.com
```

To include other Sun web sites in your search (for example, java.sun.com, www.sun.com, and developers.sun.com), use `sun.com` in place of `docs.sun.com` in the search field.

Third-Party Web Site References

Third-party URLs are referenced in this document and provide additional, related information.

Note – Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. To share your comments, go to `http://docs.sun.com` and click Send Comments. In the online form, provide the full document title and part number. The part number is a 7-digit or 9-digit number that can be found on the book's title page or in the document's URL. For example, the part number of this book is 820-7695.

PART I

Development Tasks and Tools

◆ ◆ ◆ CHAPTER 1

Setting Up a Development Environment

This chapter gives guidelines for setting up an application development environment in the Sun GlassFish Enterprise Server. Setting up an environment for creating, assembling, deploying, and debugging your code involves installing the mainstream version of the Enterprise Server and making use of development tools. In addition, sample applications are available. These topics are covered in the following sections:

Installing and Preparing the Server for Development

For more information about Enterprise Server installation, see the *Sun GlassFish Enterprise Server v3 Installation Guide*.

The following components are included in the full installation.

- JDK
- Enterprise Server core
  - Java Platform, Standard Edition (Java SE) 6
  - Java EE 6 compliant application server
  - Administration Console
  - `asadmin` utility
  - Other development and deployment tools
  - Sun GlassFish Message Queue software
  - Java DB database, based on the Derby database from Apache (`http://db.apache.org/derby/manuals`)

The NetBeans Integrated Development Environment (IDE) bundles the GlassFish edition of the Enterprise Server, so information about this IDE is provided as well.

After you have installed Enterprise Server, you can further optimize the server for development in these ways:

- Locate utility classes and libraries so they can be accessed by the proper class loaders. For more information, see "Using the Common Class Loader" on page 38.
- Set up debugging. For more information, see Chapter 4, "Debugging Applications."
- Configure the Virtual Machine for the Java platform (JVM software). For more information, see Chapter 4, "Administering the Virtual Machine for the Java Platform," in *Sun GlassFish Enterprise Server v3 Administration Guide*.

Development Tools

The following general tools are provided with the Enterprise Server:

- "The `asadmin` Command" on page 28
- "The Administration Console" on page 29

The following development tools are provided with the Enterprise Server or downloadable from Sun:

- "Enterprise Server Ant Tasks" on page 29
- "Scripting Language Support" on page 29
- "The Migration Tool" on page 29
- "The NetBeans IDE" on page 29

The following third-party tools might also be useful:

- "The Eclipse IDE" on page 30
- "Debugging Tools" on page 30
- "Profiling Tools" on page 30

The `asadmin` Command

The `asadmin` command allows you to configure a local or remote server and perform both administrative and development tasks at the command line. For general information about `asadmin`, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

The `asadmin` command is located in the *as-install*`/bin` directory. Type `asadmin help` for a list of subcommands.

The Administration Console

The Administration Console lets you configure the server and perform both administrative and development tasks using a web browser. For general information about the Administration Console, click the Help button in the Administration Console. This displays the Enterprise Server online help.

To access the Administration Console, type `http://`*host*`:4848` in your browser. The *host* is the name of the machine on which the Enterprise Server is running. By default, the *host* is `localhost`. For example:

```
http://localhost:4848
```

Enterprise Server Ant Tasks

The Enterprise Server provides server-specific tasks for administration and deployment; see Chapter 3, "Using Ant with Enterprise Server." The sample applications that can be used with the Enterprise Server use Ant `build.xml` files; see "Sample Applications" on page 30.

For more information about Ant, see the Apache Software Foundation web site at `http://ant.apache.org/`.

Scripting Language Support

The Enterprise Server supports a number of scripting languages, including Ruby on Rails and Groovy on Grails. For more information, see the *Sun GlassFish Enterprise Server v3 Scripting Framework Guide*.

The Migration Tool

The Migration Tool converts and reassembles Java EE applications and modules developed on other application servers. This tool also generates a report listing how many files are successfully and unsuccessfully migrated, with reasons for migration failure. For more information and to download the Migration Tool, see `http://java.sun.com/j2ee/tools/migration/index.html`.

The NetBeans IDE

The NetBeans IDE allows you to create, assemble, and debug code from a single, easy-to-use interface. The GlassFish edition of the Enterprise Server is bundled with the NetBeans 6.1 IDE. To download the NetBeans IDE, see `http://www.netbeans.org`. This site also provides documentation on how to use the NetBeans IDE with the bundled GlassFish edition of the Enterprise Server.

You can also use the Enterprise Server with the Sun Java Studio 8 software, which is built on the NetBeans IDE. For more information, see `http://developers.sun.com/prodtech/javatools/jsenterprise/`.

The Eclipse IDE

A plug-in for the Eclipse IDE is available at `http://glassfishplugins.dev.java.net/`. This site also provides documentation on how to register the Enterprise Server and use Sun-specific deployment descriptors.

Debugging Tools

You can use several debugging tools with the Enterprise Server. For more information, see Chapter 4, "Debugging Applications."

Profiling Tools

You can use several profilers with the Enterprise Server. For more information, see "Profiling Tools" on page 71.

Sample Applications

Sample applications that you can examine and deploy to the Enterprise Server are available. If you installed the Enterprise Server as part of installing the Java EE 6 SDK bundle from Java EE 6 Downloads (`http://java.sun.com/javaee/downloads/index.jsp`), the samples may already be installed. You can download these samples separately from the Code Samples (`http://java.sun.com/javaee/reference/code/index.jsp`) page if you installed the Enterprise Server without them initially.

Most Enterprise Server samples have the following directory structure:

- The `docs` directory contains instructions for how to use the sample.
- The `build.xml` file defines Ant targets for the sample. See Chapter 3, "Using Ant with Enterprise Server."
- The `src/java` directory under each component contains source code for the sample.
- The `src/conf` directory under each component contains the deployment descriptors.

With a few exceptions, sample applications follow the standard directory structure described here: `http://java.sun.com/blueprints/code/projectconventions.html`.

The *samples-install-dir*/`bp-project/main.xml` file defines properties common to all sample applications and implements targets needed to compile, assemble, deploy, and undeploy sample applications. In most sample applications, the `build.xml` file imports `main.xml`.

In addition to the Java EE 6 sample applications, samples are also available at GlassFish Samples (`https://glassfish-samples.dev.java.net/`) and at *as-install*/`glassfish/samples/`.

◆ ◆ ◆ CHAPTER 2

Class Loaders

Understanding Sun GlassFish Enterprise Server class loaders can help you determine where to place supporting JAR and resource files for your modules and applications. For general information about J2SE class loaders, see Understanding Network Class Loaders (`http://java.sun.com/developer/technicalArticles/Networking/classloaders/`).

In a JVM implementation, the class loaders dynamically load a specific Java class file needed for resolving a dependency. For example, when an instance of `java.util.Enumeration` needs to be created, one of the class loaders loads the relevant class into the environment. This section includes the following topics:

Note – The Web Profile of the Enterprise Server supports the EJB 3.1 Lite specification, which allows enterprise beans within web applications, among other features. The full Enterprise Server supports the entire EJB 3.1 specification. For details, see JSR 318 (`http://jcp.org/en/jsr/detail?id=318`).

The Class Loader Hierarchy

Class loaders in the Enterprise Server runtime follow a delegation hierarchy that is fully described in Table 2–1.

TABLE 2–1 Sun GlassFish Enterprise Server Class Loaders

Class Loader	Description
Bootstrap	The Bootstrap class loader loads the basic runtime classes provided by the JVM software.
Extension	The Extension class loader loads classes from JAR files present in the system extensions directory, *domain-dir*`/lib/ext`. It is parent to the Public API class loader. See "Using the Java Optional Package Mechanism" on page 35.
Public API	The Public API class loader makes available all classes specifically exported by the Enterprise Server runtime for use by deployed applications. This includes, but is not limited to, Java EE APIs and other Sun GlassFish APIs. It is parent to the Common class loader.
Common	The Common class loader loads JAR files in the *as-install*`/lib` directory, then classes in the *domain-dir*`/lib/classes` directory, followed by JAR files in the *domain-dir*`/lib` directory. Using *domain-dir*`/lib/classes` or *domain-dir*`/lib` is recommended whenever possible, and required for custom login modules and realms. It is parent to the Connector class loader. See "Using the Common Class Loader" on page 38.
Connector	The Connector class loader is a single class loader instance that loads individually deployed connector modules, which are shared across all applications. It is parent to the Applib class loader and the LifeCycleModule class loader.
LifeCycleModule	The LifeCycleModule class loader is created once per lifecycle module. Each lifecycle module's classpath is used to construct its own class loader. For more information on lifecycle modules, see Chapter 13, "Developing Lifecycle Listeners."
Applib	The Applib class loader loads the library classes, specified during deployment, for a specific enabled module or Java EE application; see "Application-Specific Class Loading" on page 36. One instance of this class loader is present in each class loader universe; see "Class Loader Universes" on page 36. It is parent to the Archive class loader. When multiple deployed applications use the same library, they share the same instance of the library. One library cannot reference classes from another library.
Archive	The Archive class loader loads classes from the WAR, EAR, and JAR files or directories (for directory deployment) of applications or modules deployed to the Enterprise Server. This class loader also loads any application-specific classes generated by the Enterprise Server runtime, such as stub classes or servlets generated by JSP pages.

Delegation

Note that the class loader hierarchy is not a Java inheritance hierarchy, but a delegation hierarchy. In the delegation design, a class loader delegates class loading to its parent before attempting to load a class itself. If the parent class loader cannot load a class, the class loader attempts to load the class itself. In effect, a class loader is responsible for loading only the classes not available to the parent. Classes loaded by a class loader higher in the hierarchy cannot refer to classes available lower in the hierarchy.

The Java Servlet specification recommends that a web module's class loader look in the local class loader before delegating to its parent. You can make this class loader follow the delegation inversion model in the Servlet specification by setting `delegate="false"` in the `class-loader` element of the `sun-web.xml` file. It is safe to do this only for a web module that does not interact with any other modules. For details, see "class-loader" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

The default value is `delegate="true"`, which causes a web module's class loader to delegate in the same manner as the other class loaders. You must use `delegate="true"` for a web application that accesses EJB components or that acts as a web service client or endpoint. For details about `sun-web.xml`, see *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

For a number of packages, including `java.*` and `javax.*`, symbol resolution is always delegated to the parent class loader regardless of the `delegate` setting. This prevents applications from overriding core Java runtime classes or changing the API versions of specifications that are part of the Java EE platform.

Using the Java Optional Package Mechanism

Optional packages are packages of Java classes and associated native code that application developers can use to extend the functionality of the core platform.

To use the Java optional package mechanism, copy the JAR files into the *domain-dir*`/lib/ext` directory, then restart the server.

For more information, see Optional Packages - An Overview (`http://java.sun.com/javase/6/docs/technotes/guides/extensions/extensions.html`) and Understanding Extension Class Loading (`http://java.sun.com/docs/books/tutorial/ext/basics/load.html`).

Using the Endorsed Standards Override Mechanism

Endorsed standards handle changes to classes and APIs that are bundled in the JDK but are subject to change by external bodies.

To use the endorsed standards override mechanism, copy the JAR files into the *domain-dir*`/lib/endorsed` directory, then restart the server.

For more information and the list of packages that can be overridden, see Endorsed Standards Override Mechanism (`http://java.sun.com/javase/6/docs/technotes/guides/standards/`).

Class Loader Universes

Access to components within applications and modules installed on the server occurs within the context of isolated class loader universes, each of which has its own Applib and Archive class loaders.

- **Application Universe** – Each Java EE application has its own class loader universe, which loads the classes in all the modules in the application.
- **Individually Deployed Module Universe** – Each individually deployed EJB JAR or web WAR has its own class loader universe, which loads the classes in the module.

A resource such as a file that is accessed by a servlet, JSP, or EJB component must be in one of the following locations:

- A directory pointed to by the Libraries field or `--libraries` option used during deployment
- A directory pointed to by the `library-directory` element in the `application.xml` deployment descriptor
- A directory pointed to by the application or module's classpath; for example, a web module's classpath includes these directories:

  module-name`/WEB-INF/classes`
  module-name`/WEB-INF/lib`

Application-Specific Class Loading

You can specify module- or application-specific library classes during deployment in one of the following ways:

- Use the Administration Console. Open the Applications component, then go to the page for the type of application or module. Select the Deploy button. Type the comma-separated paths in the Libraries field. For details, click the Help button in the Administration Console.

- Use the `asadmin deploy` command with the `--libraries` option and specify comma-separated paths. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Note – The Libraries field in the Administration Console's deployment page and the `--libraries` option of the `asadmin deploy` command do not apply to application clients. For more information, see "Using Libraries with Application Clients" on page 230.

You can only specify module- or application-specific library classes during deployment. You can update a library JAR file using dynamic reloading or by restarting (disabling and re-enabling) a module or application. To add or remove library JAR files, you must redeploy the module or application.

Application libraries are included in the Applib class loader. Paths to libraries can be relative or absolute. A relative path is relative to *domain-dir*`/lib/applibs`. If the path is absolute, the path must be accessible to the domain administration server (DAS).

Tip – You can use application-specific class loading to specify a different XML parser than the default Enterprise Server XML parser. For details, see `http://blogs.sun.com/sivakumart/entry/classloaders_in_glassfish_an_attempt`.

You can also use application-specific class loading to access different versions of a library from different applications.

If multiple applications or modules refer to the same libraries, classes in those libraries are automatically shared. This can reduce the memory footprint and allow sharing of static information. However, applications or modules using application-specific libraries are not portable. Other ways to make libraries available are described in "Circumventing Class Loader Isolation" on page 38.

One library cannot reference classes from another library.

For general information about deployment, including dynamic reloading, see the *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

Note – If you see an access control error message when you try to use a library, you may need to grant permission to the library in the `server.policy` file. For more information, see "Changing Permissions for an Application" on page 89.

Circumventing Class Loader Isolation

Since each application or individually deployed module class loader universe is isolated, an application or module cannot load classes from another application or module. This prevents two similarly named classes in different applications or modules from interfering with each other.

To circumvent this limitation for libraries, utility classes, or individually deployed modules accessed by more than one application, you can include the relevant path to the required classes in one of these ways:

- "Using the Common Class Loader" on page 38
- "Packaging the Client JAR for One Application in Another Application" on page 38

Using the Common Class Loader

To use the Common class loader, copy the JAR files into the *domain-dir*`/lib` or *as-install*`/lib` directory or copy the `.class` files (and other needed files, such as `.properties` files) into the *domain-dir*`/lib/classes` directory, then restart the server.

Using the Common class loader makes an application or module accessible to all applications or modules deployed on servers that share the same configuration. However, this accessibility does not extend to application clients. For more information, see "Using Libraries with Application Clients" on page 230.

For example, using the Common class loader is the recommended way of adding JDBC drivers to the Enterprise Server. For a list of the JDBC drivers currently supported by the Enterprise Server, see the *Sun GlassFish Enterprise Server v3 Release Notes*. For configurations of supported and other drivers, see "Configuration Specifics for JDBC Drivers" in *Sun GlassFish Enterprise Server v3 Administration Guide*.

To activate custom login modules and realms, place the JAR files in the *domain-dir*`/lib` directory or the class files in the *domain-dir*`/lib/classes` directory, then restart the server.

Packaging the Client JAR for One Application in Another Application

By packaging the client JAR for one application in a second application, you allow an EJB or web component in the second application to call an EJB component in the first (dependent) application, without making either of them accessible to any other application or module.

As an alternative for a production environment, you can have the Common class loader load the client JAR of the dependent application as described in "Using the Common Class Loader" on page 38. Restart the server to make the dependent application accessible to all applications or modules deployed on servers that share the same configuration.

▼ To Package the Client JAR for One Application in Another Application

1. **Deploy the dependent application.**

2. **Add the dependent application's client JAR file to the calling application.**

   - For a calling EJB component, add the client JAR file at the same level as the EJB component. Then add a `Class-Path` entry to the `MANIFEST.MF` file of the calling EJB component. The `Class-Path` entry has this syntax:

     ```
     Class-Path: filepath1.jar filepath2.jar ...
     ```

     Each *filepath* is relative to the directory or JAR file containing the `MANIFEST.MF` file. For details, see the Java EE specification.
   - For a calling web component, add the client JAR file under the `WEB-INF/lib` directory.

3. **If you need to package the client JAR with both the EJB and web components, set `delegate="true"` in the `class-loader` element of the `sun-web.xml` file.**

   This changes the Web class loader so that it follows the standard class loader delegation model and delegates to its parent before attempting to load a class itself.

   For most applications, packaging the client JAR file with the calling EJB component is sufficient. You do not need to package the client JAR file with both the EJB and web components unless the web component is directly calling the EJB component in the dependent application.

4. **Deploy the calling application.**

   The calling EJB or web component must specify in its `sun-ejb-jar.xml` or `sun-web.xml` file the JNDI name of the EJB component in the dependent application. Using an `ejb-link` mapping does not work when the EJB component being called resides in another application.

   You do not need to restart the server.

◆ ◆ ◆ CHAPTER 3

Using Ant with Enterprise Server

The Sun GlassFish Enterprise Server provides server-specific Ant tasks, which are described in the following sections:

Enterprise Server is compatible with Apache Ant versions 1.6.5 or greater. If you don't have Ant installed, you can download it from the Update Tool. The Apache Ant Build Tool add-on component supplies Ant version 1.7.1. For more information about the Update Tool, see "Update Tool" in *Sun GlassFish Enterprise Server v3 Administration Guide*.

For more information about Ant, see the Apache Software Foundation web site at `http://ant.apache.org/`.

For information about standard Ant tasks, see the Ant documentation at `http://ant.apache.org/manual/`.

Note – Variables in the examples in this chapter, such as `${asinstalldir}`, reference values defined in `build.xml` or properties files.

Setting Up Your Ant Environment

To set up your Ant environment for using Enterprise Server Ant tasks, you can either define the `ANT_OPTS` environment variable or define a target. In both these cases, you must also set the classpath to point to the `sun-appserv-ant.jar` file.

Defining the ANT_OPTS Variable

To define the ANT_OPTS environment variable on UNIX systems, use the following commands, where ${ASINSTALLDIR} is an environment variable defined to point to the Enterprise Server installation directory.

```
export ANT_OPTS="-Djava.library.path${ASINSTALLDIR}/modules"
export CLASSPATH=${CLASSPATH};${ASINSTALLDIR}/modules/sun-appserv-ant.jar
```

To define the ANT_OPTS environment variable on Windows systems, use the following commands, where %ASINSTALLDIR% is an environment variable defined to point to the Enterprise Server installation directory.

```
set ANT_OPTS="-Djava.library.path=%ASINSTALLDIR%\modules"
set CLASSPATH=%CLASSPATH%;%ASINSTALLDIR%\modules\sun-appserv-ant.jar
```

Defining a Target

The following target element defines the Enterprise Server Ant tasks and references the sun-appserv-ant.jar file. You can include this target in your build.xml file. The ${asinstalldir} in the classpath element refers to the Enterprise Server installation directory.

```
<target name="as-ant-init">
   <taskdef name="sun-appserv-deploy"
      classname="org.apache.tools.ant.taskdefs.optional.sun.appserv.DeployTask" />
   <taskdef name="sun-appserv-undeploy"
      classname="org.apache.tools.ant.taskdefs.optional.sun.appserv.UndeployTask" />
   <taskdef name="sun-appserv-component"
      classname="org.apache.tools.ant.taskdefs.optional.sun.appserv.ComponentTask" />
   <taskdef name="sun-appserv-admin"
      classname="org.apache.tools.ant.taskdefs.optional.sun.appserv.AdminTask" />
   <taskdef name="sun-appserv-jspc"
      classname="org.apache.tools.ant.taskdefs.optional.sun.appserv.SunJspc" />
   <taskdef name="sun-appserv-update"
      classname="org.apache.tools.ant.taskdefs.optional.sun.appserv.UpdateTask" />
   <taskdef name="sun-appserv-instance"
      classname="org.apache.tools.ant.taskdefs.optional.sun.appserv.InstanceTask" />
    <taskdef name="wsgen" classname="com.sun.tools.ws.ant.WsGen" />
    <taskdef name="wsimport" classname="com.sun.tools.ws.ant.WsImport" />
   <classpath path="${asinstalldir}/lib/sun-appserv-ant.jar" />
</target>
```

Targets that use the Enterprise Server Ant tasks then can use the as-ant-init target as a dependency. For example:

```
<target name="create-some-jdbc-resource" depends="as-ant-init">
   ...
</target>
```

Ant resolves properties from top to bottom in Ant build files. If you define the Enterprise Server Ant tasks at the project level, make sure that any properties used within the task definitions have been resolved before the task definitions. For example, the following snippet defines the `sun-appserv-admin` task at the project level:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="glassfish-admin-ant-tasks" default="default">
    <property name="asinstalldir" value="c:/glassfishv3/glassfish" />
    <taskdef name="sun-appserv-admin"
        classname="org.apache.tools.ant.taskdefs.optional.sun.appserv.AdminTask"
        classpath="${asintalldir}/modules/sun-appserv-ant.jar" />
    ...
</project>
```

Enterprise Server Ant Tasks

Use the Ant tasks provided by the Enterprise Server for assembling, deploying, and undeploying modules and applications, and for configuring the server. The tasks are as follows:

The `sun-appserv-deploy` Task

Deploys any of the following to a local or remote Enterprise Server instance.

- Enterprise application (EAR file)
- Web application (WAR file)
- Enterprise Java Bean (EJB-JAR file)
- Enterprise connector (RAR file)
- Application client

Subelements of `sun-appserv-deploy`

The following table describes subelements for the `sun-appserv-deploy` task. These are objects upon which this task acts.

TABLE 3–1 The `sun-appserv-deploy` Subelements

Element	Description
"The `server` Subelement" on page 61	An Enterprise Server instance
"The `component` Subelement" on page 63	A component to be deployed
"The `fileset` Subelement" on page 65	A set of component files that match specified parameters

Attributes of `sun-appserv-deploy`

The following table describes attributes for the `sun-appserv-deploy` task.

TABLE 3–2 The `sun-appserv-deploy` Attributes

Attribute	Default	Description
`file`	none	(optional if a `component` or `fileset` subelement is present, otherwise required) The component to deploy. If this attribute refers to a file, it must be a valid archive. If this attribute refers to a directory, it must contain a valid archive in which all components have been exploded. If `upload` is set to `false`, this must be an absolute path on the server machine.
`name`	file name without extension	(optional) The display name for the component being deployed.
`force`	`true`	(optional) If `true`, the component is overwritten if it already exists on the server. If `false`, `sun-appserv-deploy` fails if the component exists.
`retrievestubs`	client stubs not saved	(optional) The directory where client stubs are saved. This attribute is inherited by nested `component` elements.
`precompilejsp`	`false`	(optional) If `true`, all JSP files found in an enterprise application (`.ear`) or web application (`.war`) are precompiled. This attribute is ignored for other component types. This attribute is inherited by nested `component` elements.
`verify`	`false`	(optional) If `true`, syntax and semantics for all deployment descriptors are automatically verified for correctness. This attribute is inherited by nested `component` elements.
`contextroot`	file name without extension	(optional) The context root for a web module (WAR file). This attribute is ignored if the component is not a WAR file.

TABLE 3–2 The `sun-appserv-deploy` Attributes *(Continued)*

Attribute	Default	Description
`dbvendorname`	`sun-ejb-jar.xml` entry	(optional) The name of the database vendor for which tables can be created. Allowed values are `javadb`, `db2`, `mssql`, `oracle`, `postgresql`, `pointbase`, `derby` (also for CloudScape), and `sybase`, case-insensitive. If not specified, the value of the `database-vendor-name` attribute in `sun-ejb-jar.xml` is used. If no value is specified, a connection is made to the resource specified by the `jndi-name` subelement of the `cmp-resource` element in the `sun-ejb-jar.xml` file, and the database vendor name is read. If the connection cannot be established, or if the value is not recognized, SQL-92 compliance is presumed. For details, see "Generation Options for CMP" on page 197.
`createtables`	`sun-ejb-jar.xml` entry	(optional) If `true`, causes database tables to be created for beans that need them. If `false`, does not create tables. If not specified, the value of the `create-tables-at-deploy` attribute in `sun-ejb-jar.xml` is used. For details, see "Generation Options" on page 124 and "Generation Options for CMP" on page 197.
`dropandcreatetables`	`sun-ejb-jar.xml` entry	(optional) If `true`, and if tables were automatically created when this application was last deployed, tables from the earlier deployment are dropped and fresh ones are created. If `true`, and if tables were *not* automatically created when this application was last deployed, no attempt is made to drop any tables. If tables with the same names as those that would have been automatically created are found, the deployment proceeds, but a warning indicates that tables could not be created. If `false`, settings of `create-tables-at-deploy` or `drop-tables-at-undeploy` in the `sun-ejb-jar.xml` file are overridden. For details, see "Generation Options" on page 124 and "Generation Options for CMP" on page 197.
`uniquetablenames`	`sun-ejb-jar.xml` entry	(optional) If `true`, specifies that table names are unique within each application server domain. If not specified, the value of the `use-unique-table-names` property in `sun-ejb-jar.xml` is used. For details, see "Generation Options for CMP" on page 197.
`enabled`	`true`	(optional) If `true`, enables the component.
`deploymentplan`	none	(optional) A deployment plan is a JAR file containing Sun-specific descriptors. Use this attribute when deploying an EAR file that lacks Sun-specific descriptors.
`availabilityenabled`	`false`	(optional) If `true`, enables high availability features, including persistence of HTTP sessions and checkpointing of the stateful session bean state.
`generatermistubs`	`false`	(optional) If `true`, generates the static RMI-IIOP stubs and puts them in the client JAR file.

TABLE 3–2 The `sun-appserv-deploy` Attributes *(Continued)*

Attribute	Default	Description
`upload`	`true`	(optional) If `true`, the component is transferred to the server for deployment. If the component is being deployed on the local machine, set upload to `false` to reduce deployment time. If a directory is specified for deployment, upload must be `false`.
`virtualservers`	default virtual server only	(optional) A comma-separated list of virtual servers to be deployment targets. This attribute applies only to application (`.ear`) or web (`.war`) components and is ignored for other component types. This attribute is inherited by nested `server` elements.
`user`	`admin`	(optional) The user name used when logging into the application server administration instance. This attribute is inherited by nested `server` elements.
`passwordfile`	none	(optional) File containing passwords. The password from this file is retrieved for communication with the application server administration instance. This attribute is inherited by nested `server` elements.
`host`	`localhost`	(optional) Target server. When deploying to a remote server, use the fully qualified host name. This attribute is inherited by nested `server` elements.
`port`	`4848`	(optional) The administration port on the target server. This attribute is inherited by nested `server` elements.
`target`	name of default instance	(optional) Target application server instance. This attribute is inherited by nested `server` elements.
`asinstalldir`	see description	(optional) The installation directory for the local Enterprise Server installation, which is used to find the administrative classes. If not specified, the command checks if the `asinstalldir` parameter has been set. Otherwise, administrative classes must be in the system classpath.

Examples of `sun-appserv-deploy`

Here is a simple web application deployment script with many implied attributes:

```
<sun-appserv-deploy
file="${assemble}/simpleapp.war"
passwordfile="${passwordfile}" />
```

Here is an equivalent script showing all the implied attributes:

```
<sun-appserv-deploy
 file="${assemble}/simpleapp.war"
 name="simpleapp"
 force="true"
 precompilejsp="false"
 verify="false"
 upload="true"
 user="admin"
 passwordfile="${passwordfile}"
 host="localhost"
 port="4848"
```

```
 target="${default-instance-name}"
 asinstalldir="${asinstalldir}" />
```

This example deploys multiple components to the same Enterprise Server instance running on a remote server:

```
<sun-appserv-deploy passwordfile="${passwordfile}" host="greg.sun.com"
         asinstalldir="/opt/sun" >
 <component file="${assemble}/simpleapp.ear"/>
 <component file="${assemble}/simpleservlet.war"
     contextroot="test"/>
 <component file="${assemble}/simplebean.jar"/>
</sun-appserv-deploy>
```

This example deploys multiple components to two Enterprise Server instances running on remote servers. In this example, both servers are using the same admin password. If this were not the case, each password could be specified in the server element.

```
<sun-appserv-deploy passwordfile="${passwordfile}" asinstalldir="/opt/sun" >
 <server host="greg.sun.com"/>
 <server host="joe.sun.com"/>
 <component file="${assemble}/simpleapp.ear"/>
 <component file="${assemble}/simpleservlet.war"
     contextroot="test"/>
 <component file="${assemble}/simplebean.jar"/>
</sun-appserv-deploy>
```

This example deploys the same components as the previous example because the three components match the `fileset` criteria, but note that it is not possible to set some component-specific attributes. All component-specific attributes (`name` and `contextroot`) use their default values.

```
<sun-appserv-deploy passwordfile="${passwordfile}" host="greg.sun.com"
         asinstalldir="/opt/sun" >
 <fileset dir="${assemble}" includes="**/*.?ar" />
</sun-appserv-deploy>
```

The `sun-appserv-undeploy` Task

Undeploys any of the following from a local or remote Enterprise Server instance.

- Enterprise application (EAR file)
- Web application (WAR file)
- Enterprise Java Bean (EJB-JAR file)
- Enterprise connector (RAR file)
- Application client

Subelements of `sun-appserv-undeploy`

The following table describes subelements for the `sun-appserv-undeploy` task. These are objects upon which this task acts.

TABLE 3–3 The `sun-appserv-undeploy` Subelements

Element	Description
"The `server` Subelement" on page 61	An Enterprise Server instance
"The `component` Subelement" on page 63	A component to be deployed
"The `fileset` Subelement" on page 65	A set of component files that match specified parameters

Attributes of `sun-appserv-undeploy`

The following table describes attributes for the `sun-appserv-undeploy` task.

TABLE 3–4 The `sun-appserv-undeploy` Attributes

Attribute	Default	Description
`name`	file name without extension	(optional if a `component` or `fileset` subelement is present or the `file` attribute is specified, otherwise required) The display name for the component being undeployed.
`file`	none	(optional) The component to undeploy. If this attribute refers to a file, it must be a valid archive. If this attribute refers to a directory, it must contain a valid archive in which all components have been exploded.
`droptables`	`sun-ejb-jar.xml` entry	(optional) If `true`, causes database tables that were automatically created when the bean(s) were last deployed to be dropped when the bean(s) are undeployed. If `false`, does not drop tables. If not specified, the value of the `drop-tables-at-undeploy` attribute in `sun-ejb-jar.xml` is used. For details, see "Generation Options" on page 124 and "Generation Options for CMP" on page 197.
`cascade`	`false`	(optional) If `true`, deletes all connection pools and connector resources associated with the resource adapter being undeployed. If `false`, undeployment fails if any pools or resources are still associated with the resource adapter. This attribute is applicable to connectors (resource adapters) and applications with connector modules.
`user`	`admin`	(optional) The user name used when logging into the application server administration instance. This attribute is inherited by nested `server` elements.
`passwordfile`	none	(optional) File containing passwords. The password from this file is retrieved for communication with the application server administration instance. This attribute is inherited by nested `server` elements.
`host`	`localhost`	(optional) Target server. When deploying to a remote server, use the fully qualified host name. This attribute is inherited by nested `server` elements.

TABLE 3–4 The `sun-appserv-undeploy` Attributes *(Continued)*

Attribute	Default	Description
`port`	`4848`	(optional) The administration port on the target server. This attribute is inherited by nested `server` elements.
`target`	name of default instance	(optional) Target application server instance. This attribute is inherited by nested `server` elements.
`asinstalldir`	see description	(optional) The installation directory for the local Enterprise Server installation, which is used to find the administrative classes. If not specified, the command checks to see if the `asinstalldir` parameter has been set. Otherwise, administrative classes must be in the system classpath.

Examples of `sun-appserv-undeploy`

Here is a simple application undeployment script with many implied attributes:

```
<sun-appserv-undeploy name="simpleapp" passwordfile="${passwordfile}" />
```

Here is an equivalent script showing all the implied attributes:

```
<sun-appserv-undeploy
 name="simpleapp"
 user="admin"
 passwordfile="${passwordfile}"
 host="localhost"
 port="4848"
 target="${default-instance-name}"
 asinstalldir="${asinstalldir}" />
```

This example demonstrates using the archive files (EAR and WAR, in this case) for the undeployment, using the component name (for undeploying the EJB component in this example), and undeploying multiple components.

```
<sun-appserv-undeploy passwordfile="${passwordfile}">
 <component file="${assemble}/simpleapp.ear"/>
 <component file="${assemble}/simpleservlet.war"/>
 <component name="simplebean" />
</sun-appserv-undeploy>
```

As with the deployment process, components can be undeployed from multiple servers in a single command. This example shows the same three components being removed from two different instances of the Enterprise Server. In this example, the passwords for both instances are the same.

```
<sun-appserv-undeploy passwordfile="${passwordfile}">
 <server host="greg.sun.com"/>
 <server host="joe.sun.com"/>
 <component file="${assemble}/simpleapp.ear"/>
 <component file="${assemble}/simpleservlet.war"/>
 <component name="simplebean" />
</sun-appserv-undeploy>
```

The `sun-appserv-instance` Task

Starts, stops, restarts, creates, or removes application server instances.

Subelements of `sun-appserv-instance`

The following table describes subelements for the `sun-appserv-instance` task. These are objects upon which this task acts.

TABLE 3–5 The `sun-appserv-instance` Subelements

Element	Description
"The `server` Subelement" on page 61	An Enterprise Server instance

Attributes of `sun-appserv-instance`

The following table describes attributes for the `sun-appserv-instance` task.

TABLE 3–6 The `sun-appserv-instance` Attributes

Attribute	Default	Description
`action`	none	The control command for the target application server. Valid values are `start`, `stop`, `create`, and `delete`. A `restart` sends the `stop` command followed by the `start` command. The `restart` command is not supported on Windows.
`debug`	`false`	(optional) Deprecated. If `action` is set to `start`, specifies whether the server starts in debug mode. This attribute is ignored for other values of `action`. If `true`, the instance generates additional debugging output throughout its lifetime. This attribute is inherited by nested `server` elements.
`config`	none	(optional, applicable only if `action` is `create`) The configuration for the new stand-alone instance. The configuration must exist and must not be `default-config` or an already referenced stand-alone configuration (including the administration server configuration `server-config`).
`property`	none	(optional, applicable only if `action` is `create`) Defines system properties for the server instance. These properties override port settings in the server instance's configuration. The following properties are defined: `http-listener-1-port`, `http-listener-2-port`, `orb-listener-1-port`, `SSL-port`, `SSL_MUTUALAUTH-port`, `JMX_SYSTEM_CONNECTOR_port`. System properties can be changed after instance creation using the system property commands. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.
`user`	`admin`	(optional) The user name used when logging into the application server administration instance. This attribute is inherited by nested `server` elements.
`passwordfile`	none	(optional) File containing passwords. The password from this file is retrieved for communication with the application server administration instance. This attribute is inherited by nested `server` elements.

TABLE 3–6 The `sun-appserv-instance` Attributes *(Continued)*

Attribute	Default	Description
`host`	`localhost`	(optional) Target server. If it is a remote server, use the fully qualified host name. This attribute is inherited by nested `server` elements.
`port`	`4848`	(optional) The administration port on the target server. This attribute is inherited by nested `server` elements.
`instance`	name of default instance	(optional) Target application server instance. This attribute is inherited by nested `server` elements.
`asinstalldir`	see description	(optional) The installation directory for the local Enterprise Server installation, which is used to find the administrative classes. If not specified, the command checks to see if the `asinstalldir` parameter has been set. Otherwise, administrative classes must be in the system classpath.

Examples of `sun-appserv-instance`

This example starts the local Enterprise Server instance:

```
<sun-appserv-instance action="start" passwordfile="${passwordfile}"
 instance="${default-instance-name}"/>
```

Here is an equivalent script showing all the implied attributes:

```
<sun-appserv-instance
 action="start"
 user="admin"
 passwordfile="${passwordfile}"
 host="localhost"
 port="4848"
 instance="${default-instance-name}"
 asinstalldir="${asinstalldir}" />
```

Multiple servers can be controlled using a single command. In this example, two servers are restarted, and in this case each server uses a different password:

```
<sun-appserv-instance action="restart"
     instance="${default-instance-name}"/>
<server host="greg.sun.com" passwordfile="${password.greg}"/>
 <server host="joe.sun.com" passwordfile="${password.joe}"/>
</sun-appserv-instance>
```

This example creates a new Enterprise Server instance:

```
<sun-appserv-instance
 action="create" instanceport="8080"
 passwordfile="${passwordfile}"
 instance="development" />
```

Here is an equivalent script showing all the implied attributes:

```
<sun-appserv-instance
 action="create"
 instanceport="8080"
 user="admin"
 passwordfile="${passwordfile}"
 host="localhost"
 port="4848"
 instance="development"
 asinstalldir="${asinstalldir}" />
```

Instances can be created on multiple servers using a single command. This example creates a new instance named qa on two different servers. In this case, both servers use the same password.

```
<sun-appserv-instance
     action="create"
     instanceport="8080"
     instance="qa"
     passwordfile="${passwordfile}>
 <server host="greg.sun.com"/>
 <server host="joe.sun.com"/>
</sun-appserv-instance>
```

These instances can also be removed from their respective servers:

```
<sun-appserv-instance
     action="delete"
     instance="qa"
     passwordfile="${passwordfile}>
 <server host="greg.sun.com"/>
 <server host="joe.sun.com"/>
</sun-appserv-instance>
```

Different instance names and instance ports can also be specified using attributes of the server subelement:

```
<sun-appserv-instance action="create" passwordfile="${passwordfile}>
 <server host="greg.sun.com" instanceport="8080" instance="qa"/>
 <server host="joe.sun.com" instanceport="9090"
     instance="integration-test"/>
</sun-appserv-instance>
```

The `sun-appserv-component` Task

Enables or disables the following Java EE component types that have been deployed to the Enterprise Server.

- Enterprise application (EAR file)
- Web application (WAR file)
- Enterprise Java Bean (EJB-JAR file)
- Enterprise connector (RAR file)
- Application client

You do not need to specify the archive to enable or disable a component: only the component name is required. You can use the component archive, however, because it implies the component name.

Subelements of `sun-appserv-component`

The following table describes subelements for the `sun-appserv-component` task. These are objects upon which this task acts.

TABLE 3–7 The `sun-appserv-component` Subelements

Element	Description
"The `server` Subelement" on page 61	An Enterprise Server instance
"The `component` Subelement" on page 63	A component to be deployed
"The `fileset` Subelement" on page 65	A set of component files that match specified parameters

Attributes of `sun-appserv-component`

The following table describes attributes for the `sun-appserv-component` task.

TABLE 3–8 The `sun-appserv-component` Attributes

Attribute	Default	Description
`action`	none	The control command for the target application server. Valid values are `enable` and `disable`.
`name`	file name without extension	(optional if a `component` or `fileset` subelement is present or the `file` attribute is specified, otherwise required) The display name for the component being enabled or disabled.
`file`	none	(optional) The component to enable or disable. If this attribute refers to a file, it must be a valid archive. If this attribute refers to a directory, it must contain a valid archive in which all components have been exploded.
`user`	`admin`	(optional) The user name used when logging into the application server administration instance. This attribute is inherited by nested `server` elements.
`passwordfile`	none	(optional) File containing passwords. The password from this file is retrieved for communication with the application server administration instance. This attribute is inherited by nested `server` elements.
`host`	`localhost`	(optional) Target server. When enabling or disabling a remote server, use the fully qualified host name. This attribute is inherited by nested `server` elements.
`port`	`4848`	(optional) The administration port on the target server. This attribute is inherited by nested `server` elements.

TABLE 3–8 The `sun-appserv-component` Attributes *(Continued)*

Attribute	Default	Description
`target`	name of default instance	(optional) Target application server instance. This attribute is inherited by nested `server` elements.
`asinstalldir`	see description	(optional) The installation directory for the local Enterprise Server installation, which is used to find the administrative classes. If not specified, the command checks to see if the `asinstalldir` parameter has been set. Otherwise, administrative classes must be in the system classpath.

Examples of `sun-appserv-component`

Here is a simple example of disabling a component:

```
<sun-appserv-component
 action="disable"
 name="simpleapp"
 passwordfile="${passwordfile}" />
```

Here is a simple example of enabling a component:

```
<sun-appserv-component
 action="enable"
 name="simpleapp"
 passwordfile="${passwordfile}" />
```

Here is an equivalent script showing all the implied attributes:

```
<sun-appserv-component
 action="enable"
 name="simpleapp"
 user="admin"
 passwordfile="${passwordfile}"
 host="localhost"
 port="4848"
 target="${default-instance-name}"
 asinstalldir="${asinstalldir}" />
```

This example demonstrates disabling multiple components using the archive files (EAR and WAR, in this case) and using the component name (for an EJB component in this example).

```
<sun-appserv-component action="disable" passwordfile="${passwordfile}">
 <component file="${assemble}/simpleapp.ear"/>
 <component file="${assemble}/simpleservlet.war"/>
 <component name="simplebean" />
</sun-appserv-component>
```

Components can be enabled or disabled on multiple servers in a single task. This example shows the same three components being enabled on two different instances of the Enterprise Server. In this example, the passwords for both instances are the same.

```
<sun-appserv-component action="enable" passwordfile="${passwordfile}">
 <server host="greg.sun.com"/>
 <server host="joe.sun.com"/>
 <component file="${assemble}/simpleapp.ear"/>
 <component file="${assemble}/simpleservlet.war"/>
 <component name="simplebean" />
</sun-appserv-component>
```

The `sun-appserv-admin` Task

Enables arbitrary administrative commands and scripts to be executed on the Enterprise Server. This is useful for cases where a specific Ant task has not been developed or a set of related commands are in a single script.

Subelements of `sun-appserv-admin`

The following table describes subelements for the `sun-appserv-admin` task. These are objects upon which this task acts.

TABLE 3–9 The `sun-appserv-admin` Subelements

Element	Description
"The `server` Subelement" on page 61	An Enterprise Server instance

Attributes of `sun-appserv-admin`

The following table describes attributes for the `sun-appserv-admin` task.

TABLE 3–10 The `sun-appserv-admin` Attributes

Attribute	Default	Description
`command`	none	(exactly one of these is required: `command` or `explicitcommand`) The command to execute. If the `user`, `passwordfile`, `host`, `port`, or `target` attributes are also specified, they are automatically inserted into the command before execution. If any of these options are specified in the command string, the corresponding attribute values are ignored.
`explicitcommand`	none	(exactly one of these is required: `command` or `explicitcommand`) The exact command to execute. No command processing is done, and all other attributes are ignored.
`user`	`admin`	(optional) The user name used when logging into the application server administration instance. This attribute is inherited by nested `server` elements.
`passwordfile`	none	(optional) File containing passwords. The password from this file is retrieved for communication with the application server administration instance. This attribute is inherited by nested `server` elements.
`host`	`localhost`	(optional) Target server. If it is a remote server, use the fully qualified host name. This attribute is inherited by nested `server` elements.

TABLE 3–10 The `sun-appserv-admin` Attributes *(Continued)*

Attribute	Default	Description
`port`	`4848`	(optional) The administration port on the target server. This attribute is inherited by nested `server` elements.
`asinstalldir`	see description	(optional) The installation directory for the local Enterprise Server installation, which is used to find the administrative classes. If not specified, the command checks if the `asinstalldir` parameter has been set. Otherwise, administrative classes must be in the system classpath.

Examples of `sun-appserv-admin`

Here is an example of executing the `create-jms-dest` command:

```
<sun-appserv-admin command="create-jms-dest --desttype topic">
```

Here is an example of using `explicitcommand` to execute the `create-jms-dest` command:

```
<sun-appserv-admin explicitcommand="create-jms-dest --desttype topic --target server1 simpleJmsDest">
```

The `sun-appserv-jspc` Task

Precompiles JSP source code into Enterprise Server compatible Java code for initial invocation by Enterprise Server. Use this task to speed up access to JSP files or to check the syntax of JSP source code. You can feed the resulting Java code to the `javac` task to generate class files for the JSP files.

Attributes of `sun-appserv-jspc`

The following table describes attributes for the `sun-appserv-jspc` task.

TABLE 3–11 The `sun-appserv-jspc` Attributes

Attribute	Default	Description
`destdir`	none	The destination directory for the generated Java source files.
`srcdir`	none	(exactly one of these is required: `srcdir` or `webapp`) The source directory where the JSP files are located.
`webapp`	none	(exactly one of these is required: `srcdir` or `webapp`) The directory containing the web application. All JSP files within the directory are recursively parsed. The base directory must have a `WEB-INF` subdirectory beneath it. When `webapp` is used, `sun-appserv-jspc` hands off all dependency checking to the compiler.
`verbose`	`2`	(optional) The verbosity integer to be passed to the compiler.
`classpath`	none	(optional) The classpath for running the JSP compiler.
`classpathref`	none	(optional) A reference to the JSP compiler classpath.

TABLE 3–11 The `sun-appserv-jspc` Attributes *(Continued)*

Attribute	Default	Description
`uribase`	`/`	(optional) The URI context of relative URI references in the JSP files. If this context does not exist, it is derived from the location of the JSP file relative to the declared or derived value of `uriroot`. Only pages translated from an explicitly declared JSP file are affected.
`uriroot`	see description	(optional) The root directory of the web application, against which URI files are resolved. If this directory is not specified, the first JSP file is used to derive it: each parent directory of the first JSP file is searched for a `WEB-INF` directory, and the directory closest to the JSP file that has one is used. If no `WEB-INF` directory is found, the directory from which `sun-appserv-jspc` was called is used. Only pages translated from an explicitly declared JSP file (including tag libraries) are affected.
`package`	none	(optional) The destination package for the generated Java classes.
`asinstalldir`	see description	(optional) The installation directory for the local Enterprise Server installation, which is used to find the administrative classes. If not specified, the command checks if the `asinstalldir` parameter has been set. Otherwise, administrative classes must be in the system classpath.

Example of `sun-appserv-jspc`

The following example uses the `webapp` attribute to generate Java source files from JSP files. The `sun-appserv-jspc` task is immediately followed by a `javac` task, which compiles the generated Java files into class files. The `classpath` value in the `javac` task must be all on one line with no spaces.

```
<sun-appserv-jspc
 destdir="${assemble.war}/generated"
 webapp="${assemble.war}"
 classpath="${assemble.war}/WEB-INF/classes"
 asinstalldir="${asinstalldir}" />
<javac
 srcdir="${assemble.war}/WEB-INF/generated"
 destdir="${assemble.war}/WEB-INF/generated"
 debug="on"
 classpath="${assemble.war}/WEB-INF/classes:${asinstalldir}/lib/
   appserv-rt.jar:${asinstalldir}/lib/appserv-ext.jar">
 <include name="**/*.java"/>
</javac>
```

The `sun-appserv-update` Task

Enables deployed web applications (EAR files) and modules (EJB JAR, RAR, and WAR files) to be updated and reloaded for fast iterative development. This task copies modified class files, XML files, and other contents of the archive files to the appropriate subdirectory of the *domain-dir*`/applications` directory, then touches the `.reload` file to cause dynamic reloading to occur.

This is a local task and must be executed on the same machine as the Enterprise Server.

For more information about dynamic reloading, see the *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

Attributes of `sun-appserv-update`

The following table describes attributes for the `sun-appserv-update` task.

TABLE 3–12 The `sun-appserv-update` Attributes

Attribute	Default	Description
`file`	none	The component to update, which must be a valid archive.
`domain`	`domain1`	(optional) The domain in which the application has been previously deployed.

Example of `sun-appserv-update`

The following example updates the Java EE application `foo.ear`, which is deployed to the default domain, `domain1`.

```
<sun-appserv-update file="foo.ear"/>
```

The `wsgen` Task

Generates JAX-WS portable artifacts used in JAX-WS web services. Reads a web service endpoint class and generates all the required artifacts for web service deployment and invocation.

Attributes of `wsgen`

The following table describes attributes for the `wsgen` task.

TABLE 3–13 The `wsgen` Attributes

Attribute	Default	Description
`sei`	none	Specifies the name of the service endpoint interface (SEI) class.
`destdir`	current directory	(optional) Specifies where to place the output generated classes.
`classpath`	system classpath	(optional) Specifies where to find the input class files. Same as `cp` attribute.
`cp`	system classpath	(optional) Specifies where to find the input class files. Same as `classpath` attribute.
`resourcedestdir`	current directory	(optional) Specifies where to place generated resource files such as WSDL files. Used only if the `genwsdl` attribute is set to `true`.

TABLE 3–13 The `wsgen` Attributes *(Continued)*

Attribute	Default	Description
`sourcedestdir`	current directory	(optional) Specifies where to place generated source files.
`keep`	`false`	(optional) If `true`, keeps generated files.
`verbose`	`false`	(optional) If `true`, outputs compiler messages.
`genwsdl`	`true`	(optional) If `true`, generates a WSDL file.
`protocol`	`soap1.1`	(optional) Specifies the protocol to use in the `wsdl:binding`. Used only if the `genwsdl` attribute is set to `true`. Allowed values are `soap1.1` or `Xsoap1.2`. `Xsoap1.2` is not standard and is only used if the extension attribute is set to `true`.
`servicename`	none	(optional) Specifies a particular `wsdl:service` name for the generated WSDL file. Used only if the `genwsdl` attribute is set to `true`. For example: `servicename="{http://mynamespace/}MyService"`
`portname`	none	(optional) Specifies a particular `wsdl:port` name for the generated WSDL. Used only if the `genwsdl` attribute is set to `true`. For example: `portname="{http://mynamespace/}MyPort"`
`extension`	`false`	(optional) If `true`, allows vendor extensions not in the specification. Use of extensions may result in applications that are not portable and may not interoperate with other implementations.

Example of `wsgen`

The following example generates portable artifacts for `fromjava.server.AddNumbersImpl`, uses `compile.classpath` as the classpath, and writes the WSDL file to `${wsdl.dir}`.

```
<wsgen
   resourcedestdir="${wsdl.dir}"
   sei="fromjava.server.AddNumbersImpl">
   <classpath refid="compile.classpath"/>
</wsgen>
```

The `wsimport` Task

Generates JAX-WS portable artifacts for a given WSDL file. Portable artifacts include service endpoint interfaces (SEIs), services, exception classes mapped from the `wsdl:fault` and `soap:headerfault` tags, asynchronous response beans derived from the `wsdl:message` tag, and JAXB generated value types. After generation, these artifacts can be packaged in a WAR file with the WSDL and schema documents along with the endpoint implementation and then deployed.

Attributes of `wsimport`

The following table describes attributes for the `wsimport` task.

TABLE 3–14 The `wsimport` Attributes

Attribute	Default	Description
`wsdl`	none	Specifies the name of the WSDL file.
`destdir`	current directory	(optional) Specifies where to place the output generated classes.
`sourcedestdir`	current directory	(optional) Specifies where to place generated source files. Used only if the `keep` attribute is set to `true`.
`keep`	`false`	(optional) If `true`, keeps generated files.
`verbose`	`false`	(optional) If `true`, outputs compiler messages.
`binding`	none	(optional) Specifies external JAX-WS or JAXB binding files. JAX-WS and JAXB binding files can customize things like package names and bean names. More information on JAX-WS and JAXB binding files can be found in the customization documentation included with this release.
`extension`	`false`	(optional) If `true`, allows vendor extensions not in the specification. Use of extensions may result in applications that are not portable and may not interoperate with other implementations.
`wsdllocation`	none	(optional) Specifies the value of `@WebService.wsdlLocation` and `@WebServiceClient.wsdlLocation` annotation elements for the generated SEI and Service interface. This should be set to the URI of the web service WSDL file.
`catalog`	none	(optional) Specifies the catalog file to resolve external entity references. Supported formats are TR9401, XCatalog, and OASIS XML Catalog. Additionally, the Ant `xmlcatalog` type can be used to resolve entities.
`package`	none	(optional) Specifies the target package, overriding any WSDL and schema binding customization for package name, and the default package name algorithm defined in the JAX-WS specification.

Example of `wsimport`

The following example generates client-side artifacts for `AddNumbers.wsdl` and stores `.class` files in the `${build.classes.home}` directory using the `custom.xml` customization file.

```
<wsimport
   destdir="${build.classes.home}"
   wsdl="AddNumbers.wsdl"
   binding="custom.xml">
</wsimport>
```

Reusable Subelements

Reusable subelements of the Ant tasks for the Enterprise Server are as follows. These are objects upon which the Ant tasks act.

The `server` Subelement

Specifies an Enterprise Server instance. Allows a single task to act on multiple server instances. The `server` attributes override corresponding attributes in the parent task; therefore, the parent task attributes function as default values.

Attributes of `server`

The following table describes attributes for the `server` element.

TABLE 3–15 The `server` Attributes

Attribute	Default	Description
`user`	`admin`	(optional) The user name used when logging into the Enterprise Server domain administration server (DAS).
`passwordfile`	none	(optional) File containing passwords. The password from this file is retrieved for communication with the Enterprise Server DAS.
`host`	`localhost`	(optional) Target server. When targeting a remote server, use the fully qualified host name.
`port`	`4848`	(optional) The administration port on the target server.
`instance`	name of default instance	(optional) Target application server instance.
`instanceport`	none	(applies to "The `sun-appserv-instance` Task" on page 50 only) Deprecated.
`debug`	`false`	(applies to "The `sun-appserv-instance` Task" on page 50 only, optional) Deprecated. If `action` is set to `start`, specifies whether the server starts in debug mode. This attribute is ignored for other values of `action`. If `true`, the instance generates additional debugging output throughout its lifetime.
`upload`	`true`	(applies to "The `sun-appserv-deploy` Task" on page 43 only, optional) If `true`, the component is transferred to the server for deployment. If the component is being deployed on the local machine, set upload to `false` to reduce deployment time.

TABLE 3–15 The `server` Attributes *(Continued)*

Attribute	Default	Description
`virtualservers`	default virtual server only	(applies to "The `sun-appserv-deploy` Task" on page 43 only, optional) A comma-separated list of virtual servers to be deployment targets. This attribute applies only to application (`.ear`) or web (`.war`) components and is ignored for other component types.

Examples of `server`

You can control multiple servers using a single task. In this example, two servers are started, each using a different password. Only the second server is started in debug mode.

```
<sun-appserv-instance action="start">
<server host="greg.sun.com" passwordfile="${password.greg}"/>
<server host="joe.sun.com" passwordfile="${password.joe}"
     debug="true"/>
</sun-appserv-instance>
```

You can create instances on multiple servers using a single task. This example creates a new instance named `qa` on two different servers. Both servers use the same password.

```
<sun-appserv-instance action="create" instanceport="8080"
     instance="qa" passwordfile="${passwordfile}>
<server host="greg.sun.com"/>
<server host="joe.sun.com"/>
</sun-appserv-instance>
```

These instances can also be removed from their respective servers:

```
<sun-appserv-instance action="delete" instance="qa"
     passwordfile="${passwordfile}>
<server host="greg.sun.com"/>
<server host="joe.sun.com"/>
</sun-appserv-instance>
```

You can specify different instance names and instance ports using attributes of the nested `server` element:

```
<sun-appserv-instance action="create" passwordfile="${passwordfile}>
<server host="greg.sun.com" instanceport="8080" instance="qa"/>
<server host="joe.sun.com" instanceport="9090"
    instance="integration-test"/>
</sun-appserv-instance>
```

You can deploy multiple components to multiple servers (see the "The `component` Subelement" on page 63) . This example deploys each component to two Enterprise Server instances running on remote servers. Both servers use the same password.

```
<sun-appserv-deploy passwordfile="${passwordfile}"
  asinstalldir="/opt/s1as8" >
<server host="greg.sun.com"/>
<server host="joe.sun.com"/>
```

```
<component file="${assemble}/simpleapp.ear"/>
<component file="${assemble}/simpleservlet.war"
     contextroot="test"/>
<component file="${assemble}/simplebean.jar"/>
</sun-appserv-deploy>
```

You can also undeploy multiple components from multiple servers. This example shows the same three components being removed from two different instances. Both servers use the same password.

```
<sun-appserv-undeploy passwordfile="${passwordfile}">
<server host="greg.sun.com"/>
<server host="joe.sun.com"/>
<component file="${assemble}/simpleapp.ear"/>
<component file="${assemble}/simpleservlet.war"/>
<component name="simplebean" />
</sun-appserv-undeploy>
```

You can enable or disable components on multiple servers. This example shows the same three components being enabled on two different instances. Both servers use the same password.

```
<sun-appserv-component action="enable" passwordfile="${passwordfile}">
<server host="greg.sun.com"/>
<server host="joe.sun.com"/>
<component file="${assemble}/simpleapp.ear"/>
<component file="${assemble}/simpleservlet.war"/>
<component name="simplebean" />
</sun-appserv-component>
```

The `component` Subelement

Specifies a Java EE component. Allows a single task to act on multiple components. The `component` attributes override corresponding attributes in the parent task; therefore, the parent task attributes function as default values.

Attributes of `component`

The following table describes attributes for the `component` element.

TABLE 3–16 The `component` Attributes

Attribute	Default	Description
`file`	none	(optional if the parent task is "The `sun-appserv-undeploy` Task" on page 47 or "The `sun-appserv-component` Task" on page 52) The target component. If this attribute refers to a file, it must be a valid archive. If this attribute refers to a directory, it must contain a valid archive in which all components have been exploded. If `upload` is set to `false`, this must be an absolute path on the server machine.

TABLE 3–16 The `component` Attributes *(Continued)*

Attribute	Default	Description
`name`	file name without extension	(optional) The display name for the component.
`force`	`true`	(applies to "The `sun-appserv-deploy` Task" on page 43 only, optional) If `true`, the component is overwritten if it already exists on the server. If `false`, the containing element's operation fails if the component exists.
`precompilejsp`	`false`	(applies to "The `sun-appserv-deploy` Task" on page 43 only, optional) If `true`, all JSP files found in an enterprise application (`.ear`) or web application (`.war`) are precompiled. This attribute is ignored for other component types.
`retrievestubs`	client stubs not saved	(applies to "The `sun-appserv-deploy` Task" on page 43 only, optional) The directory where client stubs are saved.
`contextroot`	file name without extension	(applies to "The `sun-appserv-deploy` Task" on page 43 only, optional) The context root for a web module (WAR file). This attribute is ignored if the component is not a WAR file.
`verify`	`false`	(applies to "The `sun-appserv-deploy` Task" on page 43 only, optional) If `true`, syntax and semantics for all deployment descriptors is automatically verified for correctness.

Examples of `component`

You can deploy multiple components using a single task. This example deploys each component to the same Enterprise Server instance running on a remote server.

```
<sun-appserv-deploy passwordfile="${passwordfile}" host="greg.sun.com"
     asinstalldir="/opt/s1as8" >
<component file="${assemble}/simpleapp.ear"/>
<component file="${assemble}/simpleservlet.war"
     contextroot="test"/>
<component file="${assemble}/simplebean.jar"/>
</sun-appserv-deploy>
```

You can also undeploy multiple components using a single task. This example demonstrates using the archive files (EAR and WAR, in this case) and the component name (for the EJB component).

```
<sun-appserv-undeploy passwordfile="${passwordfile}">
<component file="${assemble}/simpleapp.ear"/
<component file="${assemble}/simpleservlet.war"/>
<component name="simplebean" />
</sun-appserv-undeploy>
```

You can deploy multiple components to multiple servers. This example deploys each component to two instances running on remote servers. Both servers use the same password.

```
<sun-appserv-deploy passwordfile="${passwordfile}" asinstalldir="/opt/s1as8" >
<server host="greg.sun.com"/>
<server host="joe.sun.com"/>
<component file="${assemble}/simpleapp.ear"/>
<component file="${assemble}/simpleservlet.war"
    contextroot="test"/>
<component file="${assemble}/simplebean.jar"/>
</sun-appserv-deploy>
```

You can also undeploy multiple components to multiple servers. This example shows the same three components being removed from two different instances. Both servers use the same password.

```
<sun-appserv-undeploy passwordfile="${passwordfile}">
<server host="greg.sun.com"/>
<server host="joe.sun.com"/>
<component file="${assemble}/simpleapp.ear"/>
<component file="${assemble}/simpleservlet.war"/>
<component name="simplebean" />
</sun-appserv-undeploy>
```

You can enable or disable multiple components. This example demonstrates disabling multiple components using the archive files (EAR and WAR, in this case) and the component name (for the EJB component).

```
<sun-appserv-component action="disable" passwordfile="${passwordfile}">
<component file="${assemble}/simpleapp.ear"/>
<component file="${assemble}/simpleservlet.war"/>
<component name="simplebean" />
</sun-appserv-component>
```

You can enable or disable multiple components on multiple servers. This example shows the same three components being enabled on two different instances. Both servers use the same password.

```
<sun-appserv-component action="enable" passwordfile="${passwordfile}">
<server host="greg.sun.com"/>
<server host="joe.sun.com"/>
<component file="${assemble}/simpleapp.ear"/>
<component file="${assemble}/simpleservlet.war"/>
<component name="simplebean" />
</sun-appserv-component>
```

The `fileset` Subelement

Selects component files that match specified parameters. When `fileset` is included as a subelement, the `name` and `contextroot` attributes of the containing element must use their default values for each file in the `fileset`. For more information, see `http://ant.apache.org/manual/CoreTypes/fileset.html`.

◆ ◆ ◆ CHAPTER 4

Debugging Applications

This chapter gives guidelines for debugging applications in the Sun GlassFish Enterprise Server. It includes the following sections:

Enabling Debugging

When you enable debugging, you enable both local and remote debugging. To start the server in debug mode, use the `--debug` option as follows:

```
asadmin start-domain --debug [domain-name]
```

You can then attach to the server from the Java Debugger (`jdb`) at its default Java Platform Debugger Architecture (JPDA) port, which is 9009. For example, for UNIX systems:

```
jdb -attach 9009
```

For Windows:

```
jdb -connect com.sun.jdi.SocketAttach:port=9009
```

For more information about the `jdb` debugger, see the following links:

- Java Platform Debugger Architecture - The Java Debugger: `http://java.sun.com/products/jpda/doc/soljdb.html`

- Java Platform Debugger Architecture - Connecting with JDB: `http://java.sun.com/products/jpda/doc/conninv.html#JDB`

Enterprise Server debugging is based on the JPDA. For more information, see "JPDA Options" on page 68.

You can attach to the Enterprise Server using any JPDA compliant debugger, including that of NetBeans (`http://www.netbeans.org`), Sun Java Studio, JBuilder, Eclipse, and so on.

You can enable debugging even when the application server is started without the `--debug` option. This is useful if you start the application server from the Windows Start Menu, or if you want to make sure that debugging is always turned on.

▼ To Set the Server to Automatically Start Up in Debug Mode

1. **Use the Administration Console. Select the Enterprise Server component and the JVM Settings tab.**
2. **Check the Debug Enabled box.**
3. **To specify a different port (from 9009, the default) to use when attaching the JVM software to a debugger, specify** `address=`*port-number* **in the Debug Options field.**
4. **To add JPDA options, add any desired JPDA debugging options in Debug Options. See "JPDA Options" on page 68.**

See Also For details, click the Help button in the Administration Console from the JVM Settings page.

JPDA Options

The default JPDA options in Enterprise Server are as follows:

```
-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=9009
```

For Windows, you can change `dt_socket` to `dt_shmem`.

If you substitute `suspend=y`, the JVM software starts in suspended mode and stays suspended until a debugger attaches to it. This is helpful if you want to start debugging as soon as the JVM software starts.

To specify a different port (from 9009, the default) to use when attaching the JVM software to a debugger, specify `address=`*port-number*.

You can include additional options. A list of JPDA debugging options is available at `http://java.sun.com/products/jpda/doc/conninv.html#Invocation`.

Generating a Stack Trace for Debugging

To generate a Java stack trace for debugging, use the `asadmin generate-jvm-report --type=thread` command. The stack trace goes to the *domain-dir*`/logs/server.log` file and also appears on the command prompt screen. For more information about the `asadmin generate-jvm-report` command, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Application Client Debugging

When the `appclient` script executes the `java` command to run the Application Client Container (ACC), which in turn runs the client, it includes on the command line the value of the `VMARGS` environment variable. You can set this variable to any suitable value. For example:

```
VMARGS=-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=8118
```

The following example also works:

```
set VMARGS=-Xdebug -agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=8118
```

For debugging an application client, you should set suspend to `y` so you can connect the debugger to the client before any code has actually executed. Otherwise, the client may start running and execute past the point you want to examine.

You should use different ports for the server and client if you are debugging both concurrently. For details about setting the port, see "JPDA Options" on page 68.

You can also include JVM options such as `-Xdebug` and `-Xrunjwdp` in the `appclient` script directly. For information about the `appclient` script, see *Sun GlassFish Enterprise Server v3 Reference Manual*.

Note – The Application Client Container is supported only in the full Enterprise Server, not in the Web Profile. See Chapter 11, "Developing Java Clients."

Sun GlassFish Message Queue Debugging

Sun GlassFish Message Queue has a broker logger, which can be useful for debugging Java Message Service (JMS) applications, including message-driven bean applications. You can adjust the logger's verbosity, and you can send the logger output to the broker's console using the broker's `-tty` option. For more information, see the *Sun GlassFish Message Queue 4.4 Administration Guide*.

Note – JMS resources are supported only in the full Enterprise Server, not in the Web Profile. See Chapter 17, "Using the Java Message Service."

Enabling Verbose Mode

To have the server logs and messages printed to `System.out` on your command prompt screen, you can start the server in verbose mode. This makes it easy to do simple debugging using print statements, without having to view the `server.log` file every time.

To start the server in verbose mode, use the `--verbose` option as follows:

```
asadmin start-domain --verbose [domain-name]
```

When the server is in verbose mode, messages are logged to the console or terminal window in addition to the log file. In addition, pressing Ctrl-C stops the server and pressing Ctrl-\ (on UNIX platforms) or Ctrl-Break (on Windows platforms) prints a thread dump. On UNIX platforms, you can also print a thread dump using the `jstack` command (see `http://java.sun.com/javase/6/docs/technotes/tools/share/jstack.html`) or the command `kill -QUIT` *process_id*.

Enterprise Server Logging

You can use the Enterprise Server's log files to help debug your applications. Use the Administration Console. Select the Enterprise Server component. Then click the View Log Files button in the General Information page.

To change logging settings, select the Logging tab.

For details about logging, click the Help button in the Administration Console.

Profiling Tools

You can use a profiler to perform remote profiling on the Enterprise Server to discover bottlenecks in server-side performance. This section describes how to configure these profilers for use with the Enterprise Server:

Information about comprehensive monitoring and management support in the Java 2 Platform, Standard Edition (J2SE platform) is available at `http://java.sun.com/javase/6/docs/technotes/guides/management/index.html`.

The NetBeans Profiler

For information on how to use the NetBeans profiler, see `http://www.netbeans.org` and `http://blogs.sun.com/roller/page/bhavani?entry=analyzing_the_performance_of_java`.

The HPROF Profiler

The Heap and CPU Profiling Agent (HPROF) is a simple profiler agent shipped with the Java 2 SDK. It is a dynamically linked library that interacts with the Java Virtual Machine Profiler Interface (JVMPI) and writes out profiling information either to a file or to a socket in ASCII or binary format.

HPROF can monitor CPU usage, heap allocation statistics, and contention profiles. In addition, it can also report complete heap dumps and states of all the monitors and threads in the Java virtual machine. For more details on the HPROF profiler, see the technical article at `http://java.sun.com/developer/technicalArticles/Programming/HPROF.html`.

After HPROF is enabled using the following instructions, its libraries are loaded into the server process.

▼ To Use HPROF Profiling on UNIX

1. **Use the Administration Console. Select the Enterprise Server component and the JVM Settings tab. Then select the Profiler tab.**
2. **Edit the following fields:**
   - Profiler Name – `hprof`

- Profiler Enabled – `true`
- Classpath – (leave blank)
- Native Library Path – (leave blank)
- JVM Option – Select Add, type the HPROF JVM option in the Value field, then check its box. The syntax of the HPROF JVM option is as follows:

  ```
  -Xrunhprof[:help]|[:param=value,param2=value2, ...]
  ```

  Here is an example of *params* you can use:

  ```
  -Xrunhprof:file=log.txt,thread=y,depth=3
  ```

  The `file` parameter determines where the stack dump is written.

  Using `help` lists parameters that can be passed to HPROF. The output is as follows:

  ```
  Hprof usage: -Xrunhprof[:help]|[:<option>=<value>, ...]

  Option Name and Value   Description             Default
  ---------------------   -----------             -------
  heap=dump|sites|all     heap profiling          all
  cpu=samples|old         CPU usage               off
  format=a|b              ascii or binary output  a
  file=<file>             write data to file      java.hprof
                                 (.txt for ascii)
  net=<host>:<port>       send data over a socket write to file
  depth=<size>            stack trace depth       4
  cutoff=<value>          output cutoff point     0.0001
  lineno=y|n              line number in traces?  y
  thread=y|n              thread in traces?       n
  doe=y|n                 dump on exit?           y
  ```

  Note – Do not use `help` in the JVM Option field. This parameter prints text to the standard output and then exits.

  The help output refers to the parameters as options, but they are not the same thing as JVM options.

3 Restart the Enterprise Server.

This writes an HPROF stack dump to the file you specified using the `file` HPROF parameter.

The JProbe Profiler

Information about JProbe from Sitraka is available at `http://www.quest.com/jprobe/`.

After JProbe is installed using the following instructions, its libraries are loaded into the server process.

To Enable Remote Profiling With JProbe

1. **Install JProbe 3.0.1.1.**

   For details, see the JProbe documentation.

2. **Configure Enterprise Server using the Administration Console:**

   a. **Select the Enterprise Server component and the JVM Settings tab. Then select the Profiler tab.**

   b. **Edit the following fields before selecting Save and restarting the server:**

   - Profiler Name – `jprobe`
   - Profiler Enabled – `true`
   - Classpath – (leave blank)
   - Native Library Path – *JProbe-dir*`/profiler`
   - JVM Option – For each of these options, select Add, type the option in the Value field, then check its box

     `-Xbootclasspath/p:`*JProbe-dir*`/profiler/jpagent.jar`

     `-Xrunjprobeagent`

     `-Xnoclassgc`

   Note – If any of the configuration options are missing or incorrect, the profiler might experience problems that affect the performance of the Enterprise Server.

   When the server starts up with this configuration, you can attach the profiler.

3. **Set the following environment variable:**

   `JPROBE_ARGS_0=-jp_input=`*JPL-file-path*

   See Step 6 for instructions on how to create the JPL file.

4. **Start the server.**

5. **Launch the `jpprofiler` and attach to Remote Session. The default port is `4444`.**

6. **Create the JPL file using the JProbe Launch Pad. Here are the required settings:**

   a. **Select Server Side for the type of application.**

   b. **On the Program tab, provide the following details:**

   - Target Server – *other-server*

- Server home Directory – *as-install*
- Server class File – `com.sun.enterprise.server.J2EERunner`
- Working Directory – *as-install*
- Classpath – *as-install*`/lib/appserv-rt.jar`
- Source File Path – *source-code-dir* (in case you want to get the line level details)
- Server class arguments – (optional)
- Main Package – `com.sun.enterprise.server`

You must also set VM, Attach, and Coverage tabs appropriately. For further details, see the JProbe documentation. After you have created the JPL file, use this an input to `JPROBE_ARGS_0`.

PART II

Developing Applications and Application Components

◆ ◆ ◆ CHAPTER 5

Securing Applications

This chapter describes how to write secure Java EE applications, which contain components that perform user authentication and access authorization for the business logic of Java EE components.

For information about administrative security for the Sun GlassFish Enterprise Server, see Chapter 11, "Administering System Security," in *Sun GlassFish Enterprise Server v3 Administration Guide*.

For general information about Java EE security, see Part VII, "Security," in *The Java EE 6 Tutorial, Volume I*.

This chapter contains the following sections:

Note – The Web Profile of the Enterprise Server supports the EJB 3.1 Lite specification, which allows enterprise beans within web applications, among other features. The full Enterprise Server supports the entire EJB 3.1 specification. For details, see JSR 318 (`http://jcp.org/en/jsr/detail?id=318`).

Security Goals

In an enterprise computing environment, there are many security risks. The goal of the Enterprise Server is to provide highly secure, interoperable, and distributed component computing based on the Java EE security model. Security goals include:

- Full compliance with the Java EE security model. This includes EJB and servlet role-based authorization.
- Support for single sign-on across all Enterprise Server applications within a single security domain.
- Support for web services message security.
- Security support for application clients.
- Support for several underlying authentication realms, such as simple file and Lightweight Directory Access Protocol (LDAP). Certificate authentication is also supported for Secure Socket Layer (SSL) client authentication. For Solaris, OS platform authentication is supported in addition to these.
- Support for declarative security through Enterprise Server specific XML-based role mapping.
- Support for Java Authorization Contract for Containers (JACC) pluggable authorization as included in the Java EE specification and defined by Java Specification Request (JSR) 115 (`http://www.jcp.org/en/jsr/detail?id=115`).
- Support for Java Authentication Service Provider Interface for Containers as included in the Java EE specification and defined by JSR 196 (`http://www.jcp.org/en/jsr/detail?id=196`).
- Support for Web Services Interoperability Technologies (WSIT) as described in Metro Users Guide (`https://metro.dev.java.net/guide/`).

Enterprise Server Specific Security Features

The Enterprise Server supports the Java EE security model, as well as the following features which are specific to the Enterprise Server:

- Message security; see "Configuring Message Security for Web Services" on page 92
- Single sign-on across all Enterprise Server applications within a single security domain; see "User Authentication for Single Sign-on" on page 104
- Programmatic login; see "Programmatic Login" on page 102

Container Security

The component containers are responsible for providing Java EE application security. The container provides two security forms:

Annotations (also called metadata) enable a declarative style of programming, and so encompass both the declarative and programmatic security concepts. Users can specify information about security within a class file using annotations. When the application is deployed, this information can either be used by or overridden by the application or module deployment descriptor.

Declarative Security

Declarative security means that the security mechanism for an application is declared and handled externally to the application. Deployment descriptors describe the Java EE application's security structure, including security roles, access control, and authentication requirements.

The Enterprise Server supports the deployment descriptors specified by Java EE and has additional security elements included in its own deployment descriptors. Declarative security is the application deployer's responsibility. For more information about Sun-specific deployment descriptors, see the *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

There are two levels of declarative security, as follows:

Application Level Security

For an application, roles used by any application must be defined in `@DeclareRoles` annotations in the code or `role-name` elements in the application deployment descriptor (`application.xml`). Those role names are scoped to the EJB XML deployment descriptors (`ejb-jar.xml` and `sun-ejb-jar.xml` files) and to the servlet XML deployment descriptors (`web.xml` and `sun-web.xml` files). For an individually deployed web or EJB module, you define roles using `@DeclareRoles` annotations or `role-name` elements in the Java EE deployment descriptor files `web.xml` or `ejb-jar.xml`.

To map roles to principals and groups, define matching `security-role-mapping` elements in the `sun-application.xml`, `sun-ejb-jar.xml`, or `sun-web.xml` file for each `role-name` used by the application. For more information, see "Roles, Principals, and Principal to Role Mapping" on page 80.

Component Level Security

Component level security encompasses web components and EJB components.

A secure web container authenticates users and authorizes access to a servlet or JSP by using the security policy laid out in the servlet XML deployment descriptors (`web.xml` and `sun-web.xml` files).

The EJB container is responsible for authorizing access to a bean method by using the security policy laid out in the EJB XML deployment descriptors (`ejb-jar.xml` and `sun-ejb-jar.xml` files).

Programmatic Security

Programmatic security involves an EJB component or servlet using method calls to the security API, as specified by the Java EE security model, to make business logic decisions based on the caller or remote user's security role. Programmatic security should only be used when declarative security alone is insufficient to meet the application's security model.

The Java EE specification defines programmatic security as consisting of two methods of the EJB `EJBContext` interface and two methods of the servlet `HttpServletRequest` interface. The Enterprise Server supports these interfaces as specified in the specification.

For more information on programmatic security, see the following:

- The Java EE Specification
- "Programmatic Login" on page 102

Roles, Principals, and Principal to Role Mapping

For applications, you define roles in `@DeclareRoles` annotations or the Java EE deployment descriptor file `application.xml`. You define the corresponding role mappings in the Enterprise Server deployment descriptor file `sun-application.xml`. For individually deployed web or EJB modules, you define roles in `@DeclareRoles` annotations or the Java EE deployment descriptor files `web.xml` or `ejb-jar.xml`. You define the corresponding role mappings in the Enterprise Server deployment descriptor files `sun-web.xml` or `sun-ejb-jar.xml`.

For more information regarding Java EE deployment descriptors, see the Java EE Specification. For more information regarding Enterprise Server deployment descriptors, see Appendix C, "Elements of the Enterprise Server Deployment Descriptors," in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

Each `security-role-mapping` element in the `sun-application.xml`, `sun-web.xml`, or `sun-ejb-jar.xml` file maps a role name permitted by the application or module to principals and groups. For example, a `sun-web.xml` file for an individually deployed web module might contain the following:

```
<sun-web-app>
    <security-role-mapping>
        <role-name>manager</role-name>
        <principal-name>jgarcia</principal-name>
        <principal-name>mwebster</principal-name>
        <group-name>team-leads</group-name>
    </security-role-mapping>
    <security-role-mapping>
        <role-name>administrator</role-name>
        <principal-name>dsmith</principal-name>
    </security-role-mapping>
</sun-web-app>
```

A role can be mapped to either specific principals or to groups (or both). The principal or group names used must be valid principals or groups in the realm for the application or module. Note that the `role-name` in this example must match the `@DeclareRoles` annotations or the `role-name` in the `security-role` element of the corresponding `web.xml` file.

You can also specify a custom principal implementation class. This provides more flexibility in how principals can be assigned to roles. A user's JAAS login module now can authenticate its custom principal, and the authenticated custom principal can further participate in the Enterprise Server authorization process. For example:

```
<security-role-mapping>
    <role-name>administrator</role-name>
    <principal-name class-name="CustomPrincipalImplClass">
        dsmith
    </principal-name>
</security-role-mapping>
```

You can specify a default principal and a default principal to role mapping, each of which applies to the entire Enterprise Server. The default principal to role mapping maps group principals to the same named roles. Web modules that omit the `run-as` element in `web.xml` use the default principal. Applications and modules that omit the `security-role-mapping` element use the default principal to role mapping. These defaults are part of the Security Service, which you can access in the following ways:

- In the Administration Console, select the Security component under the relevant configuration. For details, click the Help button in the Administration Console.
- Use the `asadmin set` command. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*. For example, you can set the default principal as follows.

  ```
  asadmin set server-config.security-service.default-principal=dsmith
  asadmin set server-config.security-service.default-principal-password=secret
  ```

  You can set the default principal to role mapping as follows.

```
asadmin set server-config.security-service.activate-default-principal-to-role-mapping=true
asadmin set server-config.security-service.mapped-principal-class=CustomPrincipalImplClass
```

Realm Configuration

This section covers the following topics:

Supported Realms

The following realms are supported in the current release of the Enterprise Server:

- `file` – Stores user information in a file. This is the default realm when you first install the Enterprise Server.
- `ldap` – Stores user information in an LDAP directory.
- `jdbc` – Stores user information in a database.

  In the JDBC realm, the server gets user credentials from a database. The Enterprise Server uses the database information and the enabled JDBC realm option in the configuration file. For digest authentication, a JDBC realm should be created with `jdbcDigestRealm` as the JAAS context.
- `certificate` – Sets up the user identity in the Enterprise Server security context, and populates it with user data obtained from cryptographically verified client certificates.
- `solaris` – Allows authentication using Solaris `username+password` data. This realm is only supported on the Solaris operating system, version 9 and above.

For information about configuring realms, see "How to Configure a Realm" on page 82.

How to Configure a Realm

You can configure a realm in one of these ways:

- In the Administration Console, open the Security component under the relevant configuration and go to the Realms page. For details, click the Help button in the Administration Console.
- Use the `asadmin create-auth-realm` command to configure realms on local servers. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

How to Set a Realm for an Application or Module

The following deployment descriptor elements have optional `realm` or `realm-name` data subelements or attributes that override the domain's default realm:

- `sun-application` element in `sun-application.xml`
- `web-app` element in `web.xml`
- `as-context` element in `sun-ejb-jar.xml`
- `client-container` element in `sun-acc.xml`
- `client-credential` element in `sun-acc.xml`

If modules within an application specify realms, these are ignored. If present, the realm defined in `sun-application.xml` is used, otherwise the domain's default realm is used.

For example, a realm is specified in `sun-application.xml` as follows:

```
<sun-application>
    ...
    <realm>ldap</realm>
</sun-application>
```

For more information about the deployment descriptor files and elements, see Appendix C, "Elements of the Enterprise Server Deployment Descriptors," in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

Creating a Custom Realm

You can create a custom realm by providing a custom Java Authentication and Authorization Service (JAAS) login module class and a custom realm class. Note that client-side JAAS login modules are not suitable for use with the Enterprise Server.

To activate the custom login modules and realms, place the JAR files in the *domain-dir*`/lib` directory or the class files in the *domain-dir*`/lib/classes` directory. For more information about class loading in the Enterprise Server, see Chapter 2, "Class Loaders."

JAAS is a set of APIs that enable services to authenticate and enforce access controls upon users. JAAS provides a pluggable and extensible framework for programmatic user authentication and authorization. JAAS is a core API and an underlying technology for Java EE security mechanisms. For more information about JAAS, refer to the JAAS specification for Java SDK, available at `http://java.sun.com/products/jaas/`.

For general information about realms and login modules, see "Working with Realms, Users, Groups, and Roles" in *The Java EE 6 Tutorial, Volume I*.

For Javadoc tool pages relevant to custom realms, go to `https://glassfish.dev.java.net/nonav/docs/v3/api/` and click on the `com.sun.appserv.security` package.

Custom login modules must extend the `com.sun.appserv.security.AppservPasswordLoginModule` class. This class implements `javax.security.auth.spi.LoginModule`. Custom login modules must not implement `LoginModule` directly.

Custom login modules must provide an implementation for one abstract method defined in `AppservPasswordLoginModule`:

```
abstract protected void authenticateUser() throws LoginException
```

This method performs the actual authentication. The custom login module must not implement any of the other methods, such as `login()`, `logout()`, `abort()`, `commit()`, or `initialize()`. Default implementations are provided in `AppservPasswordLoginModule` which hook into the Enterprise Server infrastructure.

The custom login module can access the following protected object fields, which it inherits from `AppservPasswordLoginModule`. These contain the user name and password of the user to be authenticated:

```
protected String _username;
protected String _password;
```

The `authenticateUser()` method must end with the following sequence:

```
String[] grpList;
// populate grpList with the set of groups to which
// _username belongs in this realm, if any
commitUserAuthentication(_username, _password,
  _currentRealm, grpList);
```

Custom realms must extend the `com.sun.appserv.security.AppservRealm` class and implement the following methods:

```
public void init(Properties props) throws BadRealmException,
    NoSuchRealmException
```

This method is invoked during server startup when the realm is initially loaded. The `props` argument contains the properties defined for this realm. The realm can do any initialization it needs in this method. If the method returns without throwing an exception, the Enterprise Server assumes that the realm is ready to service authentication requests. If an exception is thrown, the realm is disabled.

```
public String getAuthType()
```

This method returns a descriptive string representing the type of authentication done by this realm.

```
public abstract Enumeration getGroupNames(String username) throws
    InvalidOperationException, NoSuchUserException
```

This method returns an `Enumeration` (of `String` objects) enumerating the groups (if any) to which the given `username` belongs in this realm.

Custom realms that manage users must implement the following additional methods:

```
public abstract boolean supportsUserManagement();
```

This method returns `true` if the realm supports user management.

```
public abstract Enumeration getGroupNames() throws BadRealmException;
```

This method returns an `Enumeration` of all group names.

```
public abstract Enumeration getUserNames() throws BadRealmException;
```

This method returns an `Enumeration` of all user names.

```
public abstract void refresh() throws BadRealmException;
```

This method refreshes the realm data so that new users and groups are visible.

```
public abstract void persist() throws BadRealmException;
```

This method persists the realm data to permanent storage.

```
public abstract User getUser(String name) throws NoSuchUserException,
BadRealmException;
```

This method returns the information recorded about a particular named user.

```
public abstract void addUser(String name, String password, String[] groupList) throws
BadRealmException, IASSecurityException;
```

This method adds a new user, who cannot already exist.

```
public abstract void removeUser(String name) throws NoSuchUserException,
BadRealmException;
```

This method removes a user, who must exist.

```
public abstract void updateUser(String name, String newName, String password,
String[] groups) throws NoSuchUserException, BadRealmException, IASSecurityException;
```

This method updates data for a user, who must exist.

Note – The array passed to the `commitUseAuthentication` method should be newly created and otherwise unreferenced. This is because the group name array elements are set to null after authentication as part of cleanup. So the second time your custom realm executes it returns an array with null elements.

Ideally, your custom realm should not return member variables from the `authenticate` method. It should return local variables as the default `JDBCRealm` does. Your custom realm can create a local `String` array in its `authenticate` method, copy the values from the member variables, and return the `String` array. Or it can use `clone` on the member variables.

JACC Support

JACC (Java Authorization Contract for Containers) is part of the Java EE specification and defined by JSR 115 (`http://www.jcp.org/en/jsr/detail?id=115`). JACC defines an interface for pluggable authorization providers. Specifically, JACC is used to plug in the Java policy provider used by the container to perform Java EE caller access decisions. The Java policy provider performs Java policy decisions during application execution. This provides third parties with a mechanism to develop and plug in modules that are responsible for answering authorization decisions during Java EE application execution. The interfaces and rules used for developing JACC providers are defined in the JACC 1.0 specification.

The Enterprise Server provides a simple file-based JACC-compliant authorization engine as a default JACC provider, named `default`. An alternate provider named `simple` is also provided. To configure an alternate provider using the Administration Console, open the Security component under the relevant configuration, and select the JACC Providers component. For details, click the Help button in the Administration Console.

Pluggable Audit Module Support

Audit modules collect and store information on incoming requests (servlets, EJB components) and outgoing responses. You can create a custom audit module. This section covers the following topics:

For additional information about audit modules, see Audit Callbacks (`http://developers.sun.com/prodtech/appserver/reference/techart/ws_mgmt3.html#8.2`).

Configuring an Audit Module

To configure an audit module, you can perform one of the following tasks:

- To specify an audit module using the Administration Console, open the Security component under the relevant configuration, and select the Audit Modules component. For details, click the Help button in the Administration Console.
- You can use the `asadmin create-audit-module` command to configure an audit module. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

The `AuditModule` Class

You can create a custom audit module by implementing a class that extends `com.sun.enterprise.security.audit.AuditModule`.

For Javadoc tool pages relevant to audit modules, go to `https://glassfish.dev.java.net/nonav/docs/v3/api/` and click on the `com.sun.enterprise.security.audit` package.

The `AuditModule` class provides default "no-op" implementations for each of the following methods, which your custom class can override.

```
public void init(Properties props)
```

The preceding method is invoked during server startup when the audit module is initially loaded. The `props` argument contains the properties defined for this module. The module can do any initialization it needs in this method. If the method returns without throwing an exception, the Enterprise Server assumes the module realm is ready to service audit requests. If an exception is thrown, the module is disabled.

```
public void authentication(String user, String realm, boolean success)
```

This method is invoked when an authentication request has been processed by a realm for the given user. The `success` flag indicates whether the authorization was granted or denied.

```
public void webInvocation(String user, HttpServletRequest req, String type, boolean success)
```

This method is invoked when a web container call has been processed by authorization. The `success` flag indicates whether the authorization was granted or denied. The `req` object is the standard `HttpServletRequest` object for this request. The `type` string is one of `hasUserDataPermission` or `hasResourcePermission` (see JSR 115 (`http://www.jcp.org/en/jsr/detail?id=115`)).

```
public void ejbInvocation(String user, String ejb, String method, boolean success)
```

This method is invoked when an EJB container call has been processed by authorization. The `success` flag indicates whether the authorization was granted or denied. The `ejb` and `method` strings describe the EJB component and its method that is being invoked.

```
public void webServiceInvocation(String uri, String endpoint, boolean success)
```

This method is invoked during validation of a web service request in which the endpoint is a servlet. The `uri` is the URL representation of the web service endpoint. The `endpoint` is the name of the endpoint representation. The `success` flag indicates whether the authorization was granted or denied.

```
public void ejbAsWebServiceInvocation(String endpoint, boolean success)
```

This method is invoked during validation of a web service request in which the endpoint is a stateless session bean. The `endpoint` is the name of the endpoint representation. The `success` flag indicates whether the authorization was granted or denied.

The `server.policy` File

Each Enterprise Server domain has its own global J2SE policy file, located in *domain-dir*`/config`. The file is named `server.policy`.

The Enterprise Server is a Java EE compliant application server. As such, it follows the requirements of the Java EE specification, including the presence of the security manager (the Java component that enforces the policy) and a limited permission set for Java EE application code.

This section covers the following topics:

Default Permissions

Internal server code is granted all permissions. These are covered by the `AllPermission` grant blocks to various parts of the server infrastructure code. Do not modify these entries.

Application permissions are granted in the default grant block. These permissions apply to all code not part of the internal server code listed previously. The Enterprise Server does not distinguish between EJB and web module permissions. All code is granted the minimal set of web component permissions (which is a superset of the EJB minimal set). Do not modify these entries.

A few permissions above the minimal set are also granted in the default `server.policy` file. These are necessary due to various internal dependencies of the server implementation. Java EE application developers must not rely on these additional permissions. In some cases, deleting these permissions might be appropriate. For example, one additional permission is granted

specifically for using connectors. If connectors are not used in a particular domain, you should remove this permission, because it is not otherwise necessary.

System Properties

The following predefined system properties, also called variables, are available for use in the `server.policy` file. The system property most frequently used in `server.policy` is `${com.sun.aas.instanceRoot}`. For more information about system properties, see the `asadmin create-system-properties` command in the *Sun GlassFish Enterprise Server v3 Reference Manual*.

TABLE 5–1 Predefined System Properties

Property	Default	Description
`com.sun.aas.installRoot`	depends on operating system	Specifies the directory where the Enterprise Server is installed.
`com.sun.aas.instanceRoot`	depends on operating system	Specifies the top level directory for a server instance.
`com.sun.aas.hostName`	none	Specifies the name of the host (machine).
`com.sun.aas.javaRoot`	depends on operating system	Specifies the installation directory for the Java runtime.
`com.sun.aas.imqLib`	depends on operating system	Specifies the library directory for the Sun GlassFish Message Queue software.
`com.sun.aas.configName`	`server-config`	Specifies the name of the configuration used by a server instance.
`com.sun.aas.instanceName`	`server1`	Specifies the name of the server instance. This property is not used in the default configuration, but can be used to customize configuration.
`com.sun.aas.domainName`	`domain1`	Specifies the name of the domain. This property is not used in the default configuration, but can be used to customize configuration.

Changing Permissions for an Application

The default policy for each domain limits the permissions of Java EE deployed applications to the minimal set of permissions required for these applications to operate correctly. Do not add extra permissions to the default set (the grant block with no codebase, which applies to all code). Instead, add a new grant block with a codebase specific to the applications requiring the extra permissions, and only add the minimally necessary permissions in that block.

If you develop multiple applications that require more than this default set of permissions, you can add the custom permissions that your applications need. The `com.sun.aas.instanceRoot` variable refers to the *domain-dir*. For example:

```
grant codeBase "file:${com.sun.aas.instanceRoot}/applications/-" {
...
}
```

You can add permissions to stub code with the following grant block:

```
grant codeBase "file:${com.sun.aas.instanceRoot}/generated/-" {
...
}
```

In general, you should add extra permissions only to the applications or modules that require them, not to all applications deployed to a domain. For example:

```
grant codeBase "file:${com.sun.aas.instanceRoot}/applications/MyApp/-" {
...
}
```

For a module:

```
grant codeBase "file:${com.sun.aas.instanceRoot}/applications/MyModule/-" {
...
}
```

Note – Deployment directories may change between Enterprise Server releases.

An alternative way to add permissions to a specific application or module is to edit the `granted.policy` file for that application or module. The `granted.policy` file is located in the *domain-dir*`/generated/policy/`*app-or-module-name* directory. In this case, you add permissions to the default grant block. Do not delete permissions from this file.

When the application server policy subsystem determines that a permission should not be granted, it logs a `server.policy` message specifying the permission that was not granted and the protection domains, with indicated code source and principals that failed the protection check. For example, here is the first part of a typical message:

```
[#|2005-12-17T16:16:32.671-0200|INFO|sun-appserver-pe9.1|
javax.enterprise.system.core.security|_ThreadID=14;_ThreadName=Thread-31;|
JACC Policy Provider: PolicyWrapper.implies, context(null)-
permission((java.util.PropertyPermission java.security.manager write))
domain that failed(ProtectionDomain
(file:/E:/glassfish/domains/domain1/applications/cejug-clfds/ ... )
...
```

Granting the following permission eliminates the message:

```
grant codeBase "file:${com.sun.aas.instanceRoot}/applications/cejug-clfds/-" {
    permission java.util.PropertyPermission "java.security.manager", "write";
}
```

Note – Do not add `java.security.AllPermission` to the `server.policy` file for application code. Doing so completely defeats the purpose of the security manager, yet you still get the performance overhead associated with it.

As noted in the Java EE specification, an application should provide documentation of the additional permissions it needs. If an application requires extra permissions but does not document the set it needs, contact the application author for details.

As a last resort, you can iteratively determine the permission set an application needs by observing `AccessControlException` occurrences in the server log.

If this is not sufficient, you can add the `-Djava.security.debug=failure` JVM option to the domain. Use the following `asadmin create-jvm-options` command, then restart the server:

```
asadmin create-jvm-options -Djava.security.debug=failure
```

For more information about the `asadmin create-jvm-options` command, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

You can use the J2SE standard `policytool` or any text editor to edit the `server.policy` file. For more information, see `http://java.sun.com/docs/books/tutorial/security/tour2/index.html`.

For detailed information about policy file syntax, see `http://java.sun.com/javase/6/docs/technotes/guides/security/PolicyFiles.html#FileSyntax`.

For information about using system properties in the `server.policy` file, see `http://java.sun.com/javase/6/docs/technotes/guides/security/PolicyFiles.html#PropertyExp`.

For detailed information about the permissions you can set in the `server.policy` file, see `http://java.sun.com/javase/6/docs/technotes/guides/security/permissions.html`.

The Javadoc for the `Permission` class is at `http://java.sun.com/javase/6/docs/api/java/security/Permission.html`.

Enabling and Disabling the Security Manager

The security manager is disabled by default.

In a production environment, you may be able to safely disable the security manager if all of the following are true:

- Performance is critical
- Deployment to the production server is carefully controlled

- Only trusted applications are deployed
- Applications don't need policy enforcement

Disabling the security manager may improve performance significantly for some types of applications. To disable the security manager, do one of the following:

- To use the Administration Console, open the Security component under the relevant configuration, and uncheck the Security Manager Enabled box. Then restart the server. For details, click the Help button in the Administration Console.
- Use the following `asadmin delete-jvm-options` command, then restart the server:

  ```
  asadmin delete-jvm-options -Djava.security.manager
  ```

  To re-enable the security manager, use the corresponding `create-jvm-options` command. For more information about the `create-jvm-options` and `asadmin delete-jvm-options` commands, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Configuring Message Security for Web Services

In *message security*, security information is applied at the message layer and travels along with the web services message. Web Services Security (WSS) is the use of XML Encryption and XML Digital Signatures to secure messages. WSS profiles the use of various security tokens including X.509 certificates, Security Assertion Markup Language (SAML) assertions, and username/password tokens to achieve this.

Message layer security differs from transport layer security in that it can be used to decouple message protection from message transport so that messages remain protected after transmission, regardless of how many hops they travel.

Note – Message security (JSR 196) is supported only in the full Enterprise Server, not in the Web Profile.

Note – In this release of the Enterprise Server, message layer annotations are not supported.

For more information about web services, see Chapter 6, "Developing Web Services."

For more information about message security, see the following:

- Chapter 23, "Introduction to Security in the Java EE Platform," in *The Java EE 6 Tutorial, Volume I*
- Chapter 13, "Administering Message Security," in *Sun GlassFish Enterprise Server v3 Administration Guide*

- JSR 196 (`http://www.jcp.org/en/jsr/detail?id=196`), Java Authentication Service Provider Interface for Containers
- The Liberty Alliance Project specifications at `http://www.projectliberty.org/resources/specifications.php`
- The Oasis Web Services Security (WSS) specification at `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf`
- The Web Services Interoperability Organization (WS-I) Basic Security Profile (BSP) specification at `http://www.ws-i.org/Profiles/BasicSecurityProfile-1.0.html`
- The XML and Web Services Security page at `https://xwss.dev.java.net/`
- The WSIT page at `https://wsit.dev.java.net/`

The following web services security topics are discussed in this section:

Message Security Providers

When you first install the Enterprise Server, the providers `XWS_ClientProvider` and `XWS_ServerProvider` are configured but disabled. You can enable them in one of the following ways:

- To enable the message security providers using the Administration Console, open the Security component under the relevant configuration, select the Message Security component, and select SOAP. Then select `XWS_ServerProvider` from the Default Provider list and `XWS_ClientProvider` from the Default Client Provider list. For details, click the Help button in the Administration Console.
- You can enable the message security providers using the following commands.

```
asadmin set
server-config.security-service.message-security-config.SOAP.default_provider=XWS_ServerProvider
asadmin set
server-config.security-service.message-security-config.SOAP.default_client_provider=XWS_ClientProvider
```

For more information about the `asadmin set` command, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

The example described in "Understanding and Running the Sample Application" on page 99 uses the `ClientProvider` and `ServerProvider` providers, which are enabled when the Ant targets are run. You don't need to enable these on the Enterprise Server prior to running the example.

If you install the Access Manager, you have these additional provider choices:

- `AMClientProvider` and `AMServerProvider` – These providers secure web services and Simple Object Access Protocol (SOAP) messages using either WS-I BSP or Liberty ID-WSF tokens. These providers are used automatically if they are configured as the default providers. If you wish to override any provider settings, you can configure these providers in `message-security-binding` elements in the `sun-web.xml`, `sun-ejb-jar.xml`, and `sun-application-client.xml` deployment descriptor files.
- `AMHttpProvider` – This provider handles the initial end user authentication for securing web services using Liberty ID-WSF tokens and redirects requests to the Access Manager for single sign-on. To use this provider, specify it in the `httpservlet-security-provider` attribute of the `sun-web-app` element in the `sun-web.xml` file.

Liberty specifications can be viewed at `http://www.projectliberty.org/resources/specifications.php`. The WS-I BSP specification can be viewed at `http://www.ws-i.org/Profiles/BasicSecurityProfile-1.0.html`.

For more information about the Sun-specific deployment descriptor files, see the *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

For information about configuring these providers in the Enterprise Server, see Chapter 13, "Administering Message Security," in *Sun GlassFish Enterprise Server v3 Administration Guide*. For additional information about overriding provider settings, see "Application-Specific Message Protection" on page 96.

You can create new message security providers in one of the following ways:

- To create a message security provider using the Administration Console, open the Security component under the relevant configuration, and select the Message Security component. For details, click the Help button in the Administration Console.
- You can use the `asadmin create-message-security-provider` command to create a message security provider. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

In addition, you can set a few optional provider properties using the `asadmin set` command. For example:

```
asadmin set server-config.security-service.message-security-config.provider-config.property.debug=true
```

The following table describes these message security provider properties.

TABLE 5–2 Message Security Provider Properties

Property	Default	Description
`security.config`	*domain-dir*`/config/wss-server-config-1.0.xml`	Specifies the location of the message security configuration file. To point to a configuration file in the *domain-dir*`/config` directory, use the system property `${com.sun.aas.instanceRoot}/config/`, for example: `${com.sun.aas.instanceRoot}/config/wss-server-config-1.0.xml` See "System Properties" on page 89.
`debug`	`false`	If `true`, enables dumping of server provider debug messages to the server log.
`dynamic.username.password`	`false`	If `true`, signals the provider runtime to collect the user name and password from the `CallbackHandler` for each request. If `false`, the user name and password for `wsse:UsernameToken(s)` is collected once, during module initialization. This property is only applicable for a `ClientAuthModule`.
`encryption.key.alias`	`s1as`	Specifies the encryption key used by the provider. The key is identified by its `keystore` alias.
`signature.key.alias`	`s1as`	Specifies the signature key used by the provider. The key is identified by its `keystore` alias.

Message Security Responsibilities

In the Enterprise Server, the system administrator and application deployer roles are expected to take primary responsibility for configuring message security. In some situations, the application developer may also contribute, although in the typical case either of the other roles may secure an existing application without changing its implementation and without involving the developer. The responsibilities of the various roles are defined in the following sections:

- "Application Developer" on page 95
- "Application Deployer" on page 96
- "System Administrator" on page 96

Application Developer

The application developer can turn on message security, but is not responsible for doing so. Message security can be set up by the system administrator so that all web services are secured, or set up by the application deployer when the provider or protection policy bound to the application must be different from that bound to the container.

The application developer is responsible for the following:

- Determining if an application-specific message protection policy is required by the application. If so, ensuring that the required policy is specified at application assembly which may be accomplished by communicating with the application deployer.

- Determining if message security is necessary at the Enterprise Server level. If so, ensuring that this need is communicated to the system administrator, or taking care of implementing message security at the Enterprise Server level.

Application Deployer

The application deployer is responsible for the following:

- Specifying (at application assembly) any required application-specific message protection policies if such policies have not already been specified by upstream roles (the developer or assembler)
- Modifying Sun-specific deployment descriptors to specify application-specific message protection policies information (message-security-binding elements) to web service endpoint and service references

These security tasks are discussed in "Application-Specific Message Protection" on page 96. A sample application using message security is discussed in "Understanding and Running the Sample Application" on page 99.

System Administrator

The system administrator is responsible for the following:

- Configuring message security providers on the Enterprise Server.
- Managing user databases.
- Managing keystore and truststore files.
- Installing the sample. This is only done if the `xms` sample application is used to demonstrate the use of message layer web services security.

A system administrator uses the Administration Console to manage server security settings and uses a command line tool to manage certificate databases. Certificates and private keys are stored in key stores and are managed with `keytool`. System administrator tasks are discussed in Chapter 13, "Administering Message Security," in *Sun GlassFish Enterprise Server v3 Administration Guide*.

Application-Specific Message Protection

When the Enterprise Server provided configuration is insufficient for your security needs, and you want to override the default protection, you can apply *application-specific message security* to a web service.

Application-specific security is implemented by adding the message security binding to the web service endpoint, whether it is an EJB or servlet web service endpoint. Modify Sun-specific XML files to add the message binding information.

Message security can also be specified using a WSIT security policy in the WSDL file. For details, see the WSIT page at `https://wsit.dev.java.net/`.

For more information about message security providers, see "Message Security Providers" on page 93.

For more details on message security binding for EJB web services, servlet web services, and clients, see the XML file descriptions in Appendix C, "Elements of the Enterprise Server Deployment Descriptors," in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

- For `sun-ejb-jar.xml`, see "The sun-ejb-jar.xml File" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.
- For `sun-web.xml`, see "The sun-web.xml File" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.
- For `sun-application-client.xml`, see "The sun-application-client.xml file" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

This section contains the following topics:

Using a Signature to Enable Message Protection for All Methods

To enable message protection for all methods using digital signature, update the `message-security-binding` element for the EJB web service endpoint in the application's `sun-ejb-jar.xml` file. In this file, add `request-protection` and `response-protection` elements, which are analogous to the `request-policy` and `response-policy` elements discussed in Chapter 13, "Administering Message Security," in *Sun GlassFish Enterprise Server v3 Administration Guide*. To apply the same protection mechanisms for all methods, leave the method-name element blank. "Configuring Message Protection for a Specific Method Based on Digital Signatures" on page 98 discusses listing specific methods or using wildcard characters.

This section uses the sample application discussed in "Understanding and Running the Sample Application" on page 99 to apply application-level message security to show only the differences necessary for protecting web services using various mechanisms.

▼ To Enable Message Protection for All Methods Using Digital Signature

1 **In a text editor, open the application's `sun-ejb-jar.xml` file.**

For the `xms` example, this file is located in the directory *app-dir*`/xms-ejb/src/conf`, where *app-dir* is defined in "To Set Up the Sample Application" on page 100.

2 **Modify the `sun-ejb-jar.xml` file by adding the `message-security-binding` element as shown:**

```
<sun-ejb-jar>
  <enterprise-beans>
    <unique-id>1</unique-id>
    <ejb>
      <ejb-name>HelloWorld</ejb-name>
      <jndi-name>HelloWorld</jndi-name>
      <webservice-endpoint>
        <port-component-name>HelloIF</port-component-name>
        <endpoint-address-uri>service/HelloWorld</endpoint-address-uri>
        <message-security-binding auth-layer="SOAP">
          <message-security>
            <request-protection auth-source="content" />
            <response-protection auth-source="content"/>
          </message-security>
        </message-security-binding>
      </webservice-endpoint>
    </ejb>
  </enterprise-beans>
</sun-ejb-jar>
```

3 **Compile, deploy, and run the application as described in "To Run the Sample Application" on page 101.**

Configuring Message Protection for a Specific Method Based on Digital Signatures

To enable message protection for a specific method, or for a set of methods that can be identified using a wildcard value, follow these steps. As in the example discussed in "Using a Signature to Enable Message Protection for All Methods" on page 97, to enable message protection for a specific method, update the `message-security-binding` element for the EJB web service endpoint in the application's `sun-ejb-jar.xml` file. To this file, add `request-protection` and `response-protection` elements, which are analogous to the `request-policy` and `response-policy` elements discussed in Chapter 13, "Administering Message Security," in *Sun GlassFish Enterprise Server v3 Administration Guide*. The administration guide includes a table listing the set and order of security operations for different request and response policy configurations.

This section uses the sample application discussed in "Understanding and Running the Sample Application" on page 99 to apply application-level message security to show only the differences necessary for protecting web services using various mechanisms.

▼ To Enable Message Protection for a Particular Method or Set of Methods Using Digital Signature

1 **In a text editor, open the application's `sun-ejb-jar.xml` file.**

For the `xms` example, this file is located in the directory *app-dir*`/xms-ejb/src/conf`, where *app-dir* is defined in "To Set Up the Sample Application" on page 100.

2 **Modify the `sun-ejb-jar.xml` file by adding the `message-security-binding` element as shown:**

```
<sun-ejb-jar>
  <enterprise-beans>
  <unique-id>1</unique-id>
    <ejb>
      <ejb-name>HelloWorld</ejb-name>
      <jndi-name>HelloWorld</jndi-name>
      <webservice-endpoint>
        <port-component-name>HelloIF</port-component-name>
        <endpoint-address-uri>service/HelloWorld</endpoint-address-uri>
        <message-security-binding auth-layer="SOAP">
          <message-security>
            <message>
              <java-method>
                <method-name>ejbCreate</method-name>
              </java-method>
            </message>
            <message>
              <java-method>
                <method-name>sayHello</method-name>
              </java-method>
            </message>
            <request-protection auth-source="content" />
            <response-protection auth-source="content"/>
          </message-security>
        </message-security-binding>
      </webservice-endpoint>
    </ejb>
  </enterprise-beans>
</sun-ejb-jar>
```

3 **Compile, deploy, and run the application as described in "To Run the Sample Application" on page 101.**

Understanding and Running the Sample Application

This section discusses the WSS sample application. This sample application is installed on your system only if you installed the J2EE 1.4 samples. If you have not installed these samples, see "To Set Up the Sample Application" on page 100.

The objective of this sample application is to demonstrate how a web service can be secured with WSS. The web service in the `xms` example is a simple web service implemented using a Java EE EJB endpoint and a web service endpoint implemented using a servlet. In this example, a service endpoint interface is defined with one operation, `sayHello`, which takes a string then sends a response with `Hello` prefixed to the given string. You can view the WSDL file for the service endpoint interface at *app-dir*`/xms-ejb/src/conf/HelloWorld.wsdl`, where *app-dir* is defined in "To Set Up the Sample Application" on page 100.

In this application, the client looks up the service using the JNDI name `java:comp/env/service/HelloWorld` and gets the port information using a static stub to invoke the operation using a given name. For the name Duke, the client gets the response `Hello Duke!`

This example shows how to use message security for web services at the Enterprise Server level. For information about using message security at the application level, see "Application-Specific Message Protection" on page 96. The WSS message security mechanisms implement message-level authentication (for example, XML digital signature and encryption) of SOAP web services invocations using the X.509 and username/password profiles of the OASIS WS-Security standard, which can be viewed from the following URL: `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf`.

This section includes the following topics:

▼ To Set Up the Sample Application

Before You Begin To have access to this sample application, you must have previously installed the J2EE 1.4 samples. If the samples are not installed, follow the steps in the following section.

After you follow these steps, the sample application is located in the directory *as-install*`/j2ee14-samples/samples/webservices/security/ejb/apps/xms/` or in a directory of your choice. For easy reference throughout the rest of this section, this directory is referred to as simply *app-dir*.

1. **Go to the J2EE 1.4 download URL (`http://java.sun.com/j2ee/1.4/download.html`) in your browser.**
2. **Click on the Download button for the Samples Bundle.**
3. **Click on Accept License Agreement.**
4. **Click on the J2EE SDK Samples link.**
5. **Choose a location for the `j2eesdk-1_4_03-samples.zip` file.**

   Saving the file to *as-install* is recommended.
6. **Unzip the file.**

   Unzipping to the *as-install*`/j2ee14–samples` directory is recommended. For example, you can use the following command.

   ```
   unzip j2eesdk-1_4_03-samples.zip -d j2ee14-samples
   ```

▼ To Run the Sample Application

1. **Make sure that the Enterprise Server is running.**

   Message security providers are set up when the Ant targets are run, so you do not need to configure these on the Enterprise Server prior to running this example.

2. **If you are not running HTTP on the default port of 8080, change the WSDL file for the example to reflect the change, and change the `common.properties` file to reflect the change as well.**

   The WSDL file for this example is located at *app-dir*`/xms-ejb/src/conf/HelloWorld.wsdl`. The port number is in the following section:

   ```
   <service name="HelloWorld">
     <port name="HelloIFPort" binding="tns:HelloIFBinding">
       <soap:address location="http://localhost:8080/service/HelloWorld"/>
     </port>
   </service>
   ```

   Verify that the properties in the *as-install*`/samples/common.properties` file are set properly for your installation and environment. If you need a more detailed description of this file, refer to the "Configuration" section for the web services security applications at *as-install*`/j2ee14–samples/samples/webservices/security/docs/common.html#Logging`.

3. **Change to the *app-dir* directory.**

4. **Run the following Ant targets to compile, deploy, and run the example application:**

   a. **To compile samples:**

      ```
      ant
      ```

   b. **To deploy samples:**

      ```
      ant deploy
      ```

   c. **To run samples:**

      ```
      ant run
      ```

   If the sample has compiled and deployed properly, you see the following response on your screen after the application has run:

   ```
   run:[echo] Running the xms program:[exec] Established message level security :
   Hello Duke!
   ```

5. **To undeploy the sample, run the following Ant target:**

   ```
   ant undeploy
   ```

   All of the web services security examples use the same web service name (`HelloWorld`) and web service ports. These examples show only the differences necessary for protecting web services using various mechanisms. Make sure to undeploy an application when you have completed

running it. If you do not, you receive an `Already in Use` error and deployment failures when you try to deploy another web services example application.

Programmatic Login

Programmatic login allows a deployed Java EE application or module to invoke a login method. If the login is successful, a `SecurityContext` is established as if the client had authenticated using any of the conventional Java EE mechanisms. Programmatic login is supported for servlet and EJB components on the server side, and for stand-alone or application clients on the client side. Programmatic login is useful for an application having special needs that cannot be accommodated by any of the Java EE standard authentication mechanisms.

Note – Programmatic login is specific to the Enterprise Server and not portable to other application servers.

This section contains the following topics:

Programmatic Login Precautions

The Enterprise Server is not involved in how the login information (`user`, `password`) is obtained by the deployed application. Programmatic login places the burden on the application developer with respect to assuring that the resulting system meets security requirements. If the application code reads the authentication information across the network, the application determines whether to trust the user.

Programmatic login allows the application developer to bypass the application server-supported authentication mechanisms and feed authentication data directly to the security service. While flexible, this capability should not be used without some understanding of security issues.

Since this mechanism bypasses the container-managed authentication process and sequence, the application developer must be very careful in making sure that authentication is established before accessing any restricted resources or methods. It is also the application developer's responsibility to verify the status of the login attempt and to alter the behavior of the application accordingly.

The programmatic login state does not necessarily persist in sessions or participate in single sign-on.

Lazy authentication is not supported for programmatic login. If an access check is reached and the deployed application has not properly authenticated using the programmatic login method, access is denied immediately and the application might fail if not coded to account for this occurrence. One way to account for this occurrence is to catch the access control or security exception, perform a programmatic login, and repeat the request.

Granting Programmatic Login Permission

The `ProgrammaticLoginPermission` permission is required to invoke the programmatic login mechanism for an application if the security manager is enabled. For information about the security manager, see "The `server.policy` File" on page 88. This permission is not granted by default to deployed applications because this is not a standard Java EE mechanism.

To grant the required permission to the application, add the following to the *domain-dir*`/config/server.policy` file:

```
grant codeBase "file:jar-file-path" {
     permission com.sun.appserv.security.ProgrammaticLoginPermission
     "login";
 };
```

The *jar-file-path* is the path to the application's JAR file.

The `ProgrammaticLogin` Class

The `com.sun.appserv.security.ProgrammaticLogin` class enables a user to perform login programmatically.

For Javadoc tool pages relevant to programmatic login, go to `https://glassfish.dev.java.net/nonav/docs/v3/api/` and click on the `com.sun.appserv.security` package.

The `ProgrammaticLogin` class has four `login` methods, two for servlets or JSP files and two for EJB components.

The login methods for servlets or JSP files have the following signatures:

```
public java.lang.Boolean login(String user, String password,
    javax.servlet.http.HttpServletRequest request,
    javax.servlet.http.HttpServletResponse response)

public java.lang.Boolean login(String user, String password,
    String realm, javax.servlet.http.HttpServletRequest request,
    javax.servlet.http.HttpServletResponse response, boolean errors)
    throws java.lang.Exception
```

The login methods for EJB components have the following signatures:

```
public java.lang.Boolean login(String user, String password)

public java.lang.Boolean login(String user, String password,
    String realm, boolean errors) throws java.lang.Exception
```

All of these `login` methods accomplish the following:

- Perform the authentication
- Return `true` if login succeeded, `false` if login failed

The login occurs on the `realm` specified unless it is null, in which case the domain's default realm is used. The methods with no `realm` parameter use the domain's default realm.

If the `errors` flag is set to `true`, any exceptions encountered during the login are propagated to the caller. If set to `false`, exceptions are thrown.

On the client side, `realm` and `errors` parameters are ignored and the actual login does not occur until a resource requiring a login is accessed. A `java.rmi.AccessException` with `COBRA NO_PERMISSION` occurs if the actual login fails.

The logout methods for servlets or JSP files have the following signatures:

```
public java.lang.Boolean logout(HttpServletRequest request,
    HttpServletResponse response)

public java.lang.Boolean logout(HttpServletRequest request,
    HttpServletResponse response, boolean errors)
    throws java.lang.Exception
```

The logout methods for EJB components have the following signatures:

```
public java.lang.Boolean logout()

public java.lang.Boolean logout(boolean errors)
    throws java.lang.Exception
```

All of these `logout` methods return `true` if logout succeeded, `false` if logout failed.

If the `errors` flag is set to `true`, any exceptions encountered during the logout are propagated to the caller. If set to `false`, exceptions are thrown.

User Authentication for Single Sign-on

The single sign-on feature of the Enterprise Server allows multiple web applications deployed to the same virtual server to share the user authentication state. With single sign-on enabled, users who log in to one web application become implicitly logged into other web applications on the same virtual server that require the same authentication information. Otherwise, users would have to log in separately to each web application whose protected resources they tried to access.

A sample application using the single sign-on scenario could be a consolidated airline booking service that searches all airlines and provides links to different airline web sites. After the user signs on to the consolidated booking service, the user information can be used by each individual airline site without requiring another sign-on.

Single sign-on operates according to the following rules:

- Single sign-on applies to web applications configured for the same realm and virtual server. The realm is defined by the `realm-name` element in the `web.xml` file. For information about virtual servers, see Chapter 16, "Administering Internet Connectivity," in *Sun GlassFish Enterprise Server v3 Administration Guide*.
- As long as users access only unprotected resources in any of the web applications on a virtual server, they are not challenged to authenticate themselves.
- As soon as a user accesses a protected resource in any web application associated with a virtual server, the user is challenged to authenticate himself or herself, using the login method defined for the web application currently being accessed.
- After authentication, the roles associated with this user are used for access control decisions across all associated web applications, without challenging the user to authenticate to each application individually.
- When the user logs out of one web application (for example, by invalidating the corresponding session), the user's sessions in all web applications are invalidated. Any subsequent attempt to access a protected resource in any application requires the user to authenticate again.

The single sign-on feature utilizes HTTP cookies to transmit a token that associates each request with the saved user identity, so it can only be used in client environments that support cookies.

To configure single sign-on, set the following virtual server properties:

- `sso-enabled` - If `false`, single sign-on is disabled for this virtual server, and users must authenticate separately to every application on the virtual server. The default is `false`.
- `sso-max-inactive-seconds` - Specifies the time after which a user's single sign-on record becomes eligible for purging if no client activity is received. Since single sign-on applies across several applications on the same virtual server, access to any of the applications keeps the single sign-on record active. The default value is 5 minutes (`300` seconds). Higher values provide longer single sign-on persistence for the users at the expense of more memory use on the server.
- `sso-reap-interval-seconds` - Specifies the interval between purges of expired single sign-on records. The default value is `60`.

Here are example `asadmin set` commands with default values:

```
asadmin set server-config.http-service.virtual-server.vsrv1.property.sso-enabled="true"
asadmin set server-config.http-service.virtual-server.vsrv1.property.sso-max-inactive-seconds="300"
```

```
asadmin set server-config.http-service.virtual-server.vsrv1.property.sso-reap-interval-seconds="60"
```

For more information about the `asadmin set` command, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Adding Authentication Mechanisms to the Servlet Container

You can use JSR 196 in the web tier to facilitate the injection of pluggable authentication modules within the servlet constraint processing engine. The Enterprise Server includes implementations of a number of HTTP layer authentication mechanisms such as basic, form, and digest authentication. You can add alternative implementations of the included mechanisms or implementations of new mechanisms such as HTTP Negotiate/SPNEGO, OpenID, or CAS. JSR 196 server authentication modules are described in the following sections:

The Enterprise Server and JSR 196

The Enterprise Server implements the Servlet Container Profile of JSR 196, Java Authentication Service Provider Interface for Containers. JSR 196 defines a standard *service provider interface* (SPI) that extends the concepts of the Java Authentication and Authorization Service (JAAS) to enable pluggability of message authentication modules in message processing runtimes. The JSR 196 standard defines profiles that establish contracts for the use of the SPI in specific contexts. The Servlet Container Profile of JSR 196 defines the use of the SPI by a Servlet container such that:

- The resulting container can be configured with new authentication mechanisms.
- The container employs the configured mechanisms in its enforcement of the declarative servlet security model (declared in a `web.xml` file using `security-constraint` elements).

The JSR 196 specification defines a simple message processing model composed of four interaction points:

1. `secureRequest` on the client
2. `validateRequest` on the server
3. `secureResponse` on the server
4. `validateResponse` on the client

A message processing runtime uses the SPI at these interaction points to delegate the corresponding message security processing to authentication providers, also called *authentication modules*, integrated into the runtime by way of the SPI.

A compatible server-side message processing runtime, such as the Enterprise Server servlet container, supports the `validateRequest` and `secureResponse` interaction points of the message processing model. The servlet container uses the SPI at these interaction points to delegate the corresponding message security processing to a *server authentication module* (SAM), integrated by the SPI into the container.

Writing a Server Authentication Module

A key step in adding an authentication mechanism to a compatible server-side message processing runtime such as the Enterprise Server servlet container is acquiring a SAM that implements the desired authentication mechanism. One way to do that is to write the SAM yourself.

A SAM implements the `javax.security.auth.message.module.ServerAuthModule` interface as defined by JSR 196. A SAM is invoked indirectly by the message processing runtime at the `validateRequest` and `secureResponse` interaction points. A SAM must implement the five methods of the `ServerAuthModule` interface:

- `getSupportedMessageTypes()` — An array of `Class` objects where each element defines a message type supported by the SAM. For a SAM to be compatible with the Servlet Container Profile, the returned array must include the `HttpServletRequest.class` and `HttpServletResponse.class` objects.
- `initialize(MessagePolicy requestPolicy, MessagePolicy responsePolicy, CallbackHandler Map options)` — The container calls this method to provide the SAM with configuration values and with a `CallbackHandler`. The configuration values are returned in the policy arguments and in the options `Map`. The SAM uses `CallbackHandler` to access services, such as password validation, provided by the container.
- `AuthStatus validateRequest(MessageInfo messageInfo, Subject clientSubject, Subject serviceSubject)` — The container calls this method to process each received `HttpServletRequest`. The request and its associated `HttpServletResponse` are passed by the container to the SAM in the `messageInfo` argument. The SAM processes the request and may establish the response to be returned by the container. The SAM uses the provided `Subject` arguments to convey its authentication results. The SAM returns different status values to control the container's invocation processing. The status values and the circumstances under which they are returned are as follows:
  - `AuthStatus.SUCCESS` is returned when the application request message is successfully validated. The container responds to this status value by using the returned client `Subject` to invoke the target of the request. When this value is returned, the SAM

(provided a custom `AuthConfigProvider` is not being used) must use its `CallbackHandler` to handle a `CallerPrincipalCallback` using the `clientSubject` as an argument to the callback.

- `AuthStatus.SEND_CONTINUE` indicates that message validation is incomplete and that the SAM has established a preliminary response as the response message in `messageInfo`. The container responds to this status value by sending the response to the client.
- `AuthStatus.SEND_FAILURE` indicates that message validation failed and that the SAM has established an appropriate failure response message in `messageInfo`. The container responds to this status value by sending the response to the client.
- `AuthStatus.SEND_SUCCESS` is not typically returned. This status value indicates the end of a multi-message security dialog originating after the service interaction and during the processing of the application response. The container responds to this status value by sending the response to the client.

The `validateRequest` method may also throw an `AuthException` to indicate that the message processing by the SAM failed without establishing a failure response message in `messageInfo`.

- `secureResponse(MessageInfo messageInfo, Subject serviceSubject)` — The container calls this method before sending a response, resulting from an application invocation, to the client. The response is passed to the SAM in the `messageInfo` argument. In most cases, this method should just return the `SEND_SUCCESS` status.
- `cleanSubject(MessageInfo messageInfo, Subject subject)` — This method removes the mechanism-specific principals, credentials, or both from the subject. This method is not currently called by the container. A legitimate implementation could remove all the principals from the argument subject.

See the *Servlet Container Profile* section in the JSR 196 specification for additional background and details.

Sample Server Authentication Module

The class `MySam.java` is a sample SAM implementation. Notice that the sample implements the five methods of the `ServerAuthModule` interface. This SAM implements an approximation of HTTP basic authentication.

```
package tip.sam;

   import java.io.IOException;
   import java.util.Map;
   import javax.security.auth.Subject;
   import javax.security.auth.callback.Callback;
   import javax.security.auth.callback.CallbackHandler;
   import javax.security.auth.callback.UnsupportedCallbackException;
```

```
import javax.security.auth.message.AuthException;
import javax.security.auth.message.AuthStatus;
import javax.security.auth.message.MessageInfo;
import javax.security.auth.message.MessagePolicy;
import javax.security.auth.message.callback.CallerPrincipalCallback;
import javax.security.auth.message.callback.GroupPrincipalCallback;
import javax.security.auth.message.callback.PasswordValidationCallback;
import javax.security.auth.message.module.ServerAuthModule;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.catalina.util.Base64;

public class MySam implements ServerAuthModule {

   protected static final Class[]
     supportedMessageTypes = new Class[]{
       HttpServletRequest.class,
       HttpServletResponse.class
   };

   private MessagePolicy requestPolicy;
   private MessagePolicy responsePolicy;
   private CallbackHandler handler;
   private Map options;
   private String realmName = null;
   private String defaultGroup[] = null;
   privte static final String REALM_PROPERTY_NAME =
       "realm.name";
   private static final String GROUP_PROPERTY_NAME =
       "group.name";
   private static final String BASIC = "Basic";
   static final String AUTHORIZATION_HEADER =
       "authorization";
   static final String AUTHENTICATION_HEADER =
       "WWW-Authenticate";

   public void initialize(MessagePolicy reqPolicy,
           MessagePolicy resPolicy,
           CallbackHandler cBH, Map opts)
           throws AuthException {
       requestPolicy = reqPolicy;
       responsePolicy = resPolicy;
       handler = cBH;
       options = opts;
       if (options != null) {
           realmName = (String)
               options.get(REALM_PROPERTY_NAME);
           if (options.containsKey(GROUP_PROPERTY_NAME)) {
               defaultGroup = new String[]{(String)
                   options.get(GROUP_PROPERTY_NAME)};
           }
       }
   }

   public Class[] getSupportedMessageTypes() {
       return supportedMessageTypes;
   }

   public AuthStatus validateRequest(
```

```
            MessageInfo msgInfo, Subject client,
            Subject server) throws AuthException {
        try {

            String username =
                processAuthorizationToken(msgInfo, client);
            if (username ==
                null && requestPolicy.isMandatory()) {
                return sendAuthenticateChallenge(msgInfo);
            }

           setAuthenticationResult(
               username, client, msgInfo);
           return AuthStatus.SUCCESS;

        } catch (Exception e) {
            AuthException ae = new AuthException();
            ae.initCause(e);
            throw ae;
        }
    }

    private String processAuthorizationToken(
            MessageInfo msgInfo, Subject s)
            throws AuthException {

        HttpServletRequest request =
                (HttpServletRequest)
                msgInfo.getRequestMessage();

        String token =
                request.getHeader(AUTHORIZATION_HEADER);

        if (token != null && token.startsWith(BASIC + " ")) {

            token = token.substring(6).trim();

            // Decode and parse the authorization token
            String decoded =
                new String(Base64.decode(token.getBytes()));

            int colon = decoded.indexOf(':');
            if (colon <= 0 || colon == decoded.length() - 1) {
                return (null);
            }

            String username = decoded.substring(0, colon);

           // use the callback to ask the container to
           // validate the password
          PasswordValidationCallback pVC =
                  new PasswordValidationCallback(s, username,
                  decoded.substring(colon + 1).toCharArray());
          try {
              handler.handle(new Callback[]{pVC});
              pVC.clearPassword();
          } catch (Exception e) {
              AuthException ae = new AuthException();
              ae.initCause(e);
```

```
                throw ae;
            }

            if (pVC.getResult()) {
                return username;
            }
       }
       return null;
    }

    private AuthStatus sendAuthenticateChallenge(
            MessageInfo msgInfo) {

        String realm = realmName;
          // if the realm property is set use it,
          // otherwise use the name of the server
          // as the realm name.
          if (realm == null) {

           HttpServletRequest request =
                   (HttpServletRequest)
                   msgInfo.getRequestMessage();

           realm = request.getServerName();
         }

        HttpServletResponse response =
                (HttpServletResponse)
                msgInfo.getResponseMessage();

        String header = BASIC + " realm=\"" + realm + "\"";
        response.setHeader(AUTHENTICATION_HEADER, header);
        response.setStatus(
                HttpServletResponse.SC_UNAUTHORIZED);
        return AuthStatus.SEND_CONTINUE;
    }

    public AuthStatus secureResponse(
            MessageInfo msgInfo, Subject service)
            throws AuthException {
        return AuthStatus.SEND_SUCCESS;
    }

    public void cleanSubject(MessageInfo msgInfo,
            Subject subject)
            throws AuthException {
       if (subject != null) {
           subject.getPrincipals().clear();
       }
    }

    private static final String AUTH_TYPE_INFO_KEY =
            "javax.servlet.http.authType";

    // distinguish the caller principal
    // and assign default groups
    private void setAuthenticationResult(String name,
            Subject s, MessageInfo m)
            throws IOException,
```

```
            UnsupportedCallbackException {
        handler.handle(new Callback[]{
            new CallerPrincipalCallback(s, name)
        });
        if (name != null) {
          // add the default group if the property is set
            if (defaultGroup != null) {
                handler.handle(new Callback[]{
                    new GroupPrincipalCallback(s, defaultGroup)
                });
            }
            m.getMap().put(AUTH_TYPE_INFO_KEY, ""MySAM");
        }
    }
  }
```

Note that the `initialize` method looks for the `group.name` and `realm.name` properties. The `group.name` property configures the default group assigned as a result of any successful authentication. The `realm.name` property defines the realm value sent back to the browser in the `WWW-Authenticate` challenge.

Compiling and Installing a Server Authentication Module

Before you can use the sample SAM, you need to compile, install, and configure it. Then you can bind it to an application.

To compile the SAM, include the SPI in your classpath. When the Enterprise Server is installed, the JAR file containing the SPI, `jmac-api.jar`, is installed in the *as-install*`/lib` directory. After you compile the SAM, install it by copying a JAR file containing the compiled SAM to the *as-install*`/lib` directory.

Configuring a Server Authentication Module

You can configure a SAM in one of these ways:

- In the Administration Console, open the Security component under the relevant configuration and go to the Message Security page. Set the following options:
  - Authentication Layer — `HttpServlet`
  - Provider Type — `server` or `client-server`
  - Provider ID — Specify a unique name for the SAM, for example `MySAM`
  - Class Name — Specify the fully qualified class name, for example `tip.sam.MySam`
  - Additional Property — Name: `group-name` Value: `user`
  - Additional Property — Name: `realm-name` Value: `Sam`

  For details, click the Help button in the Administration Console.

- Use the `asadmin create-message-security-provider` command to configure a SAM. Set the following options:
  - `--layer HttpServlet`
  - `--providertype server` or `--providertype client-server`
  - `--classname tip.sam.MySam`
  - `--property group-name=user:realm-name=Sam`
  - Provider name operand — Specify a unique name for the SAM, for example `MySAM`

  For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Binding a Server Authentication Module to Your Application

After you install and configure the SAM, you can bind it for use by the container on behalf of one or more of your applications. You have two options in how you bind the SAM, depending on whether you are willing to repackage and redeploy your application:

- If you are willing to repackage and redeploy, you can bind the SAM using the `sun-web.xml` file. Set the value of the `httpservlet-security-provider` attribute of the `sun-web-app` element to the SAM's configured provider ID, for example, `MySAM`. For more information about the `sun-web.xml` file, see the *Sun GlassFish Enterprise Server v3 Application Deployment Guide*. This option leverages the native `AuthConfigProvider` implementation that ships with the Enterprise Server.
- Another approach is to develop your own `AuthConfigProvider` and register it with the Enterprise Server `AuthConfigFactory` for use on behalf of your applications. For example, a simple `AuthConfigProvider` can obtain, through its initialization properties, the classname of a SAM to configure on behalf of the applications for which the provider is registered. You can find a description of the functionality of an `AuthConfigProvider` and of the registration facilities provided by an `AuthConfigFactory` in the JSR 196 specification.

◆ ◆ ◆ CHAPTER 6

Developing Web Services

This chapter describes Sun GlassFish Enterprise Server support for web services. Java API for XML-Based Web Services (JAX-WS) version 2.2 is supported. Java API for XML-Based Remote Procedure Calls (JAX-RPC) version 1.1 is supported for backward compatibility. This chapter contains the following sections:

Note – If you installed the Web Profile, web services are not supported unless the optional Metro Web Services Stack add-on component is downloaded from the Update Tool. Without the Metro add-on component, a servlet or EJB component cannot be a web service endpoint, and the `sun-web.xml` and `sun-ejb-jar.xml` elements related to web services are ignored. For information about the Update Tool, see "Update Tool" in *Sun GlassFish Enterprise Server v3 Administration Guide*.

Part III, "Web Services," in *The Java EE 6 Tutorial, Volume I* shows how to deploy simple web services to the Enterprise Server.

For additional information about JAX-WS and web services, see Java Specification Request (JSR) 224 (`http://jcp.org/aboutJava/communityprocess/pfd/jsr224/index.html`) and JSR 109 (`http://jcp.org/en/jsr/detail?id=109`).

For information about web services security, see "Configuring Message Security for Web Services" on page 92.

The Fast Infoset standard specifies a binary format based on the XML Information Set. This format is an efficient alternative to XML. For information about using Fast Infoset, see the following links:

- Java Web Services Developer Pack 1.6 Release Notes (`http://java.sun.com/webservices/docs/1.6/ReleaseNotes.html`)
- Fast Infoset in Java Web Services Developer Pack, Version 1.6 (`http://java.sun.com/webservices/docs/1.6/jaxrpc/fastinfoset/manual.html`)
- Fast Infoset Project (`http://fi.dev.java.net`)

Creating Portable Web Service Artifacts

For a tutorial that shows how to use the `wsimport` and `wsgen` commands, see Part III, "Web Services," in *The Java EE 6 Tutorial, Volume I*. For reference information on these commands, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Deploying a Web Service

You deploy a web service endpoint to the Enterprise Server just as you would any servlet, stateless session bean (SLSB), or application.

Note – For complex services with dependent classes, user specified WSDL files, or other advanced features, autodeployment of an annotated file is not sufficient.

The Sun-specific deployment descriptor files `sun-web.xml` and `sun-ejb-jar.xml` provide optional web service enhancements in the `webservice-endpoint` and `webservice-description` elements, including a `debugging-enabled` subelement that enables the creation of a test page. The test page feature is enabled by default and described in "The Web Service URI, WSDL File, and Test Page" on page 116.

For more information about deployment, autodeployment, and deployment descriptors, see the *Sun GlassFish Enterprise Server v3 Application Deployment Guide*. For more information about the `asadmin deploy` command, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

The Web Service URI, WSDL File, and Test Page

Clients can run a deployed web service by accessing its service endpoint address URI, which has the following format:

`http://`*host*`:`*port*`/`*context-root*`/`*servlet-mapping-url-pattern*

The *context-root* is defined in the `application.xml` or `web.xml` file, and can be overridden in the `sun-application.xml` or `sun-web.xml` file. The *servlet-mapping-url-pattern* is defined in the `web.xml` file.

In the following example, the *context-root* is `my-ws` and the *servlet-mapping-url-pattern* is `/simple`:

```
http://localhost:8080/my-ws/simple
```

You can view the WSDL file of the deployed service in a browser by adding `?WSDL` to the end of the URI. For example:

```
http://localhost:8080/my-ws/simple?WSDL
```

For debugging, you can run a test page for the deployed service in a browser by adding `?Tester` to the end of the URL. For example:

```
http://localhost:8080/my-ws/simple?Tester
```

You can also test a service using the Administration Console. Open the Web Services component, select the web service in the listing on the General tab, and select Test. For details, click the Help button in the Administration Console.

Note – The test page works only for WS-I compliant web services. This means that the tester servlet does not work for services with WSDL files that use RPC/encoded binding.

Generation of the test page is enabled by default. You can disable the test page for a web service by setting the value of the `debugging-enabled` element in the `sun-web.xml` and `sun-ejb-jar.xml` deployment descriptor to `false`. For more information, see the *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

Sun Java EE Engine

Enterprise Server v3 provides the Sun Java EE Engine (Java EE Service Engine), a JSR 208 compliant Java Business Integration (JBI) runtime component that connects Java EE web services to JBI components. The Java EE Service Engine is installed as an add-on component using the Update Tool. Look for the JBI component named Java EE Service Engine. A JBI runtime is not installed with or integrated into Enterprise Server v3 and must be obtained separately. For more information about using the Update Tool to obtain the Java EE Service Engine and other add-on components, see "Update Tool" in *Sun GlassFish Enterprise Server v3 Administration Guide*.

The Java EE Service Engine acts as a bridge between the Java EE and JBI runtime environments for web service providers and web service consumers. The Java EE Service Engine provides better performance than a SOAP over HTTP binding component due to in-process communication between components and additional protocols provided by JBI binding components such as JMS, SMTP, and File.

The JSR 208 (`http://jcp.org/en/jsr/detail?id=208`) specification allows transactions to be propagated to other components using a message exchange property specified in the `JTA_TRANSACTION_PROPERTY_NAME` field. The Java EE Service Engine uses this property to set and get a transaction object from the JBI message exchange. It then uses the transaction object to take part in a transaction. This means a Java EE application or module can take part in a transaction started by a JBI application. Conversely, a JBI application can take part in a transaction started by a Java EE application or module.

Similarly, the JSR 208 specification allows a security subject to be propagated as a message exchange property named `javax.jbi.security.subject`. Thus a security subject can be propagated from a Java EE application or module to a JBI application or the reverse.

To deploy a Java EE application or module as a JBI service unit, use the `asadmin deploy` command, or autodeployment. For more information about the `asadmin deploy` command, see the *Sun GlassFish Enterprise Server v3 Reference Manual*. For more information about autodeployment, see "To Deploy an Application or Module Automatically" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

Using the `jbi.xml` File

Section 6.3.1 of the JSR 208 specification describes the `jbi.xml` file. This is a deployment descriptor, located in the `META-INF` directory. To deploy a Java EE application or module as a JBI service unit, you need only specify a small subset of elements in the `jbi.xml` file. Here is an example provider:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<jbi version="1.0" xmlns="http://java.sun.com/xml/ns/jbi" xmlns:ns0="http://ejbws.jbi.misc/">
  <services binding-component="false">
    <provides endpoint-name="MiscPort" interface-name="ns0:Misc" service-name="ns0:MiscService"/>
  </services>
</jbi>
```

Here is an example consumer:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<jbi version="1.0" xmlns="http://java.sun.com/xml/ns/jbi" xmlns:ns0="http://message.hello.jbi/">
  <services binding-component="false">
    <consumes endpoint-name="MsgPort" interface-name="ns0:Msg" service-name="ns0:MsgService"/>
  </services>
</jbi>
```

The Java EE Service Engine enables the endpoints described in the `provides` section of the `jbi.xml` file in the JBI runtime. Similarly, the Java EE Service Engine routes invocations of the endpoints described in the `consumes` section from the Java EE web service consumer to the JBI runtime.

◆ ◆ ◆ CHAPTER 7

Using the Java Persistence API

Sun GlassFish Enterprise Server support for the Java Persistence API includes all required features described in the Java Persistence Specification, also known as JSR 317 (`http://jcp.org/en/jsr/detail?id=317`). The Java Persistence API can be used with non-EJB components outside the EJB container.

The Java Persistence API provides an object/relational mapping facility to Java developers for managing relational data in Java applications. For basic information about the Java Persistence API, see Part VI, "Persistence," in *The Java EE 6 Tutorial, Volume I*.

This chapter contains Enterprise Server specific information on using the Java Persistence API in the following topics:

Note – The default persistence provider in the Enterprise Server is based on the EclipseLink Java Persistence API implementation. All configuration options in EclipseLink are available to applications that use the Enterprise Server's default persistence provider.

Note – The Web Profile of the Enterprise Server supports the EJB 3.1 Lite specification, which allows enterprise beans within web applications, among other features. The full Enterprise Server supports the entire EJB 3.1 specification. For details, see JSR 318 (`http://jcp.org/en/jsr/detail?id=318`).

Specifying the Database

The Enterprise Server uses the bundled Java DB (Derby) database by default. If the `transaction-type` element is omitted or specified as `JTA` and both the `jta-data-source` and `non-jta-data-source` elements are omitted in the `persistence.xml` file, Java DB is used as a JTA data source. If `transaction-type` is specified as `RESOURCE_LOCAL` and both `jta-data-source` and `non-jta-data-source` are omitted, Java DB is used as a non-JTA data source.

To use a non-default database, either specify a value for the `jta-data-source` element, or set the `transaction-type` element to `RESOURCE_LOCAL` and specify a value for the `non-jta-data-source` element.

If you are using the default persistence provider, the provider attempts to automatically detect the database type based on the connection metadata. This database type is used to issue SQL statements specific to the detected database type's dialect. You can specify the optional `eclipselink.target-database` property to guarantee that the database type is correct. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
    <persistence xmlns="http://java.sun.com/xml/ns/persistence">
        <persistence-unit name ="em1">
            <jta-data-source>jdbc/MyDB2DB</jta-data-source>
            <properties>
                <property name="eclipselink.target-database"
                    value="DB2"/>
            </properties>
        </persistence-unit>
    </persistence>
```

The following `eclipselink.target-database` property values are allowed. Supported platforms have been tested with the Enterprise Server and are found to be Java EE compatible.

```
//Supported platforms
JavaDB
Derby
Oracle
MySQL4
//Others available
SQLServer
DB2
Sybase
```

```
PostgreSQL
Informix
TimesTen
Attunity
HSQL
SQLAnyWhere
DBase
DB2Mainframe
Cloudscape
PointBase
```

For more information about the `eclipselink.target-database` property, see Using EclipseLink JPA Extensions for Session, Target Database and Target Application Server.

To use the Java Persistence API outside the EJB container (in Java SE mode), do not specify the `jta-data-source` or `non-jta-data-source` elements. Instead, specify the `provider` element and any additional properties required by the JDBC driver or the database. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
    <persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
        <persistence-unit name ="em2">
            <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
            <class>ejb3.war.servlet.JpaBean</class>
            <properties>
                <property name="eclipselink.target-database"
                    value="Derby"/>
                <!-- JDBC connection properties -->
                <property name="eclipselink.jdbc.driver" value="org.apache.derby.jdbc.ClientDriver"/>
                <property name="eclipselink.jdbc.url"
value="jdbc:derby://localhost:1527/testdb;retrieveMessagesFromServerOnGetMessage=true;create=true;"/>
                <property name="eclipselink.jdbc.user" value="APP"/>
                <property name="eclipselink.jdbc.password" value="APP"/>
            </properties>
        </persistence-unit>
    </persistence>
```

For more information about `eclipselink` properties, see "Additional Database Properties" on page 122.

For a list of the JDBC drivers currently supported by the Enterprise Server, see the *Sun GlassFish Enterprise Server v3 Release Notes*. For configurations of supported and other drivers, see "Configuration Specifics for JDBC Drivers" in *Sun GlassFish Enterprise Server v3 Administration Guide*.

To change the persistence provider, see "Changing the Persistence Provider" on page 126.

Additional Database Properties

If you are using the default persistence provider, you can specify in the `persistence.xml` file the database properties listed at How to Use EclipseLink JPA Extensions for JDBC Connection Communication.

For schema generation properties, see "Generation Options" on page 124. For query hints, see "Query Hints" on page 126.

Configuring the Cache

If you are using the default persistence provider, you can configure whether caching occurs, the type of caching, the size of the cache, and whether client sessions share the cache. Caching properties for the default persistence provider are described in detail at Using EclipseLink JPA Extensions for Entity Caching.

Setting the Logging Level

One of the default persistence provider's properties that you can set in the `persistence.xml` file is `eclipselink.logging.level`. For example, setting the logging level to `FINE` or higher logs all SQL statements. For details about this property, see Using EclipseLink JPA Extensions for Logging.

You can also set the EclipseLink logging level globally in the Enterprise Server by setting a JVM option using the `asadmin` command. For example:

```
asadmin create-jvm-options -Declipselink.logging.level=FINE
```

Setting the logging level to `OFF` disables EclipseLink logging. A logging level set in the `persistence.xml` file takes precedence over the global logging level.

Using Lazy Loading

`OneToMany` and `ManyToMany` mappings are loaded lazily by default in compliance with the Java Persistence Specification. `OneToOne` and `ManyToOne` mappings are loaded eagerly by default.

For basic information about lazy loading, see What You May Need to Know About EclipseLink JPA Lazy Loading.

Primary Key Generation Defaults

In the descriptions of the `@GeneratedValue`, `@SequenceGenerator`, and `@TableGenerator` annotations in the Java Persistence Specification, certain defaults are noted as specific to the persistence provider. The default persistence provider's primary key generation defaults are listed here.

`@GeneratedValue` defaults are as follows:

- Using `strategy=AUTO` (or no `strategy`) creates a `@TableGenerator` named `SEQ_GEN` with default settings. Specifying a `generator` has no effect.
- Using `strategy=TABLE` without specifying a `generator` creates a `@TableGenerator` named `SEQ_GEN_TABLE` with default settings. Specifying a `generator` but no `@TableGenerator` creates and names a `@TableGenerator` with default settings.
- Using `strategy=IDENTITY` or `strategy=SEQUENCE` produces the same results, which are database-specific.
  - For Oracle databases, not specifying a `generator` creates a `@SequenceGenerator` named `SEQ_GEN_SEQUENCE` with default settings. Specifying a `generator` but no `@SequenceGenerator` creates and names a `@SequenceGenerator` with default settings.
  - For PostgreSQL databases, a `SERIAL` column named *entity-table_pk-column*`_SEQ` is created.
  - For MySQL databases, an `AUTO_INCREMENT` column is created.
  - For other supported databases, an `IDENTITY` column is created.

The `@SequenceGenerator` annotation has one default specific to the default provider. The default `sequenceName` is the specified `name`.

`@TableGenerator` defaults are as follows:

- The default `table` is `SEQUENCE`.
- The default `pkColumnName` is `SEQ_NAME`.
- The default `valueColumnName` is `SEQ_COUNT`.
- The default `pkColumnValue` is the specified `name`, or the default `name` if no `name` is specified.

Automatic Schema Generation

The automatic schema generation feature of the Enterprise Server defines database tables based on the fields or properties in entities and the relationships between the fields or properties. This insulates developers from many of the database related aspects of development, allowing them to focus on entity development. The resulting schema is usable as-is or can be given to a database administrator for tuning with respect to performance, security, and so on. This section covers the following topics:

- "Annotations" on page 124
- "Generation Options" on page 124

Note – Automatic schema generation is supported on an all-or-none basis: it expects that no tables exist in the database before it is executed. It is not intended to be used as a tool to generate extra tables or constraints.

Deployment won't fail if all tables are not created, and undeployment won't fail if not all tables are dropped. Instead, an error is written to the server log. This is done to allow you to investigate the problem and fix it manually. You should not rely on the partially created database schema to be correct for running the application.

Annotations

The following annotations are used in automatic schema generation: `@AssociationOverride`, `@AssociationOverrides`, `@AttributeOverride`, `@AttributeOverrides`, `@Column`, `@DiscriminatorColumn`, `@DiscriminatorValue`, `@Embedded`, `@EmbeddedId`, `@GeneratedValue`, `@Id`, `@IdClass`, `@JoinColumn`, `@JoinColumns`, `@JoinTable`, `@Lob`, `@ManyToMany`, `@ManyToOne`, `@OneToMany`, `@OneToOne`, `@PrimaryKeyJoinColumn`, `@PrimaryKeyJoinColumns`, `@SecondaryTable`, `@SecondaryTables`, `@SequenceGenerator`, `@Table`, `@TableGenerator`, `@UniqueConstraint`, and `@Version`. For information about these annotations, see the Java Persistence Specification.

For `@Column` annotations, the `insertable` and `updatable` elements are not used in automatic schema generation.

For `@OneToMany` and `@ManyToOne` annotations, no `ForeignKeyConstraint` is created in the resulting DDL files.

Generation Options

Schema generation properties or `asadmin` command line options can control automatic schema generation by the following:

- Creating tables during deployment
- Dropping tables during undeployment
- Dropping and creating tables during redeployment
- Generating the DDL files

Note – Before using these options, make sure you have a properly configured database. See "Specifying the Database" on page 120.

Optional schema generation properties control the automatic creation of database tables. You can specify them in the `persistence.xml` file. For more information, see Using EclipseLink JPA Extensions for Schema Generation.

The following options of the `asadmin deploy` or `asadmin deploydir` command control the automatic creation of database tables at deployment.

TABLE 7–1 The `asadmin deploy` and `asadmin deploydir` Generation Options

Option	Default	Description
`--createtables`	none	If `true`, causes database tables to be created for entities that need them. If `false`, does not create tables. If not specified, the value of the `eclipselink.ddl-generation` property in `persistence.xml` is used.
`--dropandcreatetables`	none	If `true`, and if tables were automatically created when this application was last deployed, tables from the earlier deployment are dropped and fresh ones are created. If `true`, and if tables were *not* automatically created when this application was last deployed, no attempt is made to drop any tables. If tables with the same names as those that would have been automatically created are found, the deployment proceeds, but a warning is thrown to indicate that tables could not be created. If `false`, the `eclipselink.ddl-generation` property setting in `persistence.xml` is overridden.

The following options of the `asadmin undeploy` command control the automatic removal of database tables at undeployment.

TABLE 7–2 The `asadmin undeploy` Generation Options

Option	Default	Description
`--droptables`	none	If `true`, causes database tables that were automatically created when the entities were last deployed to be dropped when the entities are undeployed. If `false`, does not drop tables. If not specified, tables are dropped only if the `eclipselink.ddl-generation` property setting in `persistence.xml` is `drop-and-create-tables`.

For more information about the `asadmin deploy`, `asadmin deploydir`, and `asadmin undeploy` commands, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

When `asadmin` deployment options and `persistence.xml` options are both specified, the `asadmin` deployment options take precedence.

The Ant tasks `sun-appserv-deploy` and `sun-appserv-undeploy` are equivalent to `asadmin deploy` and `asadmin undeploy`, respectively. These Ant tasks also override the `persistence.xml` options. For details, see Chapter 3, "Using Ant with Enterprise Server."

Query Hints

Query hints are additional, implementation-specific configuration settings. You can use hints in your queries in the following format:

`setHint("`*hint-name*`", `*hint-value*`)`

For example:

```
Customer customer = (Customer)entityMgr.
     createNamedQuery("findCustomerBySSN").
     setParameter("SSN", "123-12-1234").
     setHint("eclipselink.refresh", true).
     getSingleResult();
```

For more information about the query hints available with the default provider, see How to Use EclipseLink JPA Query Hints.

Changing the Persistence Provider

Note – The previous sections in this chapter apply only to the default persistence provider. If you change the provider for a module or application, the provider-specific database properties, query hints, and schema generation features described in this chapter do not apply.

You can change the persistence provider for an application in the manner described in the Java Persistence API Specification.

First, install the provider. Copy the provider JAR files to the *domain-dir*`/lib` directory, and restart the Enterprise Server. For more information about the *domain-dir*`/lib` directory, see "Using the Common Class Loader" on page 38. The new persistence provider is now available to all modules and applications deployed on servers that share the same configuration. However, the *default* provider remains the same.

In your persistence unit, specify the provider and any properties the provider requires in the `persistence.xml` file. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
    <persistence xmlns="http://java.sun.com/xml/ns/persistence">
        <persistence-unit name ="em3">
            <provider>com.company22.persistence.PersistenceProviderImpl</provider>
            <properties>
                <property name="company22.database.name" value="MyDB"/>
            </properties>
        </persistence-unit>
    </persistence>
```

To migrate from Oracle TopLink to EclipseLink, see Migrating from Oracle TopLink to EclipseLink (`http://wiki.eclipse.org/EclipseLink/Examples/MigratingFromOracleTopLink`).

Restrictions and Optimizations

This section discusses restrictions and performance optimizations that affect using the Java Persistence API.

Oracle Database Enhancements

EclipseLink features a number of enhancements for use with Oracle databases. These enhancements require classes from the Oracle JDBC driver JAR files to be visible to EclipseLink at runtime. If you place the JDBC driver JAR files in *domain-dir*`/lib`, the classes are not visible to Enterprise Server components, including EclipseLink.

If you are using an Oracle database, put JDBC driver JAR files in *domain-dir*`/lib/ext` instead. This ensures that the JDBC driver classes are visible to EclipseLink.

If you do not want to take advantage of Oracle-specific extensions from EclipseLink or you cannot put JDBC driver JAR files in *domain-dir*`/lib/ext`, set the `eclipselink.target-database` property to the value `org.eclipse.persistence.platform.database.OraclePlatform`. For more information about the `eclipselink.target-database` property, see "Specifying the Database" on page 120.

Extended Persistence Context

If a stateful session bean is passivated, its extended persistence context could be lost when the stateful session bean is activated. In this environment, it is safe to store an extended persistence context in a stateful session bean only if you can safely disable stateful session bean passivation altogether. This is possible, but trade-offs in memory utilization must be carefully examined before choosing this option.

It is safe to store a reference to an extended persistence context in an `HttpSession`.

Using @OrderBy with a Shared Session Cache

Setting `@OrderBy` on a `ManyToMany` or `OneToMany` relationship field in which a `List` represents the Many side doesn't work if the session cache is shared. Use one of the following workarounds:

- Have the application maintain the order so the `List` is cached properly.
- Refresh the session cache using `EntityManager.refresh()` if you don't want to maintain the order during creation or modification of the `List`.
- Disable session cache sharing in `persistence.xml` as follows:

```
<property name="eclipselink.cache.shared.default" value="false"/>
```

Using BLOB or CLOB Types with the Inet Oraxo JDBC Driver

To use BLOB or CLOB data types larger than 4 KB for persistence using the Inet Oraxo JDBC Driver for Oracle Databases, you must set the database's `streamstolob` property value to `true`.

Database Case Sensitivity

Mapping references to column or table names must be in accordance with the expected column or table name case, and ensuring this is the programmer's responsibility. If column or table names are not explicitly specified for a field or entity, the Enterprise Server uses upper case column names by default, so any mapping references to the column or table names must be in upper case. If column or table names are explicitly specified, the case of all mapping references to the column or table names must be in accordance with the case used in the specified names.

The following are examples of how case sensitivity affects mapping elements that refer to columns or tables. Programmers must keep case sensitivity in mind when writing these mappings.

Unique Constraints

If column names are not explicitly specified on a field, unique constraints and foreign key mappings must be specified using uppercase references. For example:

```
@Table(name="Department", uniqueConstraints={ @UniqueConstraint ( columnNames= { "DEPTNAME" } ) } )
```

The other way to handle this is by specifying explicit column names for each field with the required case. For example:

```
@Table(name="Department", uniqueConstraints={ @UniqueConstraint ( columnNames= { "deptName" } ) } )
public class Department{ @Column(name="deptName") private String deptName; }
```

Otherwise, the `ALTER TABLE` statement generated by the Enterprise Server uses the incorrect case, and the creation of the unique constraint fails.

Foreign Key Mapping

Use `@OneToMany(mappedBy="COMPANY")` or specify an explicit column name for the `Company` field on the `Many` side of the relationship.

SQL Result Set Mapping

Use the following elements:

```
<sql-result-set-mapping name="SRSMName" >
   <entity-result entity-class="entities.someEntity" />
   <column-result name="UPPERCASECOLUMNNAME" />
</sql-result-set-mapping>
```

Or specify an explicit column name for the `upperCaseColumnName` field.

Named Native Queries and JDBC Queries

Column or table names specified in SQL queries must be in accordance with the expected case. For example, MySQL requires column names in the `SELECT` clause of JDBC queries to be uppercase, while PostgreSQL and Sybase require table names to be uppercase in all JDBC queries.

PostgreSQL Case Sensitivity

PostgreSQL stores column and table names in lower case. JDBC queries on PostgreSQL retrieve column or table names in lowercase unless the names are quoted. For example:

```
use aliases Select m.ID AS \"ID\" from Department m
```

Use the backslash as an escape character in the class file, but not in the `persistence.xml` file.

Sybase Finder Limitation

If a finder method with an input greater than 255 characters is executed and the primary key column is mapped to a VARCHAR column, Sybase attempts to convert type VARCHAR to type TEXT and generates the following error:

```
com.sybase.jdbc2.jdbc.SybSQLException: Implicit conversion from datatype
'TEXT' to 'VARCHAR' is not allowed.  Use the CONVERT function to run this
query.
```

To avoid this error, make sure the finder method input is less than 255 characters.

MySQL Database Restrictions

The following restrictions apply when you use a MySQL database with the Enterprise Server for persistence.

- MySQL treats `int1` and `int2` as reserved words. If you want to define `int1` and `int2` as fields in your table, use ‘`int1`‘ and ‘`int2`‘ field names in your SQL file.
- When `VARCHAR` fields get truncated, a warning is displayed instead of an error. To get an error message, start the MySQL database in strict SQL mode.
- The order of fields in a foreign key index must match the order in the explicitly created index on the primary table.
- The `CREATE TABLE` syntax in the SQL file must end with the following line.

  ```
  )  Engine=InnoDB;
  ```

  `InnoDB` provides MySQL with a transaction-safe (ACID compliant) storage engine having commit, rollback, and crash recovery capabilities.
- For a `FLOAT` type field, the correct precision must be defined. By default, MySQL uses four bytes to store a `FLOAT` type that does not have an explicit precision definition. For example, this causes a number such as 12345.67890123 to be rounded off to 12345.7 during an `INSERT`. To prevent this, specify `FLOAT(10,2)` in the DDL file, which forces the database to use an eight-byte double-precision column. For more information, see `http://dev.mysql.com/doc/mysql/en/numeric-types.html`.
- To use `||` as the string concatenation symbol, start the MySQL server with the `--sql-mode="PIPES_AS_CONCAT"` option. For more information, see `http://dev.mysql.com/doc/refman/5.0/en/server-sql-mode.html` and `http://dev.mysql.com/doc/mysql/en/ansi-mode.html`.
- MySQL always starts a new connection when `autoCommit==true` is set. This ensures that each SQL statement forms a single transaction on its own. If you try to rollback or commit an SQL statement, you get an error message.

  ```
  javax.transaction.SystemException: java.sql.SQLException:
  Can’t call rollback when autocommit=true
  ```

  ```
  javax.transaction.SystemException: java.sql.SQLException:
  Error open transaction is not closed
  ```

  To resolve this issue, add `relaxAutoCommit=true` to the JDBC URL. For more information, see `http://forums.mysql.com/read.php?39,31326,31404`.
- MySQL does not allow a `DELETE` on a row that contains a reference to itself. Here is an example that illustrates the issue.

  ```
  create table EMPLOYEE (
          empId   int         NOT NULL,
          salary  float(25,2) NULL,
          mgrId   int         NULL,
          PRIMARY KEY (empId),
  ```

```
        FOREIGN KEY (mgrId) REFERENCES EMPLOYEE (empId)
        ) ENGINE=InnoDB;

        insert into Employee values (1, 1234.34, 1);
        delete from Employee where empId = 1;
```

This example fails with the following error message.

```
ERROR 1217 (23000): Cannot delete or update a parent row:
a foreign key constraint fails
```

To resolve this issue, change the table creation script to the following:

```
create table EMPLOYEE (
        empId   int         NOT NULL,
        salary  float(25,2) NULL,
        mgrId   int         NULL,
        PRIMARY KEY (empId),
        FOREIGN KEY (mgrId) REFERENCES EMPLOYEE (empId)
        ON DELETE SET NULL
        ) ENGINE=InnoDB;

        insert into Employee values (1, 1234.34, 1);
        delete from Employee where empId = 1;
```

This can be done only if the foreign key field is allowed to be null. For more information, see http://bugs.mysql.com/bug.php?id=12449 and http://dev.mysql.com/doc/mysql/en/innodb-foreign-key-constraints.html.

◆ ◆ ◆ CHAPTER 8

Developing Web Applications

This chapter describes how web applications are supported in the Sun GlassFish Enterprise Server and includes the following sections:

For general information about web applications, see Part II, "The Web Tier," in *The Java EE 6 Tutorial, Volume I*.

Note – The Web Profile of the Enterprise Server supports the EJB 3.1 Lite specification, which allows enterprise beans within web applications, among other features. The full Enterprise Server supports the entire EJB 3.1 specification. For details, see JSR 318 (`http://jcp.org/en/jsr/detail?id=318`).

Using Servlets

Enterprise Server supports the Java Servlet Specification version 3.0.

Note – Servlet API version 3.0 is fully backward compatible with versions 2.3, 2.4, and 2.5, so all existing servlets should work without modification or recompilation.

To develop servlets, use Sun Microsystems' Java Servlet API. For information about using the Java Servlet API, see the documentation provided by Sun Microsystems at `http://java.sun.com/products/servlet/index.html`.

The Enterprise Server provides the `wscompile` and `wsdeploy` tools to help you implement a web service endpoint as a servlet. For more information about these tools, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

This section describes how to create effective servlets to control application interactions running on an Enterprise Server, including standard-based servlets. In addition, this section describes the Enterprise Server features to use to augment the standards.

This section contains the following topics:

Caching Servlet Results

The Enterprise Server can cache the results of invoking a servlet, a JSP, or any URL pattern to make subsequent invocations of the same servlet, JSP, or URL pattern faster. The Enterprise Server caches the request results for a specific amount of time. In this way, if another data call occurs, the Enterprise Server can return the cached data instead of performing the operation again. For example, if your servlet returns a stock quote that updates every 5 minutes, you set the cache to expire after 300 seconds.

Whether to cache results and how to cache them depends on the data involved. For example, it makes no sense to cache the results of a quiz submission, because the input to the servlet is different each time. However, it makes sense to cache a high level report showing demographic data taken from quiz results that is updated once an hour.

To define how an Enterprise Server web application handles response caching, you edit specific fields in the `sun-web.xml` file.

Note – A servlet that uses caching is not portable.

For Javadoc tool pages relevant to caching servlet results, go to `https://glassfish.dev.java.net/nonav/docs/v3/api/` and click on the `com.sun.appserv.web.cache` package.

For information about JSP caching, see "JSP Caching" on page 139.

The rest of this section covers the following topics:

Caching Features

The Enterprise Server has the following web application response caching capabilities:

- Caching is configurable based on the servlet name or the URI.
- When caching is based on the URI, this includes user specified parameters in the query string. For example, a response from `/garden/catalog?category=roses` is different from a response from `/garden/catalog?category=lilies`. These responses are stored under different keys in the cache.
- Cache size, entry timeout, and other caching behaviors are configurable.
- Entry timeout is measured from the time an entry is created or refreshed. To override this timeout for an individual cache mapping, specify the `cache-mapping` subelement `timeout`.
- To determine caching criteria programmatically, write a class that implements the `com.sun.appserv.web.cache.CacheHelper` interface. For example, if only a servlet knows when a back end data source was last modified, you can write a helper class to retrieve the last modified timestamp from the data source and decide whether to cache the response based on that timestamp.
- To determine cache key generation programmatically, write a class that implements the `com.sun.appserv.web.cache.CacheKeyGenerator` interface. See "The `CacheKeyGenerator` Interface" on page 137.
- All non-ASCII request parameter values specified in cache key elements must be URL encoded. The caching subsystem attempts to match the raw parameter values in the request query string.
- Since newly updated classes impact what gets cached, the web container clears the cache during dynamic deployment or reloading of classes.
- The following `HttpServletRequest` request attributes are exposed.
  - `com.sun.appserv.web.cachedServletName`, the cached servlet target
  - `com.sun.appserv.web.cachedURLPattern`, the URL pattern being cached
- Results produced by resources that are the target of a `RequestDispatcher.include()` or `RequestDispatcher.forward()` call are cached if caching has been enabled for those resources. For details, see "cache-mapping" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide* and "dispatcher" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*. These are elements in the `sun-web.xml` file.

Default Cache Configuration

If you enable caching but do not provide any special configuration for a servlet or JSP, the default cache configuration is as follows:

- The default cache timeout is 30 seconds.
- Only the HTTP GET method is eligible for caching.
- HTTP requests with cookies or sessions automatically disable caching.

- No special consideration is given to `Pragma:`, `Cache-control:`, or `Vary:` headers.
- The default key consists of the Servlet Path (minus `pathInfo` and the query string).
- A "least recently used" list is maintained to evict cache entries if the maximum cache size is exceeded.
- Key generation concatenates the servlet path with key field values, if any are specified.
- Results produced by resources that are the target of a `RequestDispatcher.include()` or `RequestDispatcher.forward()` call are never cached.

Caching Example

Here is an example cache element in the `sun-web.xml` file:

```
<cache max-capacity="8192" timeout="60">
<cache-helper name="myHelper" class-name="MyCacheHelper"/>
<cache-mapping>
    <servlet-name>myservlet</servlet-name>
    <timeout name="timefield">120</timeout>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
</cache-mapping>
<cache-mapping>
    <url-pattern> /catalog/* </url-pattern>
    <!-- cache the best selling category; cache the responses to
       -- this resource only when the given parameters exist. Cache
       -- only when the catalog parameter has 'lilies' or 'roses'
       -- but no other catalog varieties:
      -- /orchard/catalog?best&category='lilies'
      -- /orchard/catalog?best&category='roses'
      -- but not the result of
       -- /orchard/catalog?best&category='wild'
    -->
    <constraint-field name='best' scope='request.parameter'/>
    <constraint-field name='category' scope='request.parameter'>
        <value> roses </value>
        <value> lilies </value>
    </constraint-field>
     <!-- Specify that a particular field is of given range but the
       -- field doesn't need to be present in all the requests -->
    <constraint-field name='SKUnum' scope='request.parameter'>
        <value match-expr='in-range'> 1000 - 2000 </value>
    </constraint-field>
    <!-- cache when the category matches with any value other than
       -- a specific value -->
    <constraint-field name="category" scope="request.parameter>
        <value match-expr="equals" cache-on-match-failure="true">
       bogus
        </value>
    </constraint-field>
</cache-mapping>
<cache-mapping>
    <servlet-name> InfoServlet </servlet-name>
    <cache-helper-ref>myHelper</cache-helper-ref>
</cache-mapping>
</cache>
```

For more information about the `sun-web.xml` caching settings, see "cache" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

The `CacheKeyGenerator` Interface

The built-in default `CacheHelper` implementation allows web applications to customize the key generation. An application component (in a servlet or JSP) can set up a custom `CacheKeyGenerator` implementation as an attribute in the `ServletContext`.

The name of the context attribute is configurable as the `value` of the `cacheKeyGeneratorAttrName` property in the `default-helper` element of the `sun-web.xml` deployment descriptor. For more information, see "default-helper" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

About the Servlet Engine

Servlets exist in and are managed by the servlet engine in the Enterprise Server. The servlet engine is an internal object that handles all servlet meta functions. These functions include instantiation, initialization, destruction, access from other components, and configuration management. This section covers the following topics:

Instantiating and Removing Servlets

After the servlet engine instantiates the servlet, the servlet engine calls the servlet's `init()` method to perform any necessary initialization. You can override this method to perform an initialization function for the servlet's life, such as initializing a counter.

When a servlet is removed from service, the servlet engine calls the `destroy()` method in the servlet so that the servlet can perform any final tasks and deallocate resources. You can override this method to write log messages or clean up any lingering connections that won't be caught in garbage collection.

Request Handling

When a request is made, the Enterprise Server hands the incoming data to the servlet engine. The servlet engine processes the request's input data, such as form data, cookies, session information, and URL name-value pairs, into an `HttpServletRequest` request object type.

The servlet engine also creates an `HttpServletResponse` response object type. The engine then passes both as parameters to the servlet's `service()` method.

In an HTTP servlet, the default `service()` method routes requests to another method based on the HTTP transfer method: `POST`, `GET`, `DELETE`, `HEAD`, `OPTIONS`, `PUT`, or `TRACE`. For example, HTTP `POST` requests are sent to the `doPost()` method, HTTP `GET` requests are sent to the

`doGet()` method, and so on. This enables the servlet to process request data differently, depending on which transfer method is used. Since the routing takes place in the service method, you generally do not override `service()` in an HTTP servlet. Instead, override `doGet()`, `doPost()`, and so on, depending on the request type you expect.

To perform the tasks to answer a request, override the `service()` method for generic servlets, and the `doGet()` or `doPost()` methods for HTTP servlets. Very often, this means accessing EJB components to perform business transactions, then collating the information in the request object or in a JDBC `ResultSet` object.

Using JavaServer Pages

The Enterprise Server supports the following JSP features:

- JavaServer Pages (JSP) Specification
- Precompilation of JSP files, which is especially useful for production servers
- JSP tag libraries and standard portable tags

For information about creating JSP files, see Sun Microsystem's JavaServer Pages web site at `http://java.sun.com/products/jsp/index.html`.

For information about Java Beans, see Sun Microsystem's JavaBeans web page at `http://java.sun.com/beans/index.html`.

This section describes how to use JavaServer Pages (JSP files) as page templates in an Enterprise Server web application. This section contains the following topics:

- "JSP Tag Libraries and Standard Portable Tags" on page 138
- "JSP Caching" on page 139
- "Options for Compiling JSP Files" on page 142

JSP Tag Libraries and Standard Portable Tags

Enterprise Server supports tag libraries and standard portable tags. For more information, see the JavaServer Pages Standard Tag Library (JSTL) page at `http://java.sun.com/products/jsp/jstl/index.jsp`.

Web applications don't need to bundle copies of the `jsf-impl.jar` or `appserv-jstl.jar` JSP tag libraries (in *as-install*`/lib`) to use JavaServer Faces technology or JSTL, respectively. These tag libraries are automatically available to all web applications.

However, the *as-install*`/lib/appserv-tags.jar` tag library for JSP caching is not automatically available to web applications. See "JSP Caching" on page 139, next.

JSP Caching

JSP caching lets you cache tag invocation results within the Java engine. Each can be cached using different cache criteria. For example, suppose you have invocations to view stock quotes, weather information, and so on. The stock quote result can be cached for 10 minutes, the weather report result for 30 minutes, and so on. JSP caching is described in the following topics:

For more information about response caching as it pertains to servlets, see "Caching Servlet Results" on page 134.

Enabling JSP Caching

To globally enable JSP caching, set the `jspCachingEnabled` property to `true`. The default is `false`. For example:

```
asadmin set server-config.web-container.property.jspCachingEnabled="true"
```

For more information about the `asadmin set` command, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

To enable JSP caching for a single web application, follow these steps:

1. Extract the `META-INF/appserv-tags.tld` file from the *as-install*`/glassfish/modules/web-glue.jar` file.
2. Create a new JAR file (for example, `appserv-tags.jar`) containing just the `META-INF/appserv-tags.tld` file previously extracted.
3. Bundle this new JAR file in the `WEB-INF/lib` directory of your web application.

Note – Web applications that use JSP caching without bundling the tag library are not portable.

Refer to Enterprise Server tags in JSP files as follows:

```
<%@ taglib prefix="prefix" uri="Sun ONE Application Server Tags" %>
```

Subsequently, the cache tags are available as <*prefix*`:cache`> and <*prefix*`:flush`>. For example, if your *prefix* is `mypfx`, the cache tags are available as <`mypfx:cache`> and <`mypfx:flush`>.

Caching Scope

JSP caching is available in three different scopes: `request`, `session`, and `application`. The default is `application`. To use a cache in `request` scope, a web application must specify the `com.sun.appserv.web.taglibs.cache.CacheRequestListener` in its `web.xml` deployment descriptor, as follows:

```
<listener>
   <listener-class>
      com.sun.appserv.web.taglibs.cache.CacheRequestListener
   </listener-class>
</listener>
```

Likewise, for a web application to utilize a cache in `session` scope, it must specify the `com.sun.appserv.web.taglibs.cache.CacheSessionListener` in its `web.xml` deployment descriptor, as follows:

```
<listener>
   <listener-class>
      com.sun.appserv.web.taglibs.cache.CacheSessionListener
   </listener-class>
</listener>
```

To utilize a cache in `application` scope, a web application need not specify any listener. The `com.sun.appserv.web.taglibs.cache.CacheContextListener` is already specified in the `appserv-tags.tld` file.

The `cache` Tag

The cache tag caches the body between the beginning and ending tags according to the attributes specified. The first time the tag is encountered, the body content is executed and cached. Each subsequent time it is run, the cached content is checked to see if it needs to be refreshed and if so, it is executed again, and the cached data is refreshed. Otherwise, the cached data is served.

Attributes of `cache`

The following table describes attributes for the `cache` tag.

TABLE 8–1 The `cache` Attributes

Attribute	Default	Description
`key`	*ServletPath_Suffix*	(optional) The name used by the container to access the cached entry. The cache key is suffixed to the servlet path to generate a key to access the cached entry. If no key is specified, a number is generated according to the position of the tag in the page.

TABLE 8–1 The cache Attributes *(Continued)*

Attribute	Default	Description
`timeout`	`60s`	(optional) The time in seconds after which the body of the tag is executed and the cache is refreshed. By default, this value is interpreted in seconds. To specify a different unit of time, add a suffix to the timeout value as follows: `s` for seconds, `m` for minutes, `h` for hours, `d` for days. For example, `2h` specifies two hours.
`nocache`	`false`	(optional) If set to `true`, the body content is executed and served as if there were no `cache` tag. This offers a way to programmatically decide whether the cached response is sent or whether the body has to be executed, though the response is not cached.
`refresh`	`false`	(optional) If set to `true`, the body content is executed and the response is cached again. This lets you programmatically refresh the cache immediately regardless of the `timeout` setting.
`scope`	`application`	(optional) The scope of the cache. Can be `request`, `session`, or `application`. See "Caching Scope" on page 140.

Example of `cache`

The following example represents a cached JSP file:

```
<%@ taglib prefix="mypfx" uri="Sun ONE Application Server Tags" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<mypfx:cache                  key="${sessionScope.loginId}"
            nocache="${param.nocache}"
            refresh="${param.refresh}"
            timeout="10m">
<c:choose>
    <c:when test="${param.page == 'frontPage'}">
        <%-- get headlines from database --%>
    </c:when>
    <c:otherwise>
        ...
    </c:otherwise>
</c:choose>
</mypfx:cache>
<mypfx:cache timeout="1h">
<h2> Local News </h2>
    <%-- get the headline news and cache them --%>
</mypfx:cache>
```

The `flush` Tag

Forces the cache to be flushed. If a `key` is specified, only the entry with that key is flushed. If no key is specified, the entire cache is flushed.

Attributes of `flush`

The following table describes attributes for the `flush` tag.

TABLE 8–2 The `flush` Attributes

Attribute	Default	Description
`key`	*ServletPath_Suffix*	(optional) The name used by the container to access the cached entry. The cache key is suffixed to the servlet path to generate a key to access the cached entry. If no key is specified, a number is generated according to the position of the tag in the page.
`scope`	`application`	(optional) The scope of the cache. Can be `request`, `session`, or `application`. See "Caching Scope" on page 140.

Examples of `flush`

To flush the entry with `key="foobar"`:

```
<mypfx:flush key="foobar"/>
```

To flush the entire cache:

```
<c:if test="${empty sessionScope.clearCache}">
   <mypfx:flush />
</c:if>
```

Options for Compiling JSP Files

Enterprise Server provides the following ways of compiling JSP source files into servlets:

- JSP files are automatically compiled at runtime.
- The `asadmin deploy` command has a `precompilejsp` option. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.
- The `sun-appserv-jspc` Ant task allows you to precompile JSP files; see "The `sun-appserv-jspc` Task" on page 56.
- The `jspc` command line tool allows you to precompile JSP files at the command line. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Creating and Managing Sessions

This section describes how to create and manage HTTP sessions that allows users and transaction information to persist between interactions.

This section contains the following subsections:

Configuring Sessions

This section covers the following topics:

HTTP Sessions, Cookies, and URL Rewriting

To configure whether and how HTTP sessions use cookies and URL rewriting, edit the `session-properties` and `cookie-properties` elements in the `sun-web.xml` file for an individual web application. For more about the properties you can configure, see "session-properties" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide* and "cookie-properties" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

Coordinating Session Access

Make sure that multiple threads don't simultaneously modify the same session object in conflicting ways.

This is especially likely to occur in web applications that use HTML frames where multiple servlets are executing simultaneously on behalf of the same client. A good solution is to ensure that one of the servlets modifies the session and the others have read-only access.

Saving Sessions During Redeployment

Whenever a redeployment is done, the sessions at that transit time become invalid unless you use the `keepSessions=true` property of the `asadmin redeploy` command. For example:

```
asadmin redeploy --properties keepSessions=true --name hello.war
```

For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

The new class loader of the redeployed application is used to deserialize any sessions previously saved. The usual restrictions about serialization and deserialization apply. For example, any application-specific class referenced by a session attribute may evolve only in a backward-compatible fashion. For more information about class loaders, see Chapter 2, "Class Loaders."

Logging Session Attributes

You can write session attribute values to an access log. The access log format token `%session.`*name*`%` logs one of the following:

- The value of the session attribute with the name *name*

- `NULL-SESSION-ATTRIBUTE-`*name* if the named attribute does not exist in the session
- `NULL-SESSION` if no session exists

For more information about access logging and format tokens, see online help for the Access Log tab of the HTTP Service page in the Administration Console.

Session Managers

A session manager automatically creates new session objects whenever a new session starts. In some circumstances, clients do not join the session, for example, if the session manager uses cookies and the client does not accept cookies.

Enterprise Server offers these session management options, determined by the `session-manager` element's `persistence-type` attribute in the `sun-web.xml` file:

- "The `memory` Persistence Type" on page 144, the default
- "The `file` Persistence Type" on page 145, which uses a file to store session data

---

Note – If the session manager configuration contains an error, the error is written to the server log and the default (`memory`) configuration is used.

For more information, see "session-manager" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

The `memory` Persistence Type

This persistence type is not designed for a production environment that requires session persistence. It provides no session persistence. However, you can configure it so that the session state in memory is written to the file system prior to server shutdown.

To specify the `memory` persistence type for a specific web application, edit the `sun-web.xml` file as in the following example. The `persistence-type` property is optional, but must be set to `memory` if included. This overrides the web container availability settings for the web application.

```
<sun-web-app>
...
<session-config>
    <session-manager persistence-type="memory" />
        <manager-properties>
            <property name="sessionFilename" value="sessionstate" />
        </manager-properties>
    </session-manager>
    ...
</session-config>
...
</sun-web-app>
```

The only manager property that the `memory` persistence type supports is `sessionFilename`, which is listed under "manager-properties" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*. The `sessionFilename` property specifies the name of the file where sessions are serialized and persisted if the web application or the server is stopped. To disable this behavior, specify an empty string as the value of `sessionFilename`.

For more information about the `sun-web.xml` file, see *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

The `file` Persistence Type

This persistence type provides session persistence to the local file system, and allows a single server domain to recover the session state after a failure and restart. The session state is persisted in the background, and the rate at which this occurs is configurable. The store also provides passivation and activation of the session state to help control the amount of memory used. This option is not supported in a production environment. However, it is useful for a development system with a single server instance.

Note – Make sure the `delete` option is set in the `server.policy` file, or expired file-based sessions might not be deleted properly. For more information about `server.policy`, see "The `server.policy` File" on page 88.

To specify the `file` persistence type for a specific web application, edit the `sun-web.xml` file as in the following example. Note that `persistence-type` must be set to `file`. This overrides the web container availability settings for the web application.

```
<sun-web-app>
...
<session-config>
    <session-manager persistence-type="file">
        <store-properties>
            <property name="directory" value="sessiondir" />
        </store-properties>
    </session-manager>
    ...
</session-config>
...
</sun-web-app>
```

The `file` persistence type supports all the manager properties listed under "manager-properties" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide* except `sessionFilename`, and supports the `directory` store property listed under "store-properties" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

For more information about the `sun-web.xml` file, see *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

Using Comet

This section explains the Comet programming technique and how to create and deploy a Comet-enabled application with the Sun GlassFish Enterprise Server.

The following topics are addressed here:

Introduction to Comet

Comet is a programming technique that allows a web server to send updates to clients without requiring the clients to explicitly request them.

This kind of programming technique is called *server push*, which means that the server pushes data to the client. The opposite style is *client pull*, which means that the client must pull the data from the server, usually through a user-initiated event, such as a button click.

Web applications that use the Comet technique can deliver updates to clients in a more timely manner than those that use the client-pull style while avoiding the latency that results from clients frequently polling the server.

One of the many use cases for Comet is a chat room application. When the server receives a message from one of the chat clients, it needs to send the message to the other clients without requiring them to ask for it. With Comet, the server can deliver messages to the clients as they are posted rather than expecting the clients to poll the server for new messages.

To accomplish this scenario, a Comet application establishes a long-lived HTTP connection. This connection is suspended on the server side, waiting for an event to happen before resuming. This kind of connection remains open, allowing an application that uses the Comet technique to send updates to clients when they are available rather than expecting clients to reopen the connection to poll the server for updates.

The Grizzly Implementation of Comet

A limitation of the Comet technique is that you must use it with a web server that supports non-blocking connections to avoid poor performance. Non-blocking connections are those that do not need to allocate one thread for each request. If the web server were to use blocking connections then it might end up holding many thousands of threads, thereby hindering its scalability.

The GlassFish server includes the Grizzly HTTP Engine, which enables asynchronous request processing (ARP) by avoiding blocking connections. Grizzly's ARP implementation accomplishes this by using the Java NIO API.

With Java NIO, Grizzly enables greater performance and scalability by avoiding the limitations experienced by traditional web servers that must run a thread for each request. Instead, Grizzly's ARP mechanism makes efficient use of a thread pool system and also keeps the state of requests so that it can keep requests alive without holding a single thread for each of them.

Grizzly supports two different implementations of Comet:

- "Grizzly Comet" on page 148 — Based on ARP, this includes a set of APIs that you use from a web component to enable Comet functionality in your web application. Grizzly Comet is specific to the Sun GlassFish Enterprise Server.
- "Bayeux Protocol" on page 157 — Often referred to as `Cometd`, it consists of the JSON-based Bayeux message protocol, a set of Dojo or Ajax libraries, and an event handler. The Bayeux protocol uses a publish/subscribe model for server/client communication. The Bayeux protocol is portable, but it is container dependent if you want to invoke it from an Enterprise Java Beans (EJB) component. The Grizzly implementation of `Cometd` consists of a servlet that you reference from your web application.

Client Technologies to Use With Comet

In addition to creating a web component that uses the Comet APIs, you need to enable your client to accept asynchronous updates from the web component. To accomplish this, you can use JavaScript, IFrames, or a framework, such as Dojo.

An IFrame is an HTML element that allows you to include other content in an HTML page. As a result, the client can embed updated content in the IFrame without having to reload the page.

The example in this tutorial employs a combination of JavaScript and IFrames to allow the client to accept asynchronous updates. A servlet included in the example writes out JavaScript code to one of the IFrames. The JavaScript code contains the updated content and invokes a function in the page that updates the appropriate elements in the page with the new content.

The next section explains the two kinds of connections that you can make to the server. While you can use any of the client technologies listed in this section with either kind of connection, it is more difficult to use JavaScript with an HTTP-streaming connection.

Types of Comet Connections

When working with Comet, as implemented in Grizzly, you have two different ways to handle client connections to the server:

- HTTP Streaming
- Long Polling

HTTP Streaming

The HTTP Streaming technique keeps a connection open indefinitely. It never closes, even after the server pushes data to the client.

In the case of HTTP streaming, the application sends a single request and receives responses as they come, reusing the same connection forever. This technique significantly reduces the network latency because the client and the server don't need to open and close the connection.

The basic life cycle of an application using HTTP-streaming is:

request --> suspend --> data available --> write response --> data available --> write response

The client makes an initial request and then suspends the request, meaning that it waits for a response. Whenever data is available, the server writes it to the response.

Long Polling

The long-polling technique is a combination of server-push and client-pull because the client needs to resume the connection after a certain amount of time or after the server pushes an update to the client.

The basic life cycle of an application using long-polling is:

request -> suspend --> data available --> write response --> resume

The client makes an initial request and then suspends the request. When an update is available, the server writes it to the response. The connection closes, and the client optionally resumes the connection.

How to Choose the Type of Connection

If you anticipate that your web application will need to send frequent updates to the client, you should use the HTTP-streaming connection so that the client does not have to frequently reestablish a connection. If you anticipate less frequent updates, you should use the long-polling connection so that the web server does not need to keep a connection open when no updates are occurring. One caveat to using the HTTP-streaming connection is that if you are streaming through a proxy, the proxy can buffer the response from the server. So, be sure to test your application if you plan to use HTTP-streaming behind a proxy.

Grizzly Comet

The following sections describe how to use Grizzly Comet.

- "Deploying and Running a Comet-Enabled Application" on page 156

The Grizzly Comet API

Grizzly's support for Comet includes a small set of APIs that make it easy to add Comet functionality to your web applications. The Grizzly Comet APIs that developers use most often are the following:

- `CometContext`: A Comet context, which is a shareable space to which applications subscribe to receive updates.
- `CometEngine`: The entry point to any component using Comet. Components can be servlets, JavaServer Pages (JSP), JavaServer Faces components, or pure Java classes.
- `CometEvent`: Contains the state of the `CometContext` object
- `CometHandler`: The interface an application implements to be part of one or more Comet contexts.

The way a developer would use this API in a web component is to perform the following tasks:

1. Register the context path of the application with the `CometContext` object:

```
CometEngine cometEngine =
    CometEngine.getEngine();
CometContext cometContext =
    cometEngine.register(contextPath)
```

2. Register the `CometHandler` implementation with the `CometContext` object:

```
cometContext.addCometHandler(handler)
```

3. Notify one or more `CometHandler` implementations when an event happens:

```
cometContext.notify((Object)(handler))
```

The Hidden Frame Example

This rest of this tutorial uses the Hidden Frame example to explain how to develop Comet-enabled web applications. You can download the example from `grizzly.dev.java.net` at Hidden example download. From there, you can download a prebuilt WAR file as well as a JAR file containing the servlet code.

The Hidden Frame example is so called because it uses hidden IFrames. The example allows multiple clients to increment a counter on the server. When a client increments the counter, the server broadcasts the new count to the clients using the Comet technique.

The Hidden Frame example uses the long-polling technique, but you can easily modify it to use HTTP-streaming by removing two lines. See "To Notify the Comet Handler of an Event" on page 153 and "To Create a HTML Page That Updates and Displays the Content" on page 154 for more information on converting the example to use the HTTP-streaming technique.

The client side of the example uses hidden IFrames with embedded JavaScript tags to connect to the server and to asynchronously post content to and accept updates from the server.

The server side of the example consists of a single servlet that listens for updates from clients, updates the counter, and writes JavaScript code to the client that allows it to update the counter on its page.

See "Deploying and Running a Comet-Enabled Application" on page 156 for instructions on how to deploy and run the example.

When you run the example, the following happens:

1. The `index.html` page opens.
2. The browser loads three frames: The first one accesses the servlet using an HTTP GET; the second one loads the `count.html` page, which displays the current count; and the third one loads the `button.html` page, which is used to send the POST request.
3. After clicking the button on the `button.html` page, the page submits a POST request to the servlet.
4. The `doPost` method calls the `onEvent` method of the Comet handler and redirects the incremented count along with some JavaScript to the `count.html` page on the client.
5. The `updateCount` JavaScript function on the `count.html` page updates the counter on the page.
6. Because this example uses long-polling, the JavaScript code on `count.html` calls `doGet` again to resume the connection after the servlet pushes the update.

Creating a Comet-Enabled Application

This section uses the Hidden Frame example application to demonstrate how to develop a Comet application. The main tasks for creating a simple Comet-enabled application are the following:

Developing the Web Component

This section shows you how to create a Comet-enabled web component by giving you instructions for creating the servlet in the Hidden Frame example.

Developing the web component involves performing the following steps:

1. Create a web component to support Comet requests.
2. Register the component with the Comet engine.
3. Define a Comet handler that sends updates to the client.
4. Add the Comet handler to the Comet context.
5. Notify the Comet handler of an event using the Comet context.

To Create a Web Component to Support Comet

1. **Create an empty servlet class, like the following:**

```
import javax.servlet.*;

public class HiddenCometServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
private String contextPath = null;
    @Override
    public void init(ServletConfig config) throws ServletException {}

    @Override
    protected void doGet(HttpServletRequest req,
    HttpServletResponse res)
    throws ServletException, IOException {}

    @Override
    protected void doPost(HttpServletRequest req,
    HttpServletResponse res)
    throws ServletException, IOException {};
}
```

2. **Import the following Comet packages into the servlet class:**

```
import com.sun.grizzly.comet.CometContext;
import com.sun.grizzly.comet.CometEngine;
import com.sun.grizzly.comet.CometEvent;
import com.sun.grizzly.comet.CometHandler;
```

3. **Import these additional classes that you need for incrementing a counter and writing output to the clients:**

```
import java.io.IOException;
import java.io.PrintWriter;
import java.util.concurrent.atomic.AtomicInteger;
```

4. **Add a private variable for the counter:**

```
private final AtomicInteger counter = new AtomicInteger();
```

To Register the Servlet With the Comet Engine

1. **In the servlet's `init` method, add the following code to get the component's context path:**

```
ServletContext context = config.getServletContext();
contextPath = context.getContextPath() + "/hidden_comet";
```

2. **Get an instance of the Comet engine by adding this line after the lines from Step 1:**

```
 CometEngine engine = CometEngine.getEngine();
```

3. **Register the component with the Comet engine by adding the following lines after those from Step 2:**

```
CometContext cometContext = engine.register(contextPath);
cometContext.setExpirationDelay(30 * 1000);
```

▼ To Define a Comet Handler to Send Updates to the Client

1 Create a private class that implements `CometHandler` and add it to the servlet class:

```
private class CounterHandler
    implements CometHandler<HttpServletResponse> {
    private HttpServletResponse response;
}
```

2 Add the following methods to the class:

```
public void onInitialize(CometEvent event)
    throws IOException {}

    public void onInterrupt(CometEvent event)
        throws IOException {
        removeThisFromContext();
    }

    public void onTerminate(CometEvent event)
        throws IOException {
        removeThisFromContext();
    }

    public void attach(HttpServletResponse attachment) {
            this.response = attachment;
    }

    private void removeThisFromContext() throws IOException {
        response.getWriter().close();
        CometContext context =
            CometEngine.getEngine().getCometContext(contextPath);
        context.removeCometHandler(this);
    }
```

You need to provide implementations of these methods when implementing `CometHandler`. The `onInterrupt` and `onTerminate` methods execute when certain changes occur in the status of the underlying TCP communication. The `onInterrupt` method executes when communication is resumed. The `onTerminate` method executes when communication is closed. Both methods call `removeThisFromContext`, which removes the `CometHandler` object from the `CometContext` object.

▼ To Add the Comet Handler to the Comet Context

1 Get an instance of the Comet handler and attach the response to it by adding the following lines to the `doGet` method:

```
CounterHandler handler = new CounterHandler();
handler.attach(res);
```

2 Get the Comet context by adding the following lines to `doGet`:

```
CometEngine engine = CometEngine.getEngine();
CometContext context = engine.getCometContext(contextPath);
```

3 **Add the Comet handler to the Comet context by adding this line to `doGet`:**

```
context.addCometHandler(handler);
```

▼ To Notify the Comet Handler of an Event

1 **Add an `onEvent` method to the `CometHandler` class to define what happens when an event occurs:**

```
public void onEvent(CometEvent event)
    throws IOException {
    if (CometEvent.NOTIFY == event.getType()) {
        int count = counter.get();
        PrintWriter writer = response.getWriter();
        writer.write("<script type='text/javascript'>" +
            "parent.counter.updateCount('" + count + "')" +
            "</script>\n");
        writer.flush();
        event.getCometContext().resumeCometHandler(this);
    }
}
```

This method first checks if the event type is `NOTIFY`, which means that the web component is notifying the `CometHandler` object that a client has incremented the count. If the event type is `NOTIFY`, the `onEvent` method gets the updated count, and writes out JavaScript to the client. The JavaScript includes a call to the `updateCount` function, which will update the count on the clients' pages.

The last line resumes the Comet request and removes it from the list of active `CometHandler` objects. By this line, you can tell that this application uses the long-polling technique. If you were to delete this line, the application would use the HTTP-Streaming technique.

- **For HTTP-Streaming:**

  Add the same code as for long-polling, except do not include the following line:

  ```
  event.getCometContext().resumeCometHandler(this);
  ```

You don't include this line because you do not want to resume the request. Instead, you want the connection to remain open.

2 **Increment the counter and forward the response by adding the following lines to the `doPost` method:**

```
counter.incrementAndGet();
CometEngine engine = CometEngine.getEngine();
CometContext<?> context =
    engine.getCometContext(contextPath);
context.notify(null);
req.getRequestDispatcher("count.html").forward(req, res);
```

When a user clicks the button, the `doPost` method is called. The `doPost` method increments the counter. It then obtains the current `CometContext` object and calls its `notify` method. By calling `context.notify`, the `doPost` method triggers the `onEvent` method you created in the previous step. After `onEvent` executes, `doPost` forwards the response to the clients.

Creating the Client Pages

Developing the HTML pages for the client involves performing these steps:

1. Create a welcome HTML page, called `index.html`, that contains: one hidden frame for connecting to the servlet through an HTTP GET; one IFrame that embeds the `count.html` page, which contains the updated content; and one IFrame that embeds the `button.html` page, which is used for posting updates using HTTP POST.
2. Create the `count.html` page that contains an HTML element that displays the current count and the JavaScript for updating the HTML element with the new count.
3. Create the `button.html` page that contains a button for the users to submit updates.

▼ To Create a HTML Welcome Page That Contains IFrames for Receiving and Sending Updates

1 Create an HTML page called `index.html`.

2 Add the following content to the page:

```
<html>
    <head>
        <title>Comet Example: Counter with Hidden Frame</title>
    </head>
    <body>
  </body>
</html>
```

3 Add IFrames for connecting to the server and receiving and sending updates to `index.html` in between the `body` tags:

```
<frameset>
    <iframe name="hidden" src="hidden_comet"
        frameborder="0" height="0" width="100%"></iframe>
    <iframe name="counter" src="count.html"
        frameborder="0" height="100%" width="100%"></iframe>
<iframe name="button" src="button.html" frameborder="0" height="30%" widget="100%"></iframe>
</frameset>
```

The first frame, which is hidden, points to the servlet by referencing its context path. The second frame displays the content from `count.html`, which displays the current count. The second frame displays the content from `button.html`, which contains the submit button for incrementing the counter.

▼ To Create a HTML Page That Updates and Displays the Content

1 Create an HTML page called `count.html` and add the following content to it:

```
<html>
    <head>
    </head>
```

```
        <body>
            <center>
                <h3>Comet Example: Counter with Hidden Frame</h3>
                <p>
                <b id="count"> </b>
                <p>
            </center>
        </body>
</html>
```

This page displays the current count.

2 **Add JavaScript code that updates the count in the page. Add the following lines in between the `head` tags of `count.html`:**

```
<script type='text/javascript'>
    function updateCount(c) {
        document.getElementById('count').innerHTML = c;
        parent.hidden.location.href = "hidden_comet";
    };
</script>
```

The JavaScript takes the updated count it receives from the servlet and updates the count element in the page. The last line in the `updateCount` function invokes the servlet's `doGet` method again to reestablish the connection.

- **For HTTP-Streaming:**

  Add the same code as for long-polling, except for the following line:

  ```
  parent.hidden.location.href = "hidden_comet"
  ```

  This line invokes the `doGet` method of `CometServlet` again, which would reestablish the connection. In the case of HTTP-Streaming, you want the connection to remain open. Therefore, you don't include this line of code.

▼ To Create the HTML Page That Allows Submitting Updates

- **Create an HTML page called `button.html` and add the following content to it:**

```
<html>
    <head>
    </head>
        <body>
            <center>
                <form method="post" action="hidden_comet">
                    <input type="submit" value="Click">
                </form>
            </center>
        </body>
</html>
```

This page displays a form with a button that allows a user to update the count on the server. The servlet will then broadcast the updated count to all clients.

Creating the Deployment Descriptor

This section describes how to create a deployment descriptor to specify how your Comet-enabled web application should be deployed.

▼ To Create the Deployment Descriptor

- **Create a file called `web.xml` and put the following contents in it:**

```
<?xml version="1.0" encoding="UTF-8"?>
    <web-app version="3.0"
        xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation=
            "http://java.sun.com/xml/ns/javaee
            http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd ">

        <servlet>
            <servlet-name>HiddenCometServlet</servlet-name>
            <servlet-class>
                com.sun.grizzly.samples.comet.HiddenCometServlet
            </servlet-class>
            <load-on-startup>0</load-on-startup>
        </servlet>
        <servlet-mapping>
            <servlet-name>HiddenCometServlet</servlet-name>
            <url-pattern>/hidden_comet</url-pattern>
        </servlet-mapping>
    </web-app>
```

This deployment descriptor contains a servlet declaration and mapping for `HiddenCometServlet`. The `load-on-startup` attribute must be set to 0 so that the Comet-enabled servlet will not load until the client makes a request to it.

Deploying and Running a Comet-Enabled Application

Before running a Comet-enabled application in the Enterprise Server, you need to enable Comet in the server. Then you can deploy the application just as you would any other web application.

When running the application, you need to connect to it from at least two different browsers to experience the effect of the servlet updating all clients in response to one client posting an update to the server.

Enabling Comet in the Enterprise Server

Before running a Comet-enabled application, you need to enable Comet in the HTTP listener for your application by setting a special attribute in the associated protocol configuration. The following example shows the `asadmin set` command that adds this attribute:

```
asadmin set
server-config.network-config.protocols.protocol.http-1.http.comet-support-enabled="true"
```

Substitute the name of the protocol for `http-1`.

▼ To Deploy the Example

These instructions tell you how to deploy the Hidden Frame example.

1. **Download grizzly-comet-hidden-1.7.3.1.war.**
2. **Run the following command to deploy the example:**

   as-install`/bin/asadmin deploy grizzly-comet-hidden-1.7.3.1.war`

▼ To Run the Example

These instructions tell you how to run the Hidden Frame example.

1. **Open two web browsers, preferably two different brands of web browser.**
2. **Enter the following URL in both browsers:**

   `http://localhost:8080/grizzly-comet-hidden/index.html`
3. **When the first page loads in both browsers, click the button in one of the browsers and watch the count change in the other browser window.**

Bayeux Protocol

The Bayeux protocol, often referred to as `Cometd`, greatly simplifies the use of Comet. No server-side coding is needed for servers such as Enterprise Server that support the Bayeux protocol. Just enable Comet and the Bayeux protocol, then write and deploy the client as described in the following tasks:

- "Enabling Comet" on page 157
- "To Configure the `web.xml` File" on page 158
- "To Write, Deploy, and Run the Client" on page 158

Enabling Comet

Before running a Comet-enabled application, you need to enable Comet in the HTTP listener for your application by setting a special attribute in the associated protocol configuration. The following example shows the `asadmin set` command that adds this attribute:

```
asadmin set
server-config.network-config.protocols.protocol.http-1.http.comet-support-enabled="tr
```

Substitute the name of the protocol for `http-1`.

To Configure the `web.xml` File

To enable the Bayeux protocol on the Enterprise Server, you must reference the `CometdServlet` in your web application's `web.xml` file. In addition, if your web application includes a servlet, set the `load-on-startup` value for your servlet to `0` (zero) so that it will not load until the client makes a request to it.

1. **Open the `web.xml` file for your web application in a text editor.**

2. **Add the following XML code to the `web.xml` file:**

```
<servlet>
   <servlet-name>Grizzly Cometd Servlet</servlet-name>
   <servlet-class>
      com.sun.grizzly.cometd.servlet.CometdServlet
   </servlet-class>
   <init-param>
      <description>
         expirationDelay is the long delay before a request is
         resumed. -1 means never.
      </description>
      <param-name>expirationDelay</param-name>
      <param-value>-1</param-value>
   </init-param>
   <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
   <servlet-name>Grizzly Cometd Servlet</servlet-name>
   <url-pattern>/cometd/*</url-pattern>
</servlet-mapping>
```

   Note that the `load-on-startup` value for the `CometdServlet` is `1`.

3. **If your web application includes a servlet, set the `load-on-startup` value to `0` for your servlet (not the `CometdServlet`) as follows:**

```
<servlet>
   ...
   <load-on-startup>0</load-on-startup>
</servlet>
```

4. **Save the `web.xml` file.**

To Write, Deploy, and Run the Client

The examples in this task are taken from the example chat application that is posted and discussed at `http://weblogs.java.net/blog/jfarcand/archive/2007/02/gcometd_introdu_1.html`.

1. **Add script tags to the HTML page. For example:**

```
<script type="text/javascript" src="chat.js"></script>
```

2. **In the script, call the needed libraries. For example:**

```
dojo.require("dojo.io.cometd");
```

3. **In the script, use `publish` and `subscribe` methods to send and receive messages. For example:**

```
cometd.subscribe("/chat/demo", false, room, "_chat");
cometd.publish("/chat/demo", { user: room._username, chat: text});
```

4. **Deploy the web application as you would any other web application. For example:**

```
asadmin deploy cometd-example.war
```

5. **Run the application as you would any other web application.**

   The context root for the example chat application is `/cometd` and the HTML page is `index.html`. So the URL might look like this:

```
http://localhost:8080/cometd/index.html
```

See Also For more information about deployment in the Enterprise Server, see the *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

For more information about the Bayeux protocol, see *Bayeux Protocol* (`http://svn.cometd.com/trunk/bayeux/bayeux.html`).

For more information about the Dojo toolkit, see `http://dojotoolkit.org/`.

For information about pushing data from an external component such as an EJB module, see the example at `http://blogs.sun.com/swchan/entry/java_api_for_cometd`. Using this Grizzly Java API for `Cometd` makes your web application non-portable. Running your application on a server that doesn't support Grizzly Comet will not work.

For information about REpresentational State Transfer (RESTful) web services and Comet, see RESTful Web Services and Comet (`http://developers.sun.com/appserver/reference/techart/cometslideshow.html`).

Advanced Web Application Features

This section includes summaries of the following topics:

Internationalization Issues

This section covers internationalization as it applies to the following:

The Server's Default Locale

To set the default locale of the entire Enterprise Server, which determines the locale of the Administration Console, the logs, and so on, use the Administration Console. Select the Enterprise Server component, the Advanced tab, and the Domain Attributes tab. Then type a value in the Locale field. For details, click the Help button in the Administration Console.

Servlet Character Encoding

This section explains how the Enterprise Server determines the character encoding for the servlet request and the servlet response. For encodings you can use, see `http://java.sun.com/javase/6/docs/technotes/guides/intl/encoding.doc.html`.

Servlet Request

When processing a servlet request, the server uses the following order of precedence, first to last, to determine the request character encoding:

- The `getCharacterEncoding()` method
- A hidden field in the form, specified by the `form-hint-field` attribute of the `parameter-encoding` element in the `sun-web.xml` file
- The `default-charset` attribute of the `parameter-encoding` element in the `sun-web.xml` file
- The default, which is `ISO-8859-1`

For details about the `parameter-encoding` element, see "parameter-encoding" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

Servlet Response

When processing a servlet response, the server uses the following order of precedence, first to last, to determine the response character encoding:

- The `setCharacterEncoding()` or `setContentType()` method

- The `setLocale()` method
- The default, which is `ISO-8859-1`

Virtual Server Properties

You can set virtual server properties in the following ways:

- You can define virtual server properties using the `asadmin create-virtual-server` command. For example:

  ```
  asadmin create-virtual-server --hosts localhost --property authRealm=ldap MyVS
  ```

  For details and a complete list of virtual server properties, see `create-virtual-server(1)`.
- You can define virtual server properties using the `asadmin set` command. For example:

  ```
  asadmin set server-config.http-service.virtual-server.MyVS.property.authRealm="ldap"
  ```

  For details, see `set(1)`.
- You can define virtual server properties using the Administration Console. Select the HTTP Service component under the relevant configuration, select Virtual Servers, and select the desired virtual server. Select Add Property, enter the property name and value, check the enable box, and select Save. For details and a complete list of virtual server properties, click the Help button in the Administration Console.

Some virtual server properties can be set for a specific web application. For details, see "sun-web-app" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

Class Loader Delegation

The Servlet specification recommends that a web application class loader look in the local class loader before delegating to its parent. To make the web application class loader follow the delegation model in the Servlet specification, set `delegate="false"` in the `class-loader` element of the `sun-web.xml` file. It's safe to do this only for a web module that does not interact with any other modules.

The default value is `delegate="true"`, which causes the web application class loader to delegate in the same manner as the other class loaders. Use `delegate="true"` for a web application that accesses EJB components or that acts as a web service client or endpoint. For details about `sun-web.xml`, see *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

For a number of packages, including `java.*` and `javax.*`, symbol resolution is always delegated to the parent class loader regardless of the `delegate` setting. This prevents applications from overriding core Java runtime classes or changing the API versions of specifications that are part of the Java EE platform.

For general information about class loaders, see Chapter 2, "Class Loaders."

Using the `default-web.xml` File

You can use the `default-web.xml` file to define features such as filters and security constraints that apply to all web applications.

For example, directory listings are disabled by default for added security. To enable directory listings, in your domain's `default-web.xml` file, search for the definition of the servlet whose `servlet-name` is equal to `default`, and set the value of the `init-param` named `listings` to `true`. Then redeploy your web application if it has already been deployed, or restart the server.

```
<init-param>
   <param-name>listings</param-name>
   <param-value>true</param-value>
</init-param>
```

If `listings` is set to `true`, you can also determine how directory listings are sorted. Set the value of the `init-param` named `sortedBy` to `NAME`, `SIZE`, or `LAST_MODIFIED`. Then redeploy your web application if it has already been deployed, or restart the server.

```
<init-param>
   <param-name>sortedBy</param-name>
   <param-value>LAST_MODIFIED</param-value>
</init-param>
```

The `mime-mapping` elements in `default-web.xml` are global and inherited by all web applications. You can override these mappings or define your own using `mime-mapping` elements in your web application's `web.xml` file. For more information about `mime-mapping` elements, see the Servlet specification.

You can use the Administration Console to edit the `default-web.xml` file. For details, click the Help button in the Administration Console. As an alternative, you can edit the file directly using the following steps.

▼ To Use the `default-web.xml` File

1. **Place the JAR file for the filter, security constraint, or other feature in the *domain-dir*`/lib` directory.**
2. **Edit the *domain-dir*`/config/default-web.xml` file to refer to the JAR file.**
3. **Restart the server.**

Configuring Logging and Monitoring in the Web Container

For information about configuring logging and monitoring in the web container using the Administration Console, click the Help button in the Administration Console. Logging and Monitor tabs are accessible from the Application Server page.

Header Management

In all Editions of the Enterprise Server, the `Enumeration` from `request.getHeaders()` contains multiple elements (one element per request header) instead of a single, aggregated value.

The header names used in `HttpServletResponse.add`*XXX*`Header()` and `HttpServletResponse.set`*XXX*`Header()` are returned as they were created.

Configuring Valves and Catalina Listeners

You can configure custom valves and Catalina listeners for web modules or virtual servers by defining properties. A valve class must implement the `org.apache.catalina.Valve` interface from Tomcat or previous Enterprise Server releases, or the `org.glassfish.web.valve.GlassFishValve` interface from the current Enterprise Server release. A listener class for a virtual server must implement the `org.apache.catalina.ContainerListener` or `org.apache.catalina.LifecycleListener` interface. A listener class for a web module must implement the `org.apache.catalina.ContainerListener`, `org.apache.catalina.LifecycleListener`, or `org.apache.catalina.InstanceListener` interface.

In the `sun-web.xml` file, valve and listener properties for a web module look like this:

```
<sun-web-app ...>
   ...
   <property name="valve_1" value="org.glassfish.extension.Valve"/>
   <property name="listener_1" value="org.glassfish.extension.MyLifecycleListener"/>
</sun-web-app>
```

You can define these same properties for a virtual server. For more information, see "Virtual Server Properties" on page 161.

Alternate Document Roots

An alternate document root (docroot) allows a web application to serve requests for certain resources from outside its own docroot, based on whether those requests match one (or more) of the URI patterns of the web application's alternate docroots.

To specify an alternate docroot for a web application or a virtual server, use the `alternatedocroot_`*n* property, where *n* is a positive integer that allows specification of more than one. This property can be a subelement of a `sun-web-app` element in the `sun-web.xml` file or a virtual server property. For more information about these elements, see "sun-web-app" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide* or .

A virtual server's alternate docroots are considered only if a request does not map to any of the web modules deployed on that virtual server. A web module's alternate docroots are considered only once a request has been mapped to that web module.

If a request matches an alternate docroot's URI pattern, it is mapped to the alternate docroot by appending the request URI (minus the web application's context root) to the alternate docroot's physical location (directory). If a request matches multiple URI patterns, the alternate docroot is determined according to the following precedence order:

- Exact match
- Longest path match
- Extension match

For example, the following properties specify three `sun-web.xml` docroots. The URI pattern of the first alternate docroot uses an exact match, whereas the URI patterns of the second and third alternate docroots use extension and longest path prefix matches, respectively.

```
<property name="alternatedocroot_1" value="from=/my.jpg dir=/srv/images/jpg"/>
<property name="alternatedocroot_2" value="from=*.jpg dir=/srv/images/jpg"/>
<property name="alternatedocroot_3" value="from=/jpg/* dir=/src/images"/>
```

The `value` of each alternate docroot has two components: The first component, `from`, specifies the alternate docroot's URI pattern, and the second component, `dir`, specifies the alternate docroot's physical location (directory).

Suppose the above examples belong to a web application deployed at `http://company22.com/myapp`. The first alternate docroot maps any requests with this URL:

```
http://company22.com/myapp/my.jpg
```

To this resource:

```
/svr/images/jpg/my.jpg
```

The second alternate docroot maps any requests with a `*.jpg` suffix, such as:

```
http://company22.com/myapp/*.jpg
```

To this physical location:

```
/svr/images/jpg
```

The third alternate docroot maps any requests whose URI starts with `/myapp/jpg/`, such as:

```
http://company22.com/myapp/jpg/*
```

To the same directory as the second alternate docroot.

For example, the second alternate docroot maps this request:

```
http://company22.com/myapp/abc/def/my.jpg
```

To:

```
/srv/images/jpg/abc/def/my.jpg
```

The third alternate docroot maps:

```
http://company22.com/myapp/jpg/abc/resource
```

To:

```
/srv/images/jpg/abc/resource
```

If a request does not match any of the target web application's alternate docroots, or if the target web application does not specify any alternate docroots, the request is served from the web application's standard docroot, as usual.

Using a context.xml File

You can define a `context.xml` file for all web applications, for web applications assigned to a specific virtual server, or for a specific web application.

To define a global `context.xml` file, place the file in the *domain-dir*`/config` directory and name it `context.xml`.

Use the `contextXmlDefault` property to specify the name and the location, relative to *domain-dir*, of the `context.xml` file for a specific virtual server. Specify this property in one of the following ways:

- In the Administration Console, open the HTTP Service component under the relevant configuration. Open the Virtual Servers component and scroll down to the bottom of the page. Enter `contextXmlDefault` as the property name and the path and file name relative to *domain-dir* as the property value.
- Use the `asadmin create-virtual-server` command. For example:

  ```
  asadmin create-virtual-server --property contextXmlDefault=config/vs1ctx.xml vs1
  ```
- Use the `asadmin set` command for an existing virtual server. For example:

```
asadmin set server-config.http-service.virtual-server.vs1.property.contextXmlDefault=config/myctx.xml
```

To define a `context.xml` file for a specific web application, place the file in the `META-INF` directory and name it `context.xml`.

For more information about virtual server properties, see "Virtual Server Properties" on page 161. For more information about the `context.xml` file, see The Context Container (`http://tomcat.apache.org/tomcat-5.5-doc/config/context.html`). Context parameters, environment entries, and resource definitions in `context.xml` are supported in the Enterprise Server.

Enabling WebDav

To enable WebDav in the Enterprise Server, you edit the `web.xml` and `sun-web.xml` files as follows.

First, enable the WebDav servlet in your `web.xml` file:

```
<servlet>
   <servlet-name>webdav</servlet-name>
   <servlet-class>org.apache.catalina.servlets.WebdavServlet</servlet-class>
   <init-param>
      <param-name>debug</param-name>
      <param-value>0</param-value>
   </init-param>
   <init-param>
      <param-name>listings</param-name>
      <param-value>true</param-value>
   </init-param>
   <init-param>
      <param-name>readonly</param-name>
      <param-value>false</param-value>
   </init-param>
</servlet>
```

Then define the servlet mapping associated with your WebDav servlet in your `web.xml` file:

```
<servlet-mapping>
   <servlet-name>webdav</servlet-name>
   <url-pattern>/webdav/*</url-pattern>
</servlet-mapping>
```

To protect the WebDav servlet so other users can't modify it, add a security constraint in your `web.xml` file:

```
<security-constraint>
   <web-resource-collection>
      <web-resource-name>Login Resources</web-resource-name>
      <url-pattern>/webdav/*</url-pattern>
   </web-resource-collection>
   <auth-constraint>
      <role-name>Admin</role-name>
   </auth-constraint>
   <user-data-constraint>
      <transport-guarantee>NONE</transport-guarantee>
   </user-data-constraint>
   <login-config>
```

```
      <auth-method>BASIC</auth-method>
      <realm-name>default</realm-name>
   </login-config>
   <security-role>
      <role-name>Admin</role-name>
   </security-role>
</security-constraint>
```

Then define a security role mapping in your `sun-web.xml` file:

```
<security-role-mapping>
   <role-name>Admin</role-name>
   <group-name>Admin</group-name>
</security-role-mapping>
```

If you are using the `file` realm, create a user and password. For example:

```
asadmin create-file-user --groups Admin --authrealmname default admin
```

Enable the security manager as described in "Enabling and Disabling the Security Manager" on page 91.

You can now use any WebDav client by connecting to the WebDav servlet URL, which has this format:

`http://`*host*`:`*port*`/`*context-root*`/webdav/`*file*

For example:

```
http://localhost:80/glassfish-webdav/webdav/index.html
```

You can add the WebDav servlet to your `default-web.xml` file to enable it for all applications, but you can't set up a security role mapping to protect it.

Using SSI

To enable SSI (server-side includes) processing for a specific web module, add the `SSIServlet` to your `web.xml` file as follows:

```
<web-app>
   <servlet>
      <servlet-name>ssi</servlet-name>
      <servlet-class>org.apache.catalina.ssi.SSIServlet</servlet-class>
   </servlet>
   ...
   <servlet-mapping>
      <servlet-name>ssi</servlet-name>
      <url-pattern>*.shtml</url-pattern>
   </servlet-mapping>
   ...
   <mime-mapping>
```

```
      <extension>shtml</extension>
      <mime-type>text/html</mime-type>
   </mime-mapping>
</web-app>
```

To enable SSI processing for all web modules, un-comment the corresponding sections in the `default-web.xml` file.

If the `mime-mapping` is not specified in `web.xml`, Enterprise Server attempts to determine the MIME type from `default-web.xml` or the operating system default.

You can configure the following `init-param` values for the `SSIServlet`.

TABLE 8–3 `SSIServlet` `init-param` Values

init-param	Type	Default	Description
`buffered`	`boolean`	`false`	Specifies whether the output should be buffered.
`debug`	`int`	`0` (for no debugging)	Specifies the debugging level.
`expires`	`Long`	`Expires` header in HTTP response not set	Specifies the expiration time in seconds.
`inputEncoding`	`String`	operating system encoding	Specifies encoding for the SSI input if there is no URL content encoding specified.
`isVirtualWebappRelative`	`boolean`	`false` (relative to the given SSI file)	Specifies whether the virtual path of the `#include` directive is relative to the `content-root`.
`outputEncoding`	`String`	UTF-8	Specifies encoding for the SSI output.

For more information about SSI, see `http://httpd.apache.org/docs/2.2/mod/mod_include.html`.

Using CGI

To enable CGI (common gateway interface) processing for a specific web module, add the `CGIServlet` to your `web.xml` file as follows:

```
<web-app>
   <servlet>
      <servlet-name>cgi</servlet-name>
      <servlet-class>org.apache.catalina.servlets.CGIServlet</servlet-class>
   </servlet>
   ...
   <servlet-mapping>
```

```
      <servlet-name>cgi</servlet-name>
      <url-pattern>/cgi-bin/*</url-pattern>
   </servlet-mapping>
</web-app>
```

To enable CGI processing for all web modules, un-comment the corresponding sections in the `default-web.xml` file.

Package the CGI program under the `cgiPathPrefix`. The default `cgiPathPrefix` is `WEB-INF/cgi`. For security, it is highly recommended that the contents and binaries of CGI programs be prohibited from direct viewing or download. For information about hiding directory listings, see "Using the `default-web.xml` File" on page 162.

Invoke the CGI program using a URL of the following format:

`http://`*host*`:8080/`*context-root*`/cgi-bin/`*cgi-name*

For example:

```
http://localhost:8080/mycontext/cgi-bin/hello
```

You can configure the following `init-param` values for the `CGIServlet`.

TABLE 8–4 `CGIServlet` `init-param` Values

init-param	Type	Default	Description
`cgiPathPrefix`	`String`	`WEB-INF/cgi`	Specifies the subdirectory containing the CGI programs.
`debug`	`int`	`0` (for no debugging)	Specifies the debugging level.
`executable`	`String`	`perl`	Specifies the executable for running the CGI script.
`parameterEncoding`	`String`	`System.getProperty ("file.encoding", "UTF-8")`	Specifies the parameter's encoding.
`passShellEnvironment`	`boolean`	`false`	Specifies whether to pass shell environment properties to the CGI program.

To work with a native executable, do the following:

1. Set the value of the `init-param` named `executable` to an empty `String` in the `web.xml` file.
2. Make sure the executable has its executable bits set correctly.
3. Use directory deployment to deploy the web module. Do not deploy it as a WAR file, because the executable bit information is lost during the process of `jar` and `unjar`. For more information about directory deployment, see the *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

◆ ◆ ◆ CHAPTER 9

Using Enterprise JavaBeans Technology

This chapter describes how Enterprise JavaBeans (EJB) technology is supported in the Sun GlassFish Enterprise Server. This chapter addresses the following topics:

For general information about enterprise beans, see Part IV, "Enterprise Beans," in *The Java EE 6 Tutorial, Volume I*.

Note – The Web Profile of the Enterprise Server supports the EJB 3.1 Lite specification, which allows enterprise beans within web applications, among other features. The full Enterprise Server supports the entire EJB 3.1 specification. For details, see JSR 318 (`http://jcp.org/en/jsr/detail?id=318`).

The Enterprise Server is backward compatible with 1.1, 2.0, 2.1, and 3.0 enterprise beans. However, to take advantage of version 3.1 features, you should develop new beans as 3.1 enterprise beans.

Value Added Features

The Enterprise Server provides a number of value additions that relate to EJB development. These capabilities are discussed in the following sections. References to more in-depth material are included.

Read-Only Beans

Another feature that the Enterprise Server provides is the *read-only bean*, an EJB 2.1 entity bean that is never modified by an EJB client. Read-only beans avoid database updates completely.

Note – Read-only beans are specific to the Enterprise Server and are not part of the Enterprise JavaBeans Specification, v2.1. Use of this feature for an EJB 2.1 bean results in a non-portable application.

To make an EJB 3.0 entity read-only, use `@Column` annotations to mark its columns `insertable=false` and `updatable=false`.

A read-only bean can be used to cache a database entry that is frequently accessed but rarely updated (externally by other beans). When the data that is cached by a read-only bean is updated by another bean, the read-only bean can be notified to refresh its cached data.

The Enterprise Server provides a number of ways by which a read-only bean's state can be refreshed. By setting the `refresh-period-in-seconds` element in the `sun-ejb-jar.xml` file and the `trans-attribute` element (or `@TransactionAttribute` annotation) in the `ejb-jar.xml` file, it is easy to configure a read-only bean that is one of the following:

- Always refreshed
- Periodically refreshed
- Never refreshed
- Programmatically refreshed

Read-only beans are best suited for situations where the underlying data never changes, or changes infrequently. For further information and usage guidelines, see "Using Read-Only Beans" on page 178.

The `pass-by-reference` Element

The `pass-by-reference` element in the `sun-ejb-jar.xml` file allows you to specify the parameter passing semantics for colocated remote EJB invocations. This is an opportunity to improve performance. However, use of this feature results in non-portable applications. See "pass-by-reference" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

Pooling and Caching

The EJB container of the Enterprise Server pools anonymous instances (message-driven beans, stateless session beans, and entity beans) to reduce the overhead of creating and destroying objects. The EJB container maintains the free pool for each bean that is deployed. Bean instances in the free pool have no identity (that is, no primary key associated) and are used to serve method calls. The free beans are also used to serve all methods for stateless session beans.

Bean instances in the free pool transition from a Pooled state to a Cached state after `ejbCreate` and the business methods run. The size and behavior of each pool is controlled using pool-related properties in the EJB container or the `sun-ejb-jar.xml` file.

In addition, the Enterprise Server supports a number of tunable parameters that can control the number of "stateful" instances (stateful session beans and entity beans) cached as well as the duration they are cached. Multiple bean instances that refer to the same database row in a table can be cached. The EJB container maintains a cache for each bean that is deployed.

To achieve scalability, the container selectively evicts some bean instances from the cache, usually when cache overflows. These evicted bean instances return to the free bean pool. The size and behavior of each cache can be controlled using the cache-related properties in the EJB container or the `sun-ejb-jar.xml` file.

Pooling and caching parameters for the `sun-ejb-jar.xml` file are described in "bean-cache" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

Pooling Parameters

One of the most important parameters of Enterprise Server pooling is `steady-pool-size`. When `steady-pool-size` is set to greater than 0, the container not only pre-populates the bean pool with the specified number of beans, but also attempts to ensure that this number of beans is always available in the free pool. This ensures that there are enough beans in the ready to serve state to process user requests.

This parameter does not necessarily guarantee that no more than `steady-pool-size` instances exist at a given time. It only governs the number of instances that are pooled over a long period of time. For example, suppose an idle stateless session container has a fully-populated pool with a `steady-pool-size` of `10`. If 20 concurrent requests arrive for the EJB component, the container creates 10 additional instances to satisfy the burst of requests. The advantage of this is that it prevents the container from blocking any of the incoming requests. However, if the activity dies down to 10 or fewer concurrent requests, the additional 10 instances are discarded.

Another parameter, `pool-idle-timeout-in-seconds`, allows the administrator to specify the amount of time a bean instance can be idle in the pool. When `pool-idle-timeout-in-seconds` is set to greater than 0, the container removes or destroys any bean instance that is idle for this specified duration.

Caching Parameters

Enterprise Server provides a way that completely avoids caching of entity beans, using commit option C. Commit option C is particularly useful if beans are accessed in large number but very rarely reused. For additional information, refer to "Commit Options" on page 186.

The Enterprise Server caches can be either bounded or unbounded. *Bounded caches* have limits on the number of beans that they can hold beyond which beans are passivated. For stateful session beans, there are three ways (LRU, NRU and FIFO) of picking victim beans when cache overflow occurs. Caches can also passivate beans that are idle (not accessed for a specified duration).

Bean-Level Container-Managed Transaction Timeouts

The default transaction timeout for the domain is specified using the Transaction Timeout setting of the Transaction Service. A transaction started by the container must commit (or rollback) within this time, regardless of whether the transaction is suspended (and resumed), or the transaction is marked for rollback.

To override this timeout for an individual bean, use the optional `cmt-timeout-in-seconds` element in `sun-ejb-jar.xml`. The default value, `0`, specifies that the default Transaction Service timeout is used. The value of `cmt-timeout-in-seconds` is used for all methods in the bean that start a new container-managed transaction. This value is *not* used if the bean joins a client transaction.

Priority Based Scheduling of Remote Bean Invocations

You can create multiple thread pools, each having its own work queues. An optional element in the `sun-ejb-jar.xml` file, `use-thread-pool-id`, specifies the thread pool that processes the requests for the bean. The bean must have a remote interface, or `use-thread-pool-id` is ignored. You can create different thread pools and specify the appropriate thread pool ID for a bean that requires a quick response time. If there is no such thread pool configured or if the element is absent, the default thread pool is used.

Immediate Flushing

Normally, all entity bean updates within a transaction are batched and executed at the end of the transaction. The only exception is the database flush that precedes execution of a finder or select query.

Since a transaction often spans many method calls, you might want to find out if the updates made by a method succeeded or failed immediately after method execution. To force a flush at the end of a method's execution, use the `flush-at-end-of-method` element in the

`sun-ejb-jar.xml` file. Only non-finder methods in an entity bean can be flush-enabled. (For an EJB 2.1 bean, these methods must be in the Local, Local Home, Remote, or Remote Home interface.) See "flush-at-end-of-method" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

Upon completion of the method, the EJB container updates the database. Any exception thrown by the underlying data store is wrapped as follows:

- If the method that triggered the flush is a `create` method, the exception is wrapped with `CreateException`.
- If the method that triggered the flush is a `remove` method, the exception is wrapped with `RemoveException`.
- For all other methods, the exception is wrapped with `EJBException`.

All normal end-of-transaction database synchronization steps occur regardless of whether the database has been flushed during the transaction.

EJB Timer Service

The EJB Timer Service uses a database to store persistent information about EJB timers.

The EJB Timer Service in Enterprise Server is preconfigured to use an embedded version of the Java DB database.

The timer service is automatically enabled when you deploy an application or module that uses it.

You can verify that the timer service is running by accessing the following URL:

```
http://localhost:8080/ejb-timer-service-app/timer
```

The EJB Timer Service configuration can store persistent timer information in any database supported by the Enterprise Server for persistence. For a list of the JDBC drivers currently supported by the Enterprise Server, see the *Sun GlassFish Enterprise Server v3 Release Notes*. For configurations of supported and other drivers, see "Configuration Specifics for JDBC Drivers" in *Sun GlassFish Enterprise Server v3 Administration Guide*.

To change the database used by the EJB Timer Service, set the EJB Timer Service's Timer DataSource setting to a valid JDBC resource. If the EJB Timer Service has already been started, you must also create the timer database table. DDL files are located in *as-install*`/lib/install/databases`.

Using the EJB Timer Service is equivalent to interacting with a single JDBC resource manager. If an EJB component or application accesses a database either directly through JDBC or indirectly (for example, through an entity bean's persistence mechanism), and also interacts with the EJB Timer Service, its data source must be configured with an XA JDBC driver.

You can change the following EJB Timer Service settings. You must restart the server for the changes to take effect.

- Minimum Delivery Interval - Specifies the minimum time in milliseconds before an expiration for a particular timer can occur. This guards against extremely small timer increments that can overload the server. The default is `1000`.
- Maximum Redeliveries - Specifies the maximum number of times the EJB timer service attempts to redeliver a timer expiration due for exception or rollback. The default is `1`.
- Redelivery Interval - Specifies how long in milliseconds the EJB timer service waits after a failed `ejbTimeout` delivery before attempting a redelivery. The default is `5000`.
- Timer DataSource - Specifies the database used by the EJB Timer Service. The default is `jdbc/__TimerPool`.

For information about the `asadmin list-timers` command, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Using Session Beans

This section provides guidelines for creating session beans in the Enterprise Server environment. This section addresses the following topics:

- "About the Session Bean Containers" on page 176
- "Session Bean Restrictions and Optimizations" on page 178

Information on session beans is contained in the Enterprise JavaBeans Specification, v3.1.

About the Session Bean Containers

Like an entity bean, a session bean can access a database through Java Database Connectivity (JDBC) calls. A session bean can also provide transaction settings. These transaction settings and JDBC calls are referenced by the session bean's container, allowing it to participate in transactions managed by the container.

A container managing stateless session beans has a different charter from a container managing stateful session beans. This section addresses the following topics:

- "Stateless Container" on page 176
- "Stateful Container" on page 177

Stateless Container

The *stateless container* manages stateless session beans, which, by definition, do not carry client-specific states. All session beans (of a particular type) are considered equal.

A stateless session bean container uses a bean pool to service requests. The Enterprise Server specific deployment descriptor file, `sun-ejb-jar.xml`, contains the properties that define the pool:

- `steady-pool-size`
- `resize-quantity`
- `max-pool-size`
- `pool-idle-timeout-in-seconds`

For more information about `sun-ejb-jar.xml`, see "The sun-ejb-jar.xml File" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

The Enterprise Server provides the `wscompile` and `wsdeploy` tools to help you implement a web service endpoint as a stateless session bean. For more information about these tools, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Stateful Container

The *stateful container* manages the stateful session beans, which, by definition, carry the client-specific state. There is a one-to-one relationship between the client and the stateful session beans. At creation, each stateful session bean (SFSB) is given a unique session ID that is used to access the session bean so that an instance of a stateful session bean is accessed by a single client only.

Stateful session beans are managed using cache. The size and behavior of stateful session beans cache are controlled by specifying the following `sun-ejb-jar.xml` parameters:

- `max-cache-size`
- `resize-quantity`
- `cache-idle-timeout-in-seconds`
- `removal-timeout-in-seconds`
- `victim-selection-policy`

The `max-cache-size` element specifies the maximum number of session beans that are held in cache. If the cache overflows (when the number of beans exceeds `max-cache-size`), the container then passivates some beans or writes out the serialized state of the bean into a file. The directory in which the file is created is obtained from the EJB container using the configuration APIs.

For more information about `sun-ejb-jar.xml`, see "The sun-ejb-jar.xml File" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

The passivated beans are stored on the file system. The Session Store Location setting in the EJB container allows the administrator to specify the directory where passivated beans are stored. By default, passivated stateful session beans are stored in application-specific subdirectories created under *domain-dir*`/session-store`.

Note – Make sure the `delete` option is set in the `server.policy` file, or expired file-based sessions might not be deleted properly. For more information about `server.policy`, see "The `server.policy` File" on page 88.

The Session Store Location setting also determines where the session state is persisted.

Session Bean Restrictions and Optimizations

This section discusses restrictions on developing session beans and provides some optimization guidelines.

Optimizing Session Bean Performance

For stateful session beans, colocating the stateful beans with their clients so that the client and bean are executing in the same process address space improves performance.

Restricting Transactions

The following restrictions on transactions are enforced by the container and must be observed as session beans are developed:

- A session bean can participate in, at most, a single transaction at a time.
- If a session bean is participating in a transaction, a client cannot invoke a method on the bean such that the `trans-attribute` element (or `@TransactionAttribute` annotation) in the `ejb-jar.xml` file would cause the container to execute the method in a different or unspecified transaction context or an exception is thrown.
- If a session bean instance is participating in a transaction, a client cannot invoke the `remove` method on the session object's home or business interface object, or an exception is thrown.

Using Read-Only Beans

A *read-only bean* is an EJB 2.1 entity bean that is never modified by an EJB client. The data that a read-only bean represents can be updated externally by other enterprise beans, or by other means, such as direct database updates.

Note – Read-only beans are specific to the Enterprise Server and are not part of the Enterprise JavaBeans Specification, v2.1. Use of this feature for an EJB 2.1 bean results in a non-portable application.

To make an EJB 3.0 entity bean read-only, use `@Column` annotations to mark its columns `insertable=false` and `updatable=false`.

Read-only beans are best suited for situations where the underlying data never changes, or changes infrequently. The following topics are addressed in this section:

Read-Only Bean Characteristics and Life Cycle

Read-only beans are best suited for situations where the underlying data never changes, or changes infrequently. For example, a read-only bean can be used to represent a stock quote for a particular company, which is updated externally. In such a case, using a regular entity bean might incur the burden of calling `ejbStore`, which can be avoided by using a read-only bean.

Read-only beans have the following characteristics:

- Only entity beans can be read-only beans.
- Either bean-managed persistence (BMP) or container-managed persistence (CMP) is allowed. If CMP is used, do not create the database schema during deployment. Instead, work with your database administrator to populate the data into the tables. See Chapter 10, "Using Container-Managed Persistence."
- Only container-managed transactions are allowed; read-only beans cannot start their own transactions.
- Read-only beans don't update any bean state.
- `ejbStore` is never called by the container.
- `ejbLoad` is called only when a transactional method is called or when the bean is initially created (in the cache), or at regular intervals controlled by the bean's `refresh-period-in-seconds` element in the `sun-ejb-jar.xml` file.
- The home interface can have any number of find methods. The return type of the find methods must be the primary key for the same bean type (or a collection of primary keys).
- If the data that the bean represents can change, then `refresh-period-in-seconds` must be set to refresh the beans at regular intervals. `ejbLoad` is called at this regular interval.

A read-only bean comes into existence using the appropriate find methods.

Read-only beans are cached and have the same cache properties as entity beans. When a read-only bean is selected as a victim to make room in the cache, `ejbPassivate` is called and the bean is returned to the free pool. When in the free pool, the bean has no identity and is used only to serve any finder requests.

Read-only beans are bound to the naming service like regular read-write entity beans, and clients can look up read-only beans the same way read-write entity beans are looked up.

Read-Only Bean Good Practices

For best results, follow these guidelines when developing read-only beans:

- Avoid having any `create` or `remove` methods in the home interface.
- Use any of the valid EJB 2.1 transaction attributes for the `trans-attribute` element.

  The reason for having `TX_SUPPORTED` is to allow reading uncommitted data in the same transaction. Also, the transaction attributes can be used to force `ejbLoad`.

Refreshing Read-Only Beans

There are several ways of refreshing read-only beans as addressed in the following sections:

- "Invoking a Transactional Method" on page 180
- "Refreshing Periodically" on page 180
- "Refreshing Programmatically" on page 181

Invoking a Transactional Method

Invoking any transactional method invokes `ejbLoad`.

Refreshing Periodically

Use the `refresh-period-in-seconds` element in the `sun-ejb-jar.xml` file to refresh a read-only bean periodically.

- If the value specified in `refresh-period-in-seconds` is zero or not specified, which is the default, the bean is never refreshed (unless a transactional method is accessed).
- If the value is greater than zero, the bean is refreshed at the rate specified.

Note – This is the only way to refresh the bean state if the data can be modified external to the Enterprise Server.

By default, a single timer is used for all instances of a read-only bean. When that timer fires, all bean instances are marked as expired and are refreshed from the database the next time they are used.

Use the `-Dcom.sun.ejb.containers.readonly.relative.refresh.mode=true` flag to refresh each bean instance independently upon access if its refresh period has expired. The default is `false`. Note that each instance still has the same refresh period. This additional level of granularity can improve the performance of read-only beans that do not need to be refreshed at the same time.

To set this flag, use the `asadmin create-jvm-options` command. For example:

```
asadmin create-jvm-options -Dcom.sun.ejb.containers.readonly.relative.refresh.mode=true
```

Refreshing Programmatically

Typically, beans that update any data that is cached by read-only beans need to notify the read-only beans to refresh their state. Use `ReadOnlyBeanNotifier` to force the refresh of read-only beans.

To do this, invoke the following methods on the `ReadOnlyBeanNotifier` bean:

```
public interface ReadOnlyBeanNotifier extends java.rmi.Remote {
   refresh(Object PrimaryKey) throws RemoteException;
 }
```

The implementation of the `ReadOnlyBeanNotifier` interface is provided by the container. The bean looks up `ReadOnlyBeanNotifier` using a fragment of code such as the following example:

```
com.sun.appserv.ejb.ReadOnlyBeanHelper helper =
  new com.sun.appserv.ejb.ReadOnlyBeanHelper();
com.sun.appserv.ejb.ReadOnlyBeanNotifier notifier =
  helper.getReadOnlyBeanNotifier("java:comp/env/ejb/ReadOnlyCustomer");
notifier.refresh(PrimaryKey);
```

For a local read-only bean notifier, the lookup has this modification:

```
helper.getReadOnlyBeanLocalNotifier("java:comp/env/ejb/LocalReadOnlyCustomer");
```

Beans that update any data that is cached by read-only beans need to call the `refresh` methods. The next (non-transactional) call to the read-only bean invokes `ejbLoad`.

For Javadoc tool pages relevant to read-only beans, go to `https://glassfish.dev.java.net/nonav/docs/v3/api/` and click on the `com.sun.appserv.ejb` package.

Deploying Read-Only Beans

Read-only beans are deployed in the same manner as other entity beans. However, in the entry for the bean in the `sun-ejb-jar.xml` file, the `is-read-only-bean` element must be set to true. That is:

```
<is-read-only-bean>true</is-read-only-bean>
```

Also, the `refresh-period-in-seconds` element in the `sun-ejb-jar.xml` file can be set to some value that specifies the rate at which the bean is refreshed. If this element is missing, no refresh occurs.

All requests in the same transaction context are routed to the same read-only bean instance. Set the `allow-concurrent-access` element to either `true` (to allow concurrent accesses) or `false` (to serialize concurrent access to the same read-only bean). The default is `false`.

For further information on these elements, refer to "The sun-ejb-jar.xml File" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

Using Message-Driven Beans

This section describes message-driven beans and explains the requirements for creating them in the Enterprise Server environment. This section contains the following topics:

Message-Driven Bean Configuration

This section addresses the following configuration topics:

Connection Factory and Destination

A message-driven bean is a client to a Connector inbound resource adapter. The message-driven bean container uses the JMS service integrated into the Enterprise Server for message-driven beans that are JMS clients. JMS clients use JMS Connection Factory- and Destination-administered objects. A JMS Connection Factory administered object is a resource manager Connection Factory object that is used to create connections to the JMS provider.

The `mdb-connection-factory` element in the `sun-ejb-jar.xml` file for a message-driven bean specifies the connection factory that creates the container connection to the JMS provider.

The `jndi-name` element of the `ejb` element in the `sun-ejb-jar.xml` file specifies the JNDI name of the administered object for the JMS `Queue` or `Topic` destination that is associated with the message-driven bean.

Message-Driven Bean Pool

The container manages a pool of message-driven beans for the concurrent processing of a stream of messages. The `sun-ejb-jar.xml` file contains the elements that define the pool (that is, the `bean-pool` element):

- `steady-pool-size`
- `resize-quantity`
- `max-pool-size`
- `pool-idle-timeout-in-seconds`

For more information about `sun-ejb-jar.xml`, see "The sun-ejb-jar.xml File" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

Domain-Level Settings

You can control the following domain-level message-driven bean settings in the EJB container:

- Initial and Minimum Pool Size - Specifies the initial and minimum number of beans maintained in the pool. The default is `0`.
- Maximum Pool Size - Specifies the maximum number of beans that can be created to satisfy client requests. The default is 32.
- Pool Resize Quantity - Specifies the number of beans to be created if a request arrives when the pool is empty (subject to the Initial and Minimum Pool Size), or the number of beans to remove if idle for more than the Idle Timeout. The default is `8`.
- Idle Timeout - Specifies the maximum time in seconds that a bean can remain idle in the pool. After this amount of time, the bean is destroyed. The default is `600` (10 minutes). A value of `0` means a bean can remain idle indefinitely.

For information on monitoring message-driven beans, click the Help button in the Administration Console. The Monitor tab is accessible from the Application Server page.

Note – Running monitoring when it is not needed might impact performance, so you might choose to turn monitoring off when it is not in use. For details, see Chapter 8, "Administering the Monitoring Service," in *Sun GlassFish Enterprise Server v3 Administration Guide*.

Message-Driven Bean Restrictions and Optimizations

This section discusses the following restrictions and performance optimizations that pertain to developing message-driven beans:

Pool Tuning and Monitoring

The message-driven bean pool is also a pool of threads, with each message-driven bean instance in the pool associating with a server session, and each server session associating with a thread. Therefore, a large pool size also means a high number of threads, which impacts performance and server resources.

When configuring message-driven bean pool properties, make sure to consider factors such as message arrival rate and pattern, `onMessage` method processing time, overall server resources (threads, memory, and so on), and any concurrency requirements and limitations from other resources that the message-driven bean accesses.

When tuning performance and resource usage, make sure to consider potential JMS provider properties for the connection factory used by the container (the `mdb-connection-factory` element in the `sun-ejb-jar.xml` file). For example, you can tune the Sun GlassFish Message Queue flow control related properties for connection factory in situations where the message incoming rate is much higher than `max-pool-size` can handle.

Refer to Chapter 8, "Administering the Monitoring Service," in *Sun GlassFish Enterprise Server v3 Administration Guide* for information on how to get message-driven bean pool statistics.

The `onMessage` Runtime Exception

Message-driven beans, like other well-behaved MessageListeners, should not, in general, throw runtime exceptions. If a message-driven bean's `onMessage` method encounters a system-level exception or error that does not allow the method to successfully complete, the Enterprise JavaBeans Specification, v3.0 provides the following guidelines:

- If the bean method encounters a runtime exception or error, it should simply propagate the error from the bean method to the container.
- If the bean method performs an operation that results in a checked exception that the bean method cannot recover, the bean method should throw the `javax.ejb.EJBException` that wraps the original exception.
- Any other unexpected error conditions should be reported using `javax.ejb.EJBException` (`javax.ejb.EJBException` is a subclass of `java.lang.RuntimeException`).

Under container-managed transaction demarcation, upon receiving a runtime exception from a message-driven bean's `onMessage` method, the container rolls back the container-started transaction and the message is redelivered. This is because the message delivery itself is part of the container-started transaction. By default, the Enterprise Server container closes the container's connection to the JMS provider when the first runtime exception is received from a message-driven bean instance's `onMessage` method. This avoids potential message redelivery looping and protects server resources if the message-driven bean's `onMessage` method continues misbehaving. To change this default container behavior, use the `cmt-max-runtime-exceptions` property of the MDB container. Here is an example `asadmin set` command that sets this property:

```
asadmin set server-config.mdb-container.property.cmt-max-runtime-exceptions="5"
```

For more information about the `asadmin set` command, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

The `cmt-max-runtime-exceptions` property specifies the maximum number of runtime exceptions allowed from a message-driven bean's `onMessage` method before the container starts to close the container's connection to the message source. By default this value is 1; -1 disables this container protection.

A message-driven bean's `onMessage` method can use the `javax.jms.Message getJMSRedelivered` method to check whether a received message is a redelivered message.

Note – The `cmt-max-runtime-exceptions` property is deprecated.

Handling Transactions With Enterprise Beans

This section describes the transaction support built into the Enterprise JavaBeans programming model for the Enterprise Server.

As a developer, you can write an application that updates data in multiple databases distributed across multiple sites. The site might use EJB servers from different vendors. This section provides overview information on the following topics:

Flat Transactions

The Enterprise JavaBeans Specification, v3.0 requires support for flat (as opposed to nested) transactions. In a flat transaction, each transaction is decoupled from and independent of other transactions in the system. Another transaction cannot start in the same thread until the current transaction ends.

Flat transactions are the most prevalent model and are supported by most commercial database systems. Although nested transactions offer a finer granularity of control over transactions, they are supported by far fewer commercial database systems.

Global and Local Transactions

Understanding the distinction between global and local transactions is crucial in understanding the Enterprise Server support for transactions. See "Transaction Scope" on page 266.

Both local and global transactions are demarcated using the `javax.transaction.UserTransaction` interface, which the client must use. Local transactions bypass the transaction manager and are faster. For more information, see "The Transaction Manager, the Transaction Synchronization Registry, and `UserTransaction`" on page 267.

Commit Options

The EJB protocol is designed to give the container the flexibility to select the disposition of the instance state at the time a transaction is committed. This allows the container to best manage caching an entity object's state and associating an entity object identity with the EJB instances.

There are three commit-time options:

- **Option A** – The container caches a ready instance between transactions. The container ensures that the instance has exclusive access to the state of the object in persistent storage.

  In this case, the container does *not* have to synchronize the instance's state from the persistent storage at the beginning of the next transaction.

  Note – Commit option A is not supported for this Enterprise Server release.

- **Option B** – The container caches a ready instance between transactions, but the container does *not* ensure that the instance has exclusive access to the state of the object in persistent storage. This is the default.

  In this case, the container must synchronize the instance's state by invoking `ejbLoad` from persistent storage at the beginning of the next transaction.

- **Option C** – The container does *not* cache a ready instance between transactions, but instead returns the instance to the pool of available instances after a transaction has completed.

  The life cycle for every business method invocation under commit option C looks like this.

  ```
  ejbActivate   ejbLoad   business method   ejbStore   ejbPassivate
  ```

  If there is more than one transactional client concurrently accessing the same entity, the first client gets the ready instance and subsequent concurrent clients get new instances from the pool.

The Enterprise Server deployment descriptor has an element, `commit-option`, that specifies the commit option to be used. Based on the specified commit option, the appropriate handler is instantiated.

Administration and Monitoring

An administrator can control a number of domain-level Transaction Service settings. For details, see "Configuring the Transaction Service" on page 267.

The Transaction Timeout setting can be overridden by a bean. See "Bean-Level Container-Managed Transaction Timeouts" on page 174.

In addition, the administrator can monitor transactions using statistics from the transaction manager that provide information on such activities as the number of transactions completed, rolled back, or recovered since server startup, and transactions presently being processed.

For information on administering and monitoring transactions, select the Transaction Service component under the relevant configuration in the Administration Console and click the Help button. Also see Chapter 21, "Administering Transactions," in *Sun GlassFish Enterprise Server v3 Administration Guide*.

◆ ◆ ◆ CHAPTER 10

Using Container-Managed Persistence

This chapter contains information on how EJB 2.1 container-managed persistence (CMP) works in the Sun GlassFish Enterprise Server in the following topics:

Note – The Web Profile of the Enterprise Server supports the EJB 3.1 Lite specification, which allows enterprise beans within web applications, among other features. The full Enterprise Server supports the entire EJB 3.1 specification. For details, see JSR 318 (`http://jcp.org/en/jsr/detail?id=318`).

Enterprise Server Support for CMP

Enterprise Server support for EJB 2.1 CMP beans includes:

- Full support for the J2EE v1.4 specification's CMP model. Extensive information on CMP is contained in chapters 10, 11, and 14 of the Enterprise JavaBeans Specification, v2.1. This includes the following:
  - Support for commit options B and C for transactions. See "Commit Options" on page 186.
  - The primary key class must be a subclass of `java.lang.Object`. This ensures portability, and is noted because some vendors allow primitive types (such as `int`) to be used as the primary key class.

- The Enterprise Server CMP implementation, which provides the following:
  - An Object/Relational (O/R) mapping tool that creates XML deployment descriptors for EJB JAR files that contain beans that use CMP.
  - Support for compound (multi-column) primary keys.
  - Support for sophisticated custom finder methods.
  - Standards-based query language (EJB QL).
  - CMP runtime support. See "Configuring the CMP Resource" on page 202.
- Enterprise Server performance-related features, including the following:
  - Version column consistency checking
  - Relationship prefetching
  - Read-Only Beans

  For details, see "Performance-Related Features" on page 202.

CMP Mapping

Implementation for entity beans that use CMP is mostly a matter of mapping CMP fields and CMR fields (relationships) to the database. This section addresses the following topics:

Mapping Capabilities

Mapping refers to the ability to tie an object-based model to a relational model of data, usually the schema of a relational database. The CMP implementation provides the ability to tie a set of interrelated beans containing data and associated behaviors to the schema. This object representation of the database becomes part of the Java application. You can also customize this mapping to optimize these beans for the particular needs of an application. The result is a single data model through which both persistent database information and regular transient program data are accessed.

The mapping capabilities provided by the Enterprise Server include:

- Mapping a CMP bean to one or more tables
- Mapping CMP fields to one or more columns
- Mapping CMP fields to different column types
- Mapping tables with compound primary keys
- Mapping tables with unknown primary keys
- Mapping CMP relationships to foreign keys

- Mapping tables with overlapping primary and foreign keys

The Mapping Deployment Descriptor File

Each module with CMP beans must have the following files:

- `ejb-jar.xml` – The J2EE standard file for assembling enterprise beans. For a detailed description, see the Enterprise JavaBeans Specification, v2.1.
- `sun-ejb-jar.xml` – The Enterprise Server standard file for assembling enterprise beans. For a detailed description, see "The sun-ejb-jar.xml File" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.
- `sun-cmp-mappings.xml` – The *mapping deployment descriptor file*, which describes the mapping of CMP beans to tables in a database. For a detailed description, see "The sun-cmp-mappings.xml File" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

The `sun-cmp-mappings.xml` file can be automatically generated and does not have to exist prior to deployment. For details, see "Generation Options for CMP" on page 197.

The `sun-cmp-mappings.xml` file maps CMP fields and CMR fields (relationships) to the database. A primary table must be selected for each CMP bean, and optionally, multiple secondary tables. CMP fields are mapped to columns in either the primary or secondary table(s). CMR fields are mapped to pairs of column lists (normally, column lists are the lists of columns associated with primary and foreign keys).

Note – Table names in databases can be case-sensitive. Make sure that the table names in the `sun-cmp-mappings.xml` file match the names in the database.

Relationships should always be mapped to the primary key field(s) of the related table.

The `sun-cmp-mappings.xml` file conforms to the `sun-cmp-mapping_1_2.dtd` file and is packaged with the user-defined bean classes in the EJB JAR file under the `META-INF` directory.

The Enterprise Server creates the mappings in the `sun-cmp-mappings.xml` file automatically during deployment if the file is not present.

To map the fields and relationships of your entity beans manually, edit the `sun-cmp-mappings.xml` deployment descriptor. Only do this if you are proficient in editing XML.

The mapping information is developed in conjunction with the database schema (`.dbschema`) file, which can be automatically captured when you deploy the bean (see "Automatic Database Schema Capture" on page 201). You can manually generate the schema using the `capture-schema` utility ("Using the `capture-schema` Utility" on page 201).

Mapping Considerations

This section addresses the following topics:

The data types used in automatic schema generation are also suggested for manual mapping. These data types are described in "Supported Data Types for CMP" on page 195.

Join Tables and Relationships

Use of join tables in the database schema is supported for all types of relationships, not just many-to-many relationships. For general information about relationships, see section 10.3.7 of the Enterprise JavaBeans Specification, v2.1.

Automatic Primary Key Generation

The Enterprise Server supports automatic primary key generation for EJB 1.1, 2.0, and 2.1 CMP beans. To specify automatic primary key generation, give the `prim-key-class` element in the `ejb-jar-xml` file the value `java.lang.Object`. CMP beans with automatically generated primary keys can participate in relationships with other CMP beans. The Enterprise Server does not support database-generated primary key values.

If the database schema is created during deployment, the Enterprise Server creates the schema with the primary key column, then generates unique values for the primary key column at runtime.

If the database schema is not created during deployment, the primary key column in the mapped table must be of type `NUMERIC` with a precision of 19 or more, and must not be mapped to any CMP field. The Enterprise Server generates unique values for the primary key column at runtime.

Fixed Length CHAR Primary Keys

If an existing database table has a primary key column in which the values vary in length, but the type is `CHAR` instead of `VARCHAR`, the Enterprise Server automatically trims any extra spaces when retrieving primary key values. It is not a good practice to use a fixed length `CHAR` column as a primary key. Use this feature with schemas that cannot be changed, such as a schema inherited from a legacy application.

Managed Fields

A managed field is a CMP or CMR field that is mapped to the same database column as another CMP or CMR field. CMP fields mapped to the same column and CMR fields mapped to exactly the same column lists always have the same value in memory. For CMR fields that share only a subset of their mapped columns, changes to the columns affect the relationship fields in memory differently. Basically, the Enterprise Server always tries to keep the state of the objects in memory synchronized with the database.

A managed field can have any `fetched-with` subelement. If the `fetched-with` subelement is `<default/>`, the `-DAllowManagedFieldsInDefaultFetchGroup` flag must be set to `true`. See "Default Fetch Group Flags" on page 204 and "fetched-with" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

BLOB Support

Binary Large Object (BLOB) is a data type used to store values that do not correspond to other types such as numbers, strings, or dates. Java fields whose types implement `java.io.Serializable` or are represented as `byte[]` can be stored as BLOBs.

If a CMP field is defined as `Serializable`, it is serialized into a `byte[]` before being stored in the database. Similarly, the value fetched from the database is deserialized. However, if a CMP field is defined as `byte[]`, it is stored directly instead of being serialized and deserialized when stored and fetched, respectively.

To enable BLOB support in the Enterprise Server environment, define a CMP field of type `byte[]` or a user-defined type that implements the `java.io.Serializable` interface. If you map the CMP bean to an existing database schema, map the field to a column of type BLOB.

To use BLOB or CLOB data types larger than 4 KB for CMP using the Inet Oraxo JDBC Driver for Oracle Databases, you must set the `streamstolob` property value to `true`.

For a list of the JDBC drivers currently supported by the Enterprise Server, see the *Sun GlassFish Enterprise Server v3 Release Notes*. For configurations of supported and other drivers, see "Configuration Specifics for JDBC Drivers" in *Sun GlassFish Enterprise Server v3 Administration Guide*.

For automatic mapping, you might need to change the default BLOB column length for the generated schema using the `schema-generator-properties` element in `sun-ejb-jar.xml`. See your database vendor documentation to determine whether you need to specify the length. For example:

```
<schema-generator-properties>
   <property>
      <name>Employee.voiceGreeting.jdbc-type</name>
      <value>BLOB</value>
   </property>
   <property>
```

```
      <name>Employee.voiceGreeting.jdbc-maximum-length</name>
      <value>10240</value>
   </property>
   ...
</schema-generator-properties>
```

CLOB Support

Character Large Object (CLOB) is a data type used to store and retrieve very long text fields. CLOBs translate into long strings.

To enable CLOB support in the Enterprise Server environment, define a CMP field of type `java.lang.String`. If you map the CMP bean to an existing database schema, map the field to a column of type CLOB.

To use BLOB or CLOB data types larger than 4 KB for CMP using the Inet Oraxo JDBC Driver for Oracle Databases, you must set the `streamstolob` property value to `true`.

For a list of the JDBC drivers currently supported by the Enterprise Server, see the *Sun GlassFish Enterprise Server v3 Release Notes*. For configurations of supported and other drivers, see "Configuration Specifics for JDBC Drivers" in *Sun GlassFish Enterprise Server v3 Administration Guide*.

For automatic mapping, you might need to change the default CLOB column length for the generated schema using the `schema-generator-properties` element in `sun-ejb-jar.xml`. See your database vendor documentation to determine whether you need to specify the length. For example:

```
<schema-generator-properties>
   <property>
      <name>Employee.resume.jdbc-type</name>
      <value>CLOB</value>
   </property>
   <property>
      <name>Employee.resume.jdbc-maximum-length</name>
      <value>10240</value>
   </property>
   ...
</schema-generator-properties>
```

Automatic Schema Generation for CMP

The automatic schema generation feature provided in the Enterprise Server defines database tables based on the fields in entity beans and the relationships between the fields. This insulates developers from many of the database related aspects of development, allowing them to focus on entity bean development. The resulting schema is usable as-is or can be given to a database administrator for tuning with respect to performance, security, and so on.

This section addresses the following topics:

- "Supported Data Types for CMP" on page 195
- "Generation Options for CMP" on page 197

Note – Automatic schema generation is supported on an all-or-none basis: it expects that no tables exist in the database before it is executed. It is not intended to be used as a tool to generate extra tables or constraints.

Deployment won't fail if all tables are not created, and undeployment won't fail if not all tables are dropped. This is done to allow you to investigate the problem and fix it manually. You should not rely on the partially created database schema to be correct for running the application.

Supported Data Types for CMP

CMP supports a set of JDBC data types that are used in mapping Java data fields to SQL types. Supported JDBC data types are as follows: BIGINT, BIT, BLOB, CHAR, CLOB, DATE, DECIMAL, DOUBLE, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TIME, TIMESTAMP, TINYINT, VARCHAR.

The following table contains the mappings of Java types to JDBC types when automatic mapping is used.

TABLE 10–1 Java Type to JDBC Type Mappings for CMP

Java Type	JDBC Type	Nullability
`boolean`	`BIT`	No
`java.lang.Boolean`	`BIT`	Yes
`byte`	`TINYINT`	No
`java.lang.Byte`	`TINYINT`	Yes
`double`	`DOUBLE`	No
`java.lang.Double`	`DOUBLE`	Yes
`float`	`REAL`	No
`java.lang.Float`	`REAL`	Yes
`int`	`INTEGER`	No
`java.lang.Integer`	`INTEGER`	Yes
`long`	`BIGINT`	No
`java.lang.Long`	`BIGINT`	Yes

TABLE 10–1 Java Type to JDBC Type Mappings for CMP *(Continued)*

Java Type	JDBC Type	Nullability
`short`	`SMALLINT`	No
`java.lang.Short`	`SMALLINT`	Yes
`java.math.BigDecimal`	`DECIMAL`	Yes
`java.math.BigInteger`	`DECIMAL`	Yes
`char`	`CHAR`	No
`java.lang.Character`	`CHAR`	Yes
`java.lang.String`	`VARCHAR` or `CLOB`	Yes
`Serializable`	`BLOB`	Yes
`byte[]`	`BLOB`	Yes
`java.util.Date`	`DATE` (Oracle only) `TIMESTAMP` (all other databases)	Yes
`java.sql.Date`	`DATE`	Yes
`java.sql.Time`	`TIME`	Yes
`java.sql.Timestamp`	`TIMESTAMP`	Yes

Note – Java types assigned to CMP fields must be restricted to Java primitive types, Java `Serializable` types, `java.util.Date`, `java.sql.Date`, `java.sql.Time`, or `java.sql.Timestamp`. An entity bean local interface type (or a collection of such) can be the type of a CMR field.

The following table contains the mappings of JDBC types to database vendor-specific types when automatic mapping is used. For a list of the JDBC drivers currently supported by the Enterprise Server, see the *Sun GlassFish Enterprise Server v3 Release Notes*. For configurations of supported and other drivers, see "Configuration Specifics for JDBC Drivers" in *Sun GlassFish Enterprise Server v3 Administration Guide*.

TABLE 10–2 Mappings of JDBC Types to Database Vendor Specific Types for CMP

JDBC Type	Java DB, Derby, CloudScape	Oracle	DB2	Sybase ASE 12.5	MS-SQL Server
`BIT`	`SMALLINT`	`SMALLINT`	`SMALLINT`	`TINYINT`	`BIT`
`TINYINT`	`SMALLINT`	`SMALLINT`	`SMALLINT`	`TINYINT`	`TINYINT`

TABLE 10–2 Mappings of JDBC Types to Database Vendor Specific Types for CMP *(Continued)*

JDBC Type	Java DB, Derby, CloudScape	Oracle	DB2	Sybase ASE 12.5	MS-SQL Server
SMALLINT	SMALLINT	SMALLINT	SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER	INTEGER	INTEGER	INTEGER
BIGINT	BIGINT	NUMBER	BIGINT	NUMERIC	NUMERIC
REAL	REAL	REAL	FLOAT	FLOAT	REAL
DOUBLE	DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE	DOUBLE PRECISION	FLOAT
DECIMAL(p,s)	DECIMAL(p,s)	NUMBER(p,s)	DECIMAL(p,s)	DECIMAL(p,s)	DECIMAL(p,s)
VARCHAR	VARCHAR	VARCHAR2	VARCHAR	VARCHAR	VARCHAR
DATE	DATE	DATE	DATE	DATETIME	DATETIME
TIME	TIME	DATE	TIME	DATETIME	DATETIME
TIMESTAMP	TIMESTAMP	TIMESTAMP(9)	TIMESTAMP	DATETIME	DATETIME
BLOB	BLOB	BLOB	BLOB	IMAGE	IMAGE
CLOB	CLOB	CLOB	CLOB	TEXT	NTEXT

Generation Options for CMP

Deployment descriptor elements or `asadmin` command line options can control automatic schema generation by the following:

- Creating tables during deployment
- Dropping tables during undeployment
- Dropping and creating tables during redeployment
- Specifying the database vendor
- Specifying that table names are unique
- Specifying type mappings for individual CMP fields

Note – Before using these options, make sure you have a properly configured CMP resource. See "Configuring the CMP Resource" on page 202.

For a read-only bean, do not create the database schema during deployment. Instead, work with your database administrator to populate the data into the tables. See "Using Read-Only Beans" on page 178.

Automatic schema generation is not supported for beans with version column consistency checking. Instead, work with your database administrator to create the schema and add the required triggers. See "Version Column Consistency Checking" on page 202.

The following optional data subelements of the `cmp-resource` element in the `sun-ejb-jar.xml` file control the automatic creation of database tables at deployment. For more information about the `cmp-resource` element, see "cmp-resource" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide* and "Configuring the CMP Resource" on page 202.

TABLE 10–3 The `sun-ejb-jar.xml` Generation Elements

Element	Default	Description
`create-tables-at-deploy`	`false`	If `true`, causes database tables to be created for beans that are automatically mapped by the EJB container. If `false`, does not create tables.
`drop-tables-at-undeploy`	`false`	If `true`, causes database tables that were automatically created when the bean(s) were last deployed to be dropped when the bean(s) are undeployed. If `false`, does not drop tables.
`database-vendor-name`	none	Specifies the name of the database vendor for which tables are created. Allowed values are `javadb`, `db2`, `mssql`, `oracle`, `postgresql`, `pointbase`, `derby` (also for CloudScape), and `sybase`, case-insensitive. If no value is specified, a connection is made to the resource specified by the `jndi-name` subelement of the `cmp-resource` element in the `sun-ejb-jar.xml` file, and the database vendor name is read. If the connection cannot be established, or if the value is not recognized, SQL-92 compliance is presumed.

TABLE 10–3 The `sun-ejb-jar.xml` Generation Elements *(Continued)*

Element	Default	Description
`schema-generator-properties`	none	Specifies field-specific column attributes in `property` subelements. Each property name is of the following format: *bean-name*.*field-name*.*attribute* For example: `Employee.firstName.jdbc-type` Also allows you to set the `use-unique-table-names` property. If `true`, this property specifies that generated table names are unique within each application server domain. The default is `false`. For further information and an example, see "schema-generator-properties" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

The following options of the `asadmin deploy` or `asadmin deploydir` command control the automatic creation of database tables at deployment.

TABLE 10–4 The `asadmin deploy` and `asadmin deploydir` Generation Options for CMP

Option	Default	Description
`--createtables`	none	If `true`, causes database tables to be created for beans that need them. If `false`, does not create tables. If not specified, the value of the `create-tables-at-deploy` attribute in `sun-ejb-jar.xml` is used.
`--dropandcreatetables`	none	If `true`, and if tables were automatically created when this application was last deployed, tables from the earlier deployment are dropped and fresh ones are created. If `true`, and if tables were *not* automatically created when this application was last deployed, no attempt is made to drop any tables. If tables with the same names as those that would have been automatically created are found, the deployment proceeds, but a warning indicates that tables could not be created. If `false`, settings of `create-tables-at-deploy` or `drop-tables-at-undeploy` in the `sun-ejb-jar.xml` file are overridden.
`--uniquetablenames`	none	If `true`, specifies that table names are unique within each application server domain. If not specified, the value of the `use-unique-table-names` property in `sun-ejb-jar.xml` is used.

TABLE 10–4 The `asadmin deploy` and `asadmin deploydir` Generation Options for CMP *(Continued)*

Option	Default	Description
`--dbvendorname`	none	Specifies the name of the database vendor for which tables are created. Allowed values are `javadb`, `db2`, `mssql`, `oracle`, `postgresql`, `pointbase`, `derby` (also for CloudScape), and `sybase`, case-insensitive. If not specified, the value of the `database-vendor-name` attribute in `sun-ejb-jar.xml` is used. If no value is specified, a connection is made to the resource specified by the `jndi-name` subelement of the `cmp-resource` element in the `sun-ejb-jar.xml` file, and the database vendor name is read. If the connection cannot be established, or if the value is not recognized, SQL-92 compliance is presumed.

If one or more of the beans in the module are manually mapped and you use any of the `asadmin deploy` or `asadmin deploydir` options, the deployment is not harmed in any way, but the options have no effect, and a warning is written to the server log.

The following options of the `asadmin undeploy` command control the automatic removal of database tables at undeployment.

TABLE 10–5 The `asadmin undeploy` Generation Options for CMP

Option	Default	Description
`--droptables`	none	If `true`, causes database tables that were automatically created when the bean(s) were last deployed to be dropped when the bean(s) are undeployed. If `false`, does not drop tables. If not specified, the value of the `drop-tables-at-undeploy` attribute in `sun-ejb-jar.xml` is used.

For more information about the `asadmin deploy`, `asadmin deploydir`, and `asadmin undeploy` commands, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

When command line and `sun-ejb-jar.xml` options are both specified, the `asadmin` options take precedence.

The Ant tasks `sun-appserv-deploy` and `sun-appserv-undeploy` are equivalent to `asadmin deploy` and `asadmin undeploy`, respectively. These Ant tasks also override the `sun-ejb-jar.xml` options. For details, see Chapter 3, "Using Ant with Enterprise Server."

Schema Capture

This section addresses the following topics:

Automatic Database Schema Capture

You can configure a CMP bean in Enterprise Server to automatically capture the database metadata and save it in a `.dbschema` file during deployment. If the `sun-cmp-mappings.xml` file contains an empty `<schema/>` entry, the `cmp-resource` entry in the `sun-ejb-jar.xml` file is used to get a connection to the database, and automatic generation of the schema is performed.

Note – Before capturing the database schema automatically, make sure you have a properly configured CMP resource. See "Configuring the CMP Resource" on page 202.

Using the `capture-schema` Utility

You can use the `capture-schema` command to manually generate the database metadata (`.dbschema`) file. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

The `capture-schema` utility does *not* modify the schema in any way. Its only purpose is to provide the persistence engine with information about the structure of the database (the schema).

Keep the following in mind when using the `capture-schema` command:

- The name of a `.dbschema` file must be unique across all deployed modules in a domain.
- If more than one schema is accessible for the schema user, more than one table with the same name might be captured if the `-schemaname` parameter of `capture-schema` is not set.
- The schema name must be upper case.
- Table names in databases are case-sensitive. Make sure that the table name matches the name in the database.
- PostgreSQL databases internally convert all names to lower case. Before running the `capture-schema` command on a PostgreSQL database, make sure table and column names are lower case in the `sun-cmp-mappings.xml` file.
- An Oracle database user running the `capture-schema` command needs ANALYZE ANY TABLE privileges if that user does not own the schema. These privileges are granted to the user by the database administrator.

Configuring the CMP Resource

An EJB module that contains CMP beans requires the JNDI name of a JDBC resource in the `jndi-name` subelement of the `cmp-resource` element in the `sun-ejb-jar.xml` file. Set `PersistenceManagerFactory` properties as properties of the `cmp-resource` element in the `sun-ejb-jar.xml` file. See "cmp-resource" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

In the Administration Console, open the Resources component, then select JDBC. Click the Help button in the Administration Console for information on creating a new JDBC resource.

For a list of the JDBC drivers currently supported by the Enterprise Server, see the *Sun GlassFish Enterprise Server v3 Release Notes*. For configurations of supported and other drivers, see "Configuration Specifics for JDBC Drivers" in *Sun GlassFish Enterprise Server v3 Administration Guide*.

For example, if the JDBC resource has the JNDI name `jdbc/MyDatabase`, set the CMP resource in the `sun-ejb-jar.xml` file as follows:

```
<cmp-resource>
   <jndi-name>jdbc/MyDatabase</jndi-name>
</cmp-resource>
```

Performance-Related Features

The Enterprise Server provides the following features to enhance performance or allow more fine-grained data checking. These features are supported only for entity beans with container managed persistence.

- "Version Column Consistency Checking" on page 202
- "Relationship Prefetching" on page 203
- "Read-Only Beans" on page 204
- "Default Fetch Group Flags" on page 204

Note – Use of any of these features results in a non-portable application.

Version Column Consistency Checking

The version consistency feature saves the bean state at first transactional access and caches it between transactions. The state is copied from the cache instead of being read from the database. The bean state is verified by primary key and version column values at flush for custom queries (for dirty instances only) and at commit (for clean and dirty instances).

▼ To Use Version Consistency

1. **Create the version column in the primary table.**
2. **Give the version column a numeric data type.**
3. **Provide appropriate update triggers on the version column.**

   These triggers must increment the version column on each update of the specified row.
4. **Specify the version column.**

   This is specified in the `check-version-of-accessed-instances` subelement of the `consistency` element in the `sun-cmp-mappings.xml` file. See "consistency" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.
5. **Map the CMP bean to an existing schema.**

   Automatic schema generation is not supported for beans with version column consistency checking. Instead, work with your database administrator to create the schema and add the required triggers.

Relationship Prefetching

In many cases when an entity bean's state is fetched from the database, its relationship fields are always accessed in the same transaction. Relationship prefetching saves database round trips by fetching data for an entity bean and those beans referenced by its CMR fields in a single database round trip.

To enable relationship prefetching for a CMR field, use the `default` subelement of the `fetched-with` element in the `sun-cmp-mappings.xml` file. By default, these CMR fields are prefetched whenever `findByPrimaryKey` or a custom finder is executed for the entity, or when the entity is navigated to from a relationship. (Recursive prefetching is not supported, because it does not usually enhance performance.) See "fetched-with" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

To disable prefetching for specific custom finders, use the `prefetch-disabled` element in the `sun-ejb-jar.xml` file. See "prefetch-disabled" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

Multilevel relationship prefetching is supported for CMP 2.1 entity beans. To enable multilevel relationship prefetching, set the following property using the `asadmin create-jvm-options` command:

```
asadmin create-jvm-options -Dcom.sun.jdo.spi.persistence.support.sqlstore.MULTILEVEL_PREFETCH=true
```

Read-Only Beans

Another feature that the Enterprise Server provides is the *read-only bean*, an entity bean that is never modified by an EJB client. Read-only beans avoid database updates completely.

Note – Read-only beans are specific to the Enterprise Server and are not part of the Enterprise JavaBeans Specification, v2.1. Use of this feature for an EJB 2.1 bean results in a non-portable application.

A read-only bean can be used to cache a database entry that is frequently accessed but rarely updated (externally by other beans). When the data that is cached by a read-only bean is updated by another bean, the read-only bean can be notified to refresh its cached data.

The Enterprise Server provides a number of ways by which a read-only bean's state can be refreshed. By setting the `refresh-period-in-seconds` element in the `sun-ejb-jar.xml` file and the `trans-attribute` element (or `@TransactionAttribute` annotation) in the `ejb-jar.xml` file, it is easy to configure a read-only bean that is one of the following:

- Always refreshed
- Periodically refreshed
- Never refreshed
- Programmatically refreshed

Access to CMR fields of read-only beans is not supported. Deployment will succeed, but an exception will be thrown at runtime if a get or set method is invoked.

Read-only beans are best suited for situations where the underlying data never changes, or changes infrequently. For further information and usage guidelines, see "Using Read-Only Beans" on page 178.

Default Fetch Group Flags

Using the following flags can improve performance.

Setting `-DAllowManagedFieldsInDefaultFetchGroup=true` allows CMP fields that by default cannot be placed into the default fetch group to be loaded along with all other fields that are fetched when the CMP state is loaded into memory. These could be multiple fields mapped to the same column in the database table, for example, an instance field and a CMR. By default this flag is set to `false`.

For additional information, see "level" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

Setting `-DAllowMediatedWriteInDefaultFetchGroup` specifies how updated CMP fields are written back to the database. If the flag is `false`, all fields in the CMP bean are written back to the database if at least one field in the default fetch group has been changed in a transaction. If the flag is `true`, only fields modified by the bean are written back to the database. Specifying `true` can improve performance, particularly on database tables with many columns that have not been updated. By default this flag is set to `false`.

To set one of these flags, use the `asadmin create-jvm-options` command. For example:

```
asadmin create-jvm-options -DAllowManagedFieldsInDefaultFetchGroup=true
```

Configuring Queries for 1.1 Finders

This section contains the following topics:

About JDOQL Queries

The Enterprise JavaBeans Specification, v1.1 does not specify the format of the finder method description. The Enterprise Server uses an extension of Java Data Objects Query Language (JDOQL) queries to implement finder and selector methods. You can specify the following elements of the underlying JDOQL query:

- **Filter expression** - A Java-like expression that specifies a condition that each object returned by the query must satisfy. Corresponds to the WHERE clause in EJB QL.
- **Query parameter declaration** - Specifies the name and the type of one or more query input parameters. Follows the syntax for formal parameters in the Java language.
- **Query variable declaration** - Specifies the name and type of one or more query variables. Follows the syntax for local variables in the Java language. A query filter might use query variables to implement joins.
- **Query ordering declaration** - Specifies the ordering expression of the query. Corresponds to the ORDER BY clause of EJB QL.

The Enterprise Server specific deployment descriptor (`sun-ejb-jar.xml`) provides the following elements to store the EJB 1.1 finder method settings:

```
query-filter
query-params
query-variables
query-ordering
```

The bean developer uses these elements to construct a query. When the finder method that uses these elements executes, the values of these elements are used to execute a query in the database. The objects from the JDOQL query result set are converted into primary key instances to be returned by the EJB 1.1 `ejbFind` method.

The JDO specification, JSR 12 (`http://jcp.org/en/jsr/detail?id=12`), provides a comprehensive description of JDOQL. The following information summarizes the elements used to define EJB 1.1 finders.

Query Filter Expression

The filter expression is a String containing a Boolean expression evaluated for each instance of the candidate class. If the filter is not specified, it defaults to true. Rules for constructing valid expressions follow the Java language, with the following differences:

- Equality and ordering comparisons between primitives and instances of wrapper classes are valid.
- Equality and ordering comparisons of Date fields and Date parameters are valid.
- Equality and ordering comparisons of String fields and String parameters are valid.
- White space (non-printing characters space, tab, carriage return, and line feed) is a separator and is otherwise ignored.
- The following assignment operators are not supported.
  - Comparison operators such as =, +=, and so on
  - Pre- and post-increment
  - Pre- and post-decrement
- Methods, including object construction, are not supported, except for these methods.

  ```
  Collection.contains(Object o)
  Collection.isEmpty()
  String.startsWith(String s)
  String.endsWith(String e)
  ```

  In addition, the Enterprise Server supports the following nonstandard JDOQL methods.

  ```
  String.like(String pattern)
  String.like(String pattern, char escape)
  String.substring(int start, int length)
  String.indexOf(String str)
  String.indexOf(String str, int start)
  String.length()
  Math.abs(numeric n)
  Math.sqrt(double d)
  ```
- Navigation through a null-valued field, which throws a `NullPointerException`, is treated as if the sub-expression returned `false`.

Note – Comparisons between floating point values are by nature inexact. Therefore, equality comparisons (== and !=) with floating point values should be used with caution. Identifiers in the expression are considered to be in the name space of the candidate class, with the addition of declared parameters and variables. As in the Java language, `this` is a reserved word, and refers to the current instance being evaluated.

The following expressions are supported.

- Relational operators (==, !=, >, <, >=, <=)
- Boolean operators (&, &&, |, ||, ~, !)
- Arithmetic operators (+, -, *, /)
- String concatenation, only for String + String
- Parentheses to explicitly mark operator precedence
- Cast operator
- Promotion of numeric operands for comparisons and arithmetic operations

The rules for promotion follow the Java rules extended by BigDecimal, BigInteger, and numeric wrapper classes. See the numeric promotions of the Java language specification.

Query Parameters

The parameter declaration is a String containing one or more parameter type declarations separated by commas. This follows the Java syntax for method signatures.

Query Variables

The type declarations follow the Java syntax for local variable declarations.

JDOQL Examples

This section provides a few query examples.

Example 1

The following query returns all players called Michael. It defines a filter that compares the name field with a string literal:

```
name == "Michael"
```

The `finder` element of the `sun-ejb-jar.xml` file looks like this:

```
<finder>
   <method-name>findPlayerByName</method-name>
   <query-filter>name == "Michael"</query-filter>
</finder>
```

Example 2

This query returns all products in a specified price range. It defines two query parameters which are the lower and upper bound for the price: double low, double high. The filter compares the query parameters with the price field:

```
low < price && price < high
```

Query ordering is set to `price ascending`.

The `finder` element of the `sun-ejb-jar.xml` file looks like this:

```
<finder>
   <method-name>findInRange</method-name>
   <query-params>double low, double high</query-params>
   <query-filter>low &lt; price &amp;&amp; price &lt high</query-filter>
   <query-ordering>price ascending</query-ordering>
</finder>
```

Example 3

This query returns all players having a higher salary than the player with the specified name. It defines a query parameter for the name `java.lang.String name`. Furthermore, it defines a variable to which the player's salary is compared. It has the type of the persistence capable class that corresponds to the bean:

```
    mypackage.PlayerEJB_170160966_JDOState player
```

The filter compares the salary of the current player denoted by the `this` keyword with the salary of the player with the specified name:

```
    (this.salary > player.salary) && (player.name == name)
```

The `finder` element of the `sun-ejb-jar.xml` file looks like this:

```
<finder>
   <method-name>findByHigherSalary</method-name>
   <query-params>java.lang.String name</query-params>
   <query-filter>
      (this.salary &gt; player.salary) &amp;&amp; (player.name == name)
   </query-filter>
   <query-variables>
      mypackage.PlayerEJB_170160966_JDOState player
   </query-variables>
</finder>
```

CMP Restrictions and Optimizations

This section discusses restrictions and performance optimizations that pertain to using CMP.

Disabling ORDER BY Validation

EJB QL as defined in the EJB 2.1 Specification defines certain restrictions for the SELECT clause of an ORDER BY query (see section 11.2.8 ORDER BY Clause). This ensures that a query does not order by a field that is not returned by the query. By default, the EJB QL compiler checks the above restriction and throws an exception if the query does not conform.

However, some databases support SQL statements with an ORDER BY column that is not included in the SELECT clause. To disable the validation of the ORDER BY clause against the SELECT clause, set the `DISABLE_ORDERBY_VALIDATION` JVM option as follows:

```
asadmin create-jvm-options
-Dcom.sun.jdo.spi.persistence.support.ejb.ejbqlc.DISABLE_ORDERBY_VALIDATION=true
```

The `DISABLE_ORDERBY_VALIDATION` option is set to `false` by default. Setting it to `true` results in a non-portable module or application.

Setting the Heap Size on DB2

On DB2, the database configuration parameter `APPLHEAPSZ` determines the heap size. If you are using the Sun GlassFish or DataDirect database driver, set this parameter to at least `2048` for CMP. For more information, see `http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/opt/tsbp2024.htm`.

Eager Loading of Field State

By default, the EJB container loads the state for all persistent fields (excluding relationship, BLOB, and CLOB fields) before invoking the `ejbLoad` method of the abstract bean. This approach might not be optimal for entity objects with large state if most business methods require access to only parts of the state.

Use the `fetched-with` element in `sun-cmp-mappings.xml` for fields that are used infrequently. See "fetched-with" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

Restrictions on Remote Interfaces

The following restrictions apply to the remote interface of an EJB 2.1 bean that uses CMP:

- Do not expose the `get` and `set` methods for CMR fields or the persistence collection classes that are used in container-managed relationships through the remote interface of the bean.

  However, you are free to expose the `get` and `set` methods that correspond to the CMP fields of the entity bean through the bean's remote interface.
- Do not expose the container-managed collection classes that are used for relationships through the remote interface of the bean.
- Do not expose local interface types or local home interface types through the remote interface or remote home interface of the bean.

Dependent value classes can be exposed in the remote interface or remote home interface, and can be included in the client EJB JAR file.

PostgreSQL Case Insensitivity

Case-sensitive behavior cannot be achieved for PostgreSQL databases. PostgreSQL databases internally convert all names to lower case, which makes the following workarounds necessary:

- In the CMP 2.1 runtime, PostgreSQL table and column names are not quoted, which makes these names case insensitive.
- Before running the `capture-schema` command on a PostgreSQL database, make sure table and column names are lower case in the `sun-cmp-mappings.xml` file.

No Support for `lock-when-loaded` on Sybase

For EJB 2.1 beans, the `lock-when-loaded` consistency level is implemented by placing update locks on the data corresponding to a bean when the data is loaded from the database. There is no suitable mechanism available on Sybase databases to implement this feature. Therefore, the `lock-when-loaded` consistency level is not supported on Sybase databases. See "consistency" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

Sybase Finder Limitation

If a finder method with an input greater than 255 characters is executed and the primary key column is mapped to a VARCHAR column, Sybase attempts to convert type VARCHAR to type TEXT and generates the following error:

```
com.sybase.jdbc2.jdbc.SybSQLException: Implicit conversion from datatype
'TEXT' to 'VARCHAR' is not allowed.  Use the CONVERT function to run this
query.
```

To avoid this error, make sure the finder method input is less than 255 characters.

Date and Time Fields

If a field type is a Java date or time type (`java.util.Date`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`), make sure that the field value exactly matches the value in the database.

For example, the following code uses a `java.sql.Date` type as a primary key field:

```
java.sql.Date myDate = new java.sql.Date(System.currentTimeMillis())
BeanA.create(myDate, ...);
```

For some databases, this code results in only the year, month, and date portion of the field value being stored in the database. Later if the client tries to find this bean by primary key as follows, the bean is not found in the database because the value does not match the one that is stored in the database.

```
myBean = BeanA.findByPrimaryKey(myDate);
```

Similar problems can happen if the database truncates the timestamp value while storing it, or if a custom query has a date or time value comparison in its WHERE clause.

For automatic mapping to an Oracle database, fields of type `java.util.Date`, `java.sql.Date`, and `java.sql.Time` are mapped to Oracle's DATE data type. Fields of type `java.sql.Timestamp` are mapped to Oracle's `TIMESTAMP(9)` data type.

Set `RECURSIVE_TRIGGERS` to `false` on MSSQL

For version consistency triggers on MSSQL, the property `RECURSIVE_TRIGGERS` must be set to `false`, which is the default. If set to `true`, triggers throw a `java.sql.SQLException`.

Set this property as follows:

```
EXEC sp_dboption 'database-name', 'recursive triggers', 'FALSE'
go
```

You can test this property as follows:

```
SELECT DATABASEPROPERTYEX('database-name', 'IsRecursiveTriggersEnabled')
go
```

MySQL Database Restrictions

The following restrictions apply when you use a MySQL database with the Enterprise Server for persistence.

- MySQL treats `int1` and `int2` as reserved words. If you want to define `int1` and `int2` as fields in your table, use ‘`int1`‘ and ‘`int2`‘ field names in your SQL file.
- When `VARCHAR` fields get truncated, a warning is displayed instead of an error. To get an error message, start the MySQL database in strict SQL mode.
- The order of fields in a foreign key index must match the order in the explicitly created index on the primary table.
- The `CREATE TABLE` syntax in the SQL file must end with the following line.

  ```
  )  Engine=InnoDB;
  ```

  `InnoDB` provides MySQL with a transaction-safe (ACID compliant) storage engine having commit, rollback, and crash recovery capabilities.
- For a `FLOAT` type field, the correct precision must be defined. By default, MySQL uses four bytes to store a `FLOAT` type that does not have an explicit precision definition. For example, this causes a number such as 12345.67890123 to be rounded off to 12345.7 during an `INSERT`. To prevent this, specify `FLOAT(10,2)` in the DDL file, which forces the database to use an eight-byte double-precision column. For more information, see `http://dev.mysql.com/doc/mysql/en/numeric-types.html`.
- To use `||` as the string concatenation symbol, start the MySQL server with the `--sql-mode="PIPES_AS_CONCAT"` option. For more information, see `http://dev.mysql.com/doc/refman/5.0/en/server-sql-mode.html` and `http://dev.mysql.com/doc/mysql/en/ansi-mode.html`.
- MySQL always starts a new connection when `autoCommit==true` is set. This ensures that each SQL statement forms a single transaction on its own. If you try to rollback or commit an SQL statement, you get an error message.

  ```
  javax.transaction.SystemException: java.sql.SQLException:
  Can't call rollback when autocommit=true
  ```

  ```
  javax.transaction.SystemException: java.sql.SQLException:
  Error open transaction is not closed
  ```

  To resolve this issue, add `relaxAutoCommit=true` to the JDBC URL. For more information, see `http://forums.mysql.com/read.php?39,31326,31404`.
- Change the trigger create format from the following:

```
CREATE TRIGGER T_UNKNOWNPKVC1
BEFORE UPDATE ON UNKNOWNPKVC1
FOR EACH ROW
        WHEN (NEW.VERSION = OLD.VERSION)
BEGIN
        :NEW.VERSION := :OLD.VERSION + 1;
END;
/
```

To the following:

```
DELIMITER |
CREATE TRIGGER T_UNKNOWNPKVC1
BEFORE UPDATE ON UNKNOWNPKVC1
FOR EACH ROW
        WHEN (NEW.VERSION = OLD.VERSION)
BEGIN
        :NEW.VERSION := :OLD.VERSION + 1;
END
|
DELIMITER ;
```

For more information, see `http://dev.mysql.com/doc/mysql/en/create-trigger.html`.

- MySQL does not allow a `DELETE` on a row that contains a reference to itself. Here is an example that illustrates the issue.

```
create table EMPLOYEE (
        empId   int         NOT NULL,
        salary  float(25,2) NULL,
        mgrId   int         NULL,
        PRIMARY KEY (empId),
        FOREIGN KEY (mgrId) REFERENCES EMPLOYEE (empId)
        ) ENGINE=InnoDB;

        insert into Employee values (1, 1234.34, 1);
        delete from Employee where empId = 1;
```

This example fails with the following error message.

```
ERROR 1217 (23000): Cannot delete or update a parent row:
a foreign key constraint fails
```

To resolve this issue, change the table creation script to the following:

```
create table EMPLOYEE (
        empId   int         NOT NULL,
        salary  float(25,2) NULL,
        mgrId   int         NULL,
        PRIMARY KEY (empId),
        FOREIGN KEY (mgrId) REFERENCES EMPLOYEE (empId)
        ON DELETE SET NULL
        ) ENGINE=InnoDB;

        insert into Employee values (1, 1234.34, 1);
        delete from Employee where empId = 1;
```

This can be done only if the foreign key field is allowed to be null. For more information, see http://bugs.mysql.com/bug.php?id=12449 and http://dev.mysql.com/doc/mysql/en/innodb-foreign-key-constraints.html.

- When an SQL script has foreign key constraints defined, `capture-schema` fails to capture the table information correctly. To work around the problem, remove the constraints and then run `capture-schema`. Here is an example that illustrates the issue.

```
CREATE TABLE ADDRESSBOOKBEANTABLE (ADDRESSBOOKNAME VARCHAR(255)
    NOT NULL PRIMARY KEY,
CONNECTEDUSERS              BLOB NULL,
OWNER                       VARCHAR(256),
FK_FOR_ACCESSPRIVILEGES     VARCHAR(256),
CONSTRAINT FK_ACCESSPRIVILEGE FOREIGN KEY (FK_FOR_ACCESSPRIVILEGES)
    REFERENCES ACCESSPRIVILEGESBEANTABLE (ROOT)
) ENGINE=InnoDB;
```

To resolve this issue, change the table creation script to the following:

```
CREATE TABLE ADDRESSBOOKBEANTABLE (ADDRESSBOOKNAME VARCHAR(255)
    NOT NULL PRIMARY KEY,
CONNECTEDUSERS              BLOB NULL,
OWNER                       VARCHAR(256),
FK_FOR_ACCESSPRIVILEGES     VARCHAR(256)
) ENGINE=InnoDB;
```

◆ ◆ ◆ CHAPTER 11

Developing Java Clients

This chapter describes how to develop, assemble, and deploy Java clients in the following sections:

Note – The Web Profile of the Sun GlassFishEnterprise Server supports the EJB 3.1 Lite specification, which allows enterprise beans within web applications, among other features. The full Enterprise Server supports the entire EJB 3.1 specification. For details, see JSR 318 (`http://jcp.org/en/jsr/detail?id=318`).

Accordingly, the Application Client Container is supported only in the full Enterprise Server, not in the Web Profile.

JMS resources are supported only in the full Enterprise Server, not in the Web Profile. See Chapter 17, "Using the Java Message Service."

Introducing the Application Client Container

The Application Client Container (ACC) includes a set of Java classes, libraries, and other files that are required for and distributed with Java client programs that execute in their own Java Virtual Machine (JVM). The ACC manages the execution of Java EE application client components (application clients), which are used to access a variety of Java EE services (such as JMS resources, EJB components, web services, security, and so on.) from a JVM outside the Sun GlassFish Enterprise Server.

The ACC communicates with the Enterprise Server using RMI-IIOP protocol and manages the details of RMI-IIOP communication using the client ORB that is bundled with it. Compared to other Java EE containers, the ACC is lightweight.

For information about debugging application clients, see "Application Client Debugging" on page 69.

Note – Interoperability between application clients and Enterprise Servers running under different major versions is not supported.

ACC Security

The ACC determines when authentication is needed. This typically occurs when the client refers to an EJB component that requires authorization or when annotations in the client's `main` class trigger injection which, in turn, requires contact with the Enterprise Server's naming service. To authenticate the end user, the ACC prompts for any required information, such as a username and password. The ACC itself provides a very simple dialog box to prompt for and read these values.

The ACC integrates with the Enterprise Server's authentication system. It also supports SSL (Secure Socket Layer)/IIOP if configured and when necessary; see "Using RMI/IIOP Over SSL" on page 227.

You can provide an alternate implementation to gather authentication information, tailored to the needs of the application client. To do so, include the class to perform these duties in the application client and identify the fully-qualified name of this class in the `callback-handler` element of the `application-client.xml` descriptor for the client. The ACC uses this class instead of its default class for asking for and reading the authentication information. The class must implement the `javax.security.auth.callback.CallbackHandler` interface. See the Java EE specification, section 9.2, *Application Clients: Security*, for more details.

Application clients can use "Programmatic Login" on page 102.

ACC Naming

The client container enables the application clients to use the Java Naming and Directory Interface (JNDI) to look up Java EE services (such as JMS resources, EJB components, web services, security, and so on.) and to reference configurable parameters set at the time of deployment.

ACC Annotation

Annotation is supported for the main class and the optional callback handler class in application clients. For more information, see "Deployment Descriptors and Annotations" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

Java Web Start

Java Web Start allows your application client to be easily launched and automatically downloaded and updated. It is enabled for all application clients by default. For more information, see "Using Java Web Start" on page 220.

Application Client JAR File

In Enterprise Server v3, the appclient JAR file is smaller than in previous releases, with dependent classes in separate JAR files. When copying the appclient to another location, make sure to include the JAR files containing the dependent classes as well. You can also use the `asadmin get-client-stubs` command to retrieve the appclient and all associated application JAR files and place them in another location.

Developing Clients Using the ACC

This section describes the procedure to develop, assemble, and deploy client applications using the ACC. This section describes the following topics:

▼ To Access an EJB Component From an Application Client

1 **In your client code, reference the EJB component by using an `@EJB` annotation or by looking up the JNDI name as defined in the `ejb-jar.xml` file.**

For more information about naming and lookups, see "Accessing the Naming Context" on page 271.

2 **Define the `@EJB` annotations or the `ejb-ref` elements in the `application-client.xml` file. Define the corresponding `ejb-ref` elements in the `sun-application-client.xml` file.**

For more information on the `sun-application-client.xml` file, see "The sun-application-client.xml file" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*. For a general explanation of how to map JNDI names using reference elements, see "Mapping References" on page 278.

3 **Deploy the application client and EJB component together in an application.**

For more information on deployment, see the *Sun GlassFish Enterprise Server v3 Application Deployment Guide*. To get the client JAR file, use the `--retrieve` option of the `asadmin deploy` command.

To retrieve the stubs and ties whether or not you requested their generation during deployment, use the `asadmin get-client-stubs` command. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

4 **Ensure that the client JAR file includes the following files:**

- A Java class to access the bean.
- `application-client.xml` - (optional) Java EE application client deployment descriptor.
- `sun-application-client.xml` - (optional) Enterprise Server specific client deployment descriptor. For information on the `sun-application-client.xml` file, see "The sun-application-client.xml file" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.
- The `MANIFEST.MF` file. This file contains a reference to the `main` class, which states the complete package prefix and class name of the Java client.

5 **Prepare the client machine. This step is not needed for Java Web Start.**

If you are using the `appclient` script, package the Enterprise Server system files required to launch application clients on remote systems using the `package-appclient` script, then retrieve the application client itself using the `asadmin get-client-stubs` command.

For more information, see "Using the `package-appclient` Script" on page 227 and the *Sun GlassFish Enterprise Server v3 Reference Manual*.

6 **To access EJB components that are residing in a remote system, make the following changes to the `sun-acc.xml` file or the `appclient` script. This step is not needed for Java Web Start.**

- Define the `target-server` element's `address` and `port` attributes to reference the remote server machine and its ORB port. See "target-server" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.
- Use the `-targetserver` option of the appclient script to reference the remote server machine and its ORB port. For more information, see "Running an Application Client Using the `appclient` Script" on page 226.

To determine the ORB port on the remote server, use the `asadmin get` command. For example:

```
asadmin --host rmtsrv get server-config.iiop-service.iiop-listener.iiop-listener1.port
```

For more information about the `asadmin get` command, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

7 **Run the application client.**

See "Using Java Web Start" on page 220 or "Running an Application Client Using the `appclient` Script" on page 226.

▼ To Access a JMS Resource From an Application Client

1 **Create a JMS client.**

2 **Next, configure a JMS resource on the Enterprise Server.**

For information on configuring JMS resources, see "Creating JMS Resources: Destinations and Connection Factories" on page 284.

3 **Define the `@Resource` or `@Resources` annotations or the `resource-ref` elements in the `application-client.xml` file. Define the corresponding `resource-ref` elements in the `sun-application-client.xml` file.**

For more information on the `sun-application-client.xml` file, see "The sun-application-client.xml file" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*. For a general explanation of how to map JNDI names using reference elements, see "Mapping References" on page 278.

4 **Ensure that the client JAR file includes the following files:**

- A Java class to access the resource.
- `application-client.xml` - (optional) Java EE application client deployment descriptor.
- `sun-application-client.xml` - (optional) Enterprise Server specific client deployment descriptor. For information on the `sun-application-client.xml` file, see "The sun-application-client.xml file" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.
- The `MANIFEST.MF` file. This file contains a reference to the `main` class, which states the complete package prefix and class name of the Java client.

5 **Prepare the client machine. This step is not needed for Java Web Start.**

If you are using the `appclient` script, package the Enterprise Server system files required to launch application clients on remote systems using the `package-appclient` script, then retrieve the application client itself using the `asadmin get-client-stubs` command.

For more information, see "Using the `package-appclient` Script" on page 227 and the *Sun GlassFish Enterprise Server v3 Reference Manual*.

6 **Run the application client.**

See "Using Java Web Start" on page 220 or "Running an Application Client Using the `appclient` Script" on page 226.

Using Java Web Start

Java Web Start allows your application client to be easily launched and automatically downloaded and updated. General information about Java Web Start is available at `http://java.sun.com/products/javawebstart/reference/api/index.html`.

Java Web Start is discussed in the following topics:

Enabling and Disabling Java Web Start

Java Web Start is enabled for all application clients by default.

The application developer or deployer can specify that Java Web Start is always disabled for an application client by setting the value of the `eligible` element to `false` in the `sun-application-client.xml` file. See the *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

The Enterprise Server administrator can disable Java Web Start for a previously deployed eligible application client using the `asadmin set` command.

To disable Java Web Start for all eligible application clients in an application, use the following command:

```
asadmin set domain1.applications.j2ee-application.app-name.java-web-start-enabled="false"
```

To disable Java Web Start for a stand-alone eligible application client, use the following command:

```
asadmin set domain1.applications.appclient-module.module-name.java-web-start-enabled="false"
```

Setting `java-web-start-enabled="true"` re-enables Java Web Start for an eligible application client. For more information about the `asadmin set` command, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Downloading and Launching an Application Client

If Java Web Start is enabled for your deployed application client, you can launch it for testing. Simply click on the Launch button next to the application client or application's listing on the App Client Modules page in the Administration Console.

On other machines, you can download and launch the application client using Java Web Start in the following ways:

- Using a web browser, directly enter the URL for the application client. See "The Application Client URL" on page 221.
- Click on a link to the application client from a web page.
- Use the Java Web Start command `javaws`, specifying the URL of the application client as a command line argument.
- If the application has previously been downloaded using Java Web Start, you have additional alternatives.
  - Use the desktop icon that Java Web Start created for the application client. When Java Web Start downloads an application client for the first time it asks you if such an icon should be created.
  - Use the Java Web Start control panel to launch the application client.

When you launch an application client, Java Web Start contacts the server to see if a newer client version is available. This means you can redeploy an application client without having to worry about whether client machines have the latest version.

The Application Client URL

The default URL for an application or module generally is as follows:

`http://`*host*`:`*port*`/`*context-root*

The default URL for a stand-alone application client module is as follows:

`http://`*host*`:`*port*`/`*appclient-module-id*

The default URL for an application client module embedded within an application is as follows. Note that the relative path to the application client JAR file is included.

`http://`*host*`:`*port*`/`*application-id*`/`*appclient-path*

If the *context-root*, *appclient-module-id*, or *application-id* is not specified during deployment, the name of the JAR or EAR file without the extension is used. If the application client module or application is not in JAR or EAR file format, an *appclient-module-id* or *application-id* is generated.

Regardless of how the *context-root* or *id* is determined, it is written to the server log. For details about naming, see "Naming Standards" in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

To set a different URL for an application client, use the `context-root` subelement of the `java-web-start-access` element in the `sun-application-client.xml` file. This overrides the *appclient-module-id* or *application-id*. See *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

You can also pass arguments to the ACC or to the application client's `main` method as query parameters in the URL. If multiple application client arguments are specified, they are passed in the order specified.

A question mark separates the context root from the arguments. Ampersands (`&`) separate the arguments and their values. Each argument and each value must begin with `arg=`. Here is an example URL with a `-color` argument for a stand-alone application client. The `-color` argument is passed to the application client's `main` method.

```
http://localhost:8080/testClient?arg=-color&arg=red
```

Note – If you are using the `javaws` *URL* command to launch Java Web Start with a URL that contains arguments, enclose the URL in double quotes (") to avoid breaking the URL at the ampersand (&) symbol.

Ideally, you should build your production application clients with user-friendly interfaces that collect information which might otherwise be gathered as command-line arguments. This minimizes the degree to which users must customize the URLs that launch application clients using Java Web Start. Command-line argument support is useful in a development environment and for existing application clients that depend on it.

Signing JAR Files Used in Java Web Start

Java Web Start enforces a security sandbox. By default it grants any application, including application clients, only minimal privileges. Because Java Web Start applications can be so easily downloaded, Java Web Start provides protection from potentially harmful programs that might be accessible over the network. If an application requires a higher privilege level than the sandbox permits, the code that needs privileges must be in a JAR file that was signed. When Java Web Start downloads such a signed JAR file, it displays information about the certificate that was used to sign the JAR, and it asks you whether you want to trust that signed code. If you agree, the code receives elevated permissions and runs. If you reject the signed code, Java Web Start does not start the downloaded application.

The Enterprise Server serves two types of signed JAR files in response to Java Web Start requests. One type is a JAR file installed as part of the Enterprise Server, which starts an application client during a Java Web Start launch: *as-install*`/modules/gf-client.jar`.

The other type is a generated application client JAR file. As part of deployment, the Enterprise Server generates a new application client JAR file that contains classes, resources, and descriptors needed to run the application client on end-user systems. When you deploy an application with the `asadmin deploy` command's `--retrieve` option, use the `asadmin get-client-stubs` command, or select the Generate RMIStubs option from the EJB Modules deployment page in the Administration Console, this is the JAR file retrieved to your system. Because application clients need access beyond the minimal sandbox permissions to work in the Java Web Start environment, the generated application client JAR file must be signed before it can be downloaded to and executed on an end-user system.

A JAR file can be signed automatically or manually. The following sections describe the ways of signing JAR files.

- "Automatically Signing JAR Files" on page 223
- "Using the `jar-signing-alias` Deployment Property" on page 223
- "Manually Signing the Generated Application Client JAR File" on page 224

Automatically Signing JAR Files

The Enterprise Server automatically creates a signed version of the required JAR file if none exists. When a Java Web Start request for the `gf-client.jar` file arrives, the Enterprise Server looks for *domain-dir*`/java-web-start/gf-client.jar`. When a request for an application's generated application client JAR file arrives, the Enterprise Server looks in the directory *domain-dir*`/java-web-start/`*app-name* for a file with the same name as the generated JAR file created during deployment.

In either case, if the requested signed JAR file is absent or older than its unsigned counterpart, the Enterprise Server creates a signed version of the JAR file automatically and deposits it in the relevant directory. Whether the Enterprise Server just signed the JAR file or not, it serves the file from the *domain-dir*`/java-web-start` directory tree in response to the Java Web Start request.

To sign these JAR files, the Enterprise Server uses its self-signed certificate. When you create a new domain, either by installing the Enterprise Server or by using the `asadmin create-domain` command, the Enterprise Server creates a self-signed certificate and adds it to the domain's key store.

A self-signed certificate is generally untrustworthy because no certification authority vouches for its authenticity. The automatic signing feature uses the same certificate to create all required signed JAR files. To sign different JAR files with different certificates, do the signing manually.

Using the `jar-signing-alias` Deployment Property

The `asadmin deploy` command property `jar-signing-alias` specifies the alias for the security certificate with which the application client container JAR file is signed.

Java Web Start won't execute code requiring elevated permissions unless it resides in a JAR file signed with a certificate that the user's system trusts. For your convenience, Enterprise Server signs the JAR file automatically using the self-signed certificate from the domain, `s1as`. Java Web Start then asks the user whether to trust the code and displays the Enterprise Server certificate information.

To sign this JAR file with a different certificate, first add the certificate to the domain keystore. You can use a certificate from a trusted authority, which avoids the Java Web Start prompt, or from your own company, which users know they can trust. To add a certificate to the domain keystore, see "Administering JSSE Certificates " in *Sun GlassFish Enterprise Server v3 Administration Guide*.

Next, deploy your application using the `jar-signing-alias` property. For example:

```
asadmin deploy --property jar-signing-alias=MyAlias MyApp.ear
```

For more information about the `asadmin deploy` command, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Manually Signing the Generated Application Client JAR File

You can sign the generated application client JAR file for an application any time after you have deployed the application. As you deploy the application, you can specify the `asadmin deploy` command's `--retrieve` option or select the Generate RMIStubs option on the EJB Modules deployment page in the Administration Console. Doing either of these tasks returns a copy of the generated application client JAR file to a directory you specify. Or, after you have deployed an application, you can download the generated application client JAR file using the `asadmin get-client-stubs` command.

Once you have a copy of the generated application client JAR file, you can sign it using the `jarsigner` tool and your certificate. Then place the signed JAR file in the *domain-dir*`/java-web-start/`*app-name* directory. You do not need to restart the server to start using the new signed JAR file.

Error Handling

When an application client is launched using Java Web Start, any error that the application client logic does not catch and handle is written to `System.err` and displayed in a dialog box. This display appears if an error occurs even before the application client logic receives control. It also appears if the application client code does not catch and handle errors itself.

Vendor Icon, Splash Screen, and Text

To specify a vendor-specific icon, splash screen, text string, or a combination of these for Java Web Start download and launch screens, use the `vendor` element in the `sun-application-client.xml` file. The complete format of this element's data is as follows:

`<vendor>`*icon-image-URI*`::`*splash-screen-image-URI*`::`*vendor-text*`</vendor>`

The following example vendor element contains an icon, a splash screen, and a text string:

```
<vendor>images/icon.jpg::otherDir/splash.jpg::MyCorp, Inc.</vendor>
```

The following example vendor element contains an icon and a text string:

```
<vendor>images/icon.jpg::MyCorp, Inc.</vendor>
```

The following example vendor element contains a splash screen and a text string; note the initial double colon:

```
<vendor>::otherDir/splash.jpg::MyCorp, Inc.</vendor>
```

The following example vendor element contains only a text string:

```
<vendor>MyCorp, Inc.</vendor>
```

The default value is the text string `Application Client`.

For more information about the `sun-application-client.xml` file, see the *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

Using the Embeddable ACC

You can embed the ACC into your application client. If you place the *as-install*`/modules/gf-client.jar` file in your runtime classpath, your application creates the ACC after your application code has started, then requests that the ACC start the application client portion. The basic model for coding is as follows:

1. Create a builder object.
2. Operate on the builder to configure the ACC.
3. Obtain a new ACC instance from the builder.
4. Present a client archive or class to the ACC instance.
5. Start the client running within the newly created ACC instance.

Your code should follow this general pattern:

```
// one TargetServer for each ORB endpoint for bootstrapping
TargetServer[] servers = ...;

// Get a builder to set up the ACC
AppClientContainer.Builder builder = AppClientContainer.newBuilder(servers);

// Fine-tune the ACC's configuration.  Note ability to "chain" invocations.
builder.callbackHandler("com.acme.MyHandler").authRealm("myRealm"); // Modify config

// Get a container for a client.
```

```
URI clientURI = ...; // URI to the client JAR
AppClientContainer acc = builder.newContainer(clientURI);

or

Class mainClass = ...;
AppClientContainer acc = builder.newContainer(mainClass);

// In either case, start the client running.
String[] appArgs = ...;
acc.startClient(appArgs); // Start the client

...

acc.close(); // close the ACC(optional)
```

The ACC loads the application client's `main` class, performs any required injection, and transfers control to the `static` `main` method. The ACC's `run` method returns to the calling application as soon as the client's `main` method returns to the ACC.

If the application client's `main` method starts any asynchronous activity, that work continues after the ACC returns. The ACC has no knowledge of whether the client's `main` method triggers asynchronous work. Therefore, if the client causes work on threads other than the calling thread, and if the embedding application needs to know when the client's asynchronous work completes, the embedding application and the client must agree on how this happens.

The ACC's shutdown handling is invoked from the ACC's `close` method. The calling application can invoke `acc.close()` to close down any services started by the ACC. If the application client code started any asynchronous activity that might still depend on ACC services, invoking `close` before that asynchronous activity completes could cause unpredictable and undesirable results. The shutdown handling is also run automatically at VM shutdown if the code has not invoked `close` before then.

The ACC does not prevent the calling application from creating or running more than one ACC instance during a single execution of the application either serially or concurrently. However, other services used by the ACC (transaction manager, security, ORB, and so on) might or might not support such serial or concurrent reuse.

Running an Application Client Using the `appclient` Script

To run an application client, you can launch the ACC using the `appclient` script, whether or not Java Web Start is enabled. This is optional. This script is located in the *as-install*`/bin` directory. Enterprise Server v3 introduces new features and syntax for the `appclient` script, including the `-targetserver` option and the ability to specify JVM options more conveniently. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Using the `package-appclient` Script

You can package the Enterprise Server system files required to launch application clients on remote systems into a single JAR file using the `package-appclient` script. This is optional. This script is located in the *as-install*`/bin` directory. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

The `client.policy` File

The `client.policy` file is the J2SE policy file used by the application client. Each application client has a `client.policy` file. The default policy file limits the permissions of Java EE deployed application clients to the minimal set of permissions required for these applications to operate correctly. If an application client requires more than this default set of permissions, edit the `client.policy` file to add the custom permissions that your application client needs. Use the J2SE standard policy tool or any text editor to edit this file.

For more information on using the J2SE policy tool, see `http://java.sun.com/docs/books/tutorial/security1.2/tour2/index.html`.

For more information about the permissions you can set in the `client.policy` file, see `http://java.sun.com/javase/6/docs/technotes/guides/security/permissions.html`.

Using RMI/IIOP Over SSL

You can configure RMI/IIOP over SSL in two ways: using a username and password, or using a client certificate.

To use a username and password, configure the `ior-security-config` element in the `sun-ejb-jar.xml` file. The following configuration establishes SSL between an application client and an EJB component using a username and password. The user has to login to the ACC using either the `sun-acc.xml` mechanism or the "Programmatic Login" on page 102 mechanism.

```
<ior-security-config>
  <transport-config>
    <integrity>required</integrity>
    <confidentiality>required</confidentiality>
    <establish-trust-in-target>supported</establish-trust-in-target>
    <establish-trust-in-client>none</establish-trust-in-client>
  </transport-config>
  <as-context>
    <auth-method>username_password</auth-method>
    <realm>default</realm>
    <required>true</required>
  </as-context>
 <sas-context>
```

```
    <caller-propagation>none</caller-propagation>
 </sas-context>
</ior-security-config>
```

For more information about the `sun-ejb-jar.xml` and `sun-acc.xml` files, see the *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

To use a client certificate, configure the `ior-security-config` element in the `sun-ejb-jar.xml` file. The following configuration establishes SSL between an application client and an EJB component using a client certificate.

```
<ior-security-config>
  <transport-config>
    <integrity>required</integrity>
    <confidentiality>required</confidentiality>
    <establish-trust-in-target>supported</establish-trust-in-target>
    <establish-trust-in-client>required</establish-trust-in-client>
  </transport-config>
  <as-context>
    <auth-method>none</auth-method>
    <realm>default</realm>
    <required>false</required>
  </as-context>
  <sas-context>
    <caller-propagation>none</caller-propagation>
  </sas-context>
</ior-security-config>
```

To use a client certificate, you must also specify the system properties for the keystore and truststore to be used in establishing SSL. To use SSL with the Application Client Container (ACC), you need to set these system properties in one of the following ways:

- Use the new syntax of the `appclient` script and specify the system properties as JVM options. See "Running an Application Client Using the `appclient` Script" on page 226.
- Set the environment variable `VMARGS` in the shell. For example, in the `ksh` or `bash` shell, the command to set this environment variable would be as follows:

  ```
  export VMARGS="-Djavax.net.ssl.keyStore=${keystore.db.file}
  -Djavax.net.ssl.trustStore=${truststore.db.file}
  -Djavax.net.ssl.keyStorePass word=${ssl.password}
  -Djavax.net.ssl.trustStorePassword=${ssl.password}"
  ```
- Set the `env` element using Ant (see Chapter 3, "Using Ant with Enterprise Server"). For example:

  ```
  <target name="runclient">
    <exec executable="${S1AS_HOME}/bin/appclient">
      <env key="VMARGS" value=" -Djavax.net.ssl.keyStore=${keystore.db.file}
        -Djavax.net.ssl.trustStore=${truststore.db.file}
        -Djavax.net.ssl.keyStorePasword=${ssl.password}
        -Djavax.net.ssl.trustStorePassword=${ssl.password}"/>
      <arg value="-client"/>
      <arg value="${appClient.jar}"/>
    </exec>
  </target>
  ```

Connecting to a Remote EJB Module Through a Firewall

To deploy and run an application client that connects to an EJB module on a Enterprise Server instance that is behind a firewall, you must set ORB Virtual Address Agent Implementation (ORBVAA) options. Use the `asadmin create-jvm-options` command as follows:

```
asadmin create-jvm-options -Dcom.sun.corba.ee.ORBVAAHost=public-IP-adress
asadmin create-jvm-options -Dcom.sun.corba.ee.ORBVAAPort=public-port
asadmin create-jvm-options
-Dcom.sun.corba.ee.ORBUserConfigurators.com.sun.corba.ee.impl.plugin.hwlb.VirtualAddressAgentImpl=x
```

Set the `ORBVAAHost` and `ORBVAAPort` options to the host and port of the public address. The `ORBUserConfigurators` option tells the ORB to create an instance of the `VirtualAddressAgentImpl` class and invoke the `configure` method on the resulting object, which must implement the `com.sun.corba.ee.spi.orb.ORBConfigurator` interface. The `ORBUserConfigurators` value doesn't matter. Together, these options create an ORB that in turn creates `Object` references (the underlying implementation of remote EJB references) containing the public address, while the ORB listens on the private address specified for the IIOP port in the Enterprise Server configuration.

Using JavaFX Code

To use JavaFX code in an application client, compile the JavaFX script into a Java class and place the Java class in the appclient JAR file. To access back-end resources such as EJB components from a JavaFX script, you can write static public methods in your application client main class that refer to injected resources. The JavaFX script code can then refer to those static methods.

Specifying a Splash Screen

Java SE 6 offers splash screen support, either through a Java command-line option or a manifest entry in the application's JAR file. To take advantage of this Java SE feature in your application client, you can do one of the following:

- Create the appclient JAR file so that its manifest contains a `SplashScreen-Image` entry that specifies the path to the image in the client. The `java` command displays the splash screen before starting the ACC or your client, just as with any Java application.
- Use the new `appclient ... -jar` launch format, using the `-splash` command-line option at runtime or the `SplashScreen-Image` manifest entry at development time. See "Running an Application Client Using the `appclient` Script" on page 226.
- In the environment that runs the `appclient` script, set the `VMOPTS` environment variable to include the `-splash` option before invoking the `appclient` script to launch the client.

- Build an application client that uses the embeddable ACC feature and specify the splash screen image using one of the following:
  - The `-splash` option of the `java` command
  - `SplashScreen-Image` in the manifest for your program (not the manifest for the application client)

  See "Using the Embeddable ACC" on page 225.

During application (EAR file) deployment, the Enterprise Server generates façade JAR files, one for the application and one for each application client in the application. During application client module deployment, the Enterprise Server generates a single facade JAR for the application client. The `appclient` script supports splash screens inside the application client JAR only if you launch an application client facade or appclient client JAR. If you launch the facade for an application or the undeployed application itself, the `appclient` script cannot take advantage of the Java SE 6 splash screen feature.

Setting Login Retries

You can set a JVM option using the `appclient` script that determines the number of login retries allowed. This option is `-Dorg.glassfish.appclient.acc.maxLoginRetries=`*n* where *n* is a positive integer. The default number of retries is 3.

This retry loop happens when the ACC attempts to perform injection if you annotated the client's `main` class (for example, using `@Resource`). If instead of annotations your client uses the `InitialContext` explicitly to look up remote resources, the retry loop does not apply. In this case, you could write logic to catch an exception around the lookup and retry explicitly.

For details about the `appclient` script syntax, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Using Libraries with Application Clients

The Libraries field in the Administration Console's deployment page and the `--libraries` option of the `asadmin deploy` command do not apply to application clients. Neither do the *as-install*`/lib`, *domain-dir*`/lib`, and *domain-dir*`/lib/classes` directories comprising the Common Class Loader. These apply only to applications and modules deployed to the server. For more information, see Chapter 2, "Class Loaders."

To use libraries with an application client, package the application client in an application (EAR file). Then, either place the libraries in the `/lib` directory of the EAR file or specify their location in the application client JAR file's manifest `Class-Path`.

◆ ◆ ◆ CHAPTER 12

Developing Connectors

This chapter describes Sun GlassFish Enterprise Server support for the Java EE 1.6 Connector Architecture, also known as JSR 322 (`http://jcp.org/en/jsr/detail?id=322`).

The Java EE Connector Architecture provides a Java solution to the problem of connectivity between multiple application servers and existing enterprise information systems (EISs). By using the Java EE Connector architecture, EIS vendors no longer need to customize their product for each application server. Application server vendors who conform to the Java EE Connector architecture do not need to write custom code to add connectivity to a new EIS.

This chapter uses the terms *connector* and *resource adapter* interchangeably. Both terms refer to a resource adapter module that is developed in conformance with the Java EE Connector Architecture Specification.

Note – If you installed the Web Profile, connector modules that use only outbound communication features and work-management that does not involve inbound communication features are supported. Other connector features are supported only in the full Enterprise Server.

For more information about connectors, see Java EE Connector Architecture (`http://java.sun.com/j2ee/connector/`).

For connector examples, see `http://developers.sun.com/prodtech/appserver/reference/techart/as8_connectors`.

For information about deploying a connector to the Enterprise Server, see the *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

This chapter includes the following topics:

Connector Support in the Enterprise Server

The Enterprise Server supports the development and deployment of resource adapters that are compatible with the Connector 1.6 specification (and, for backward compatibility, the Connector 1.0 and 1.5 specifications).

The Connector 1.0 specification defines the outbound connectivity system contracts between the resource adapter and the Enterprise Server. The Connector 1.5 specification introduces major additions in defining system level contracts between the Enterprise Server and the resource adapter with respect to inbound connectivity, life cycle management, and thread management. The Connector 1.6 specification introduces further additions in defining system level contracts between the Enterprise Server and the resource adapter with respect to the following:

- Generic work context contract — A generic contract that enables a resource adapter to control the execution context of a `Work` instance that it has submitted to the Enterprise Server for execution. The `Generic` work contract provides the mechanism for a resource adapter to augment the runtime context of a `Work` instance with additional contextual information flown-in from the EIS. This contract enables a resource adapter to control, in a more flexible manner, the contexts in which the `Work` instances submitted by it are executed by the application server's `WorkManager`.
- Security work context — A standard contract that enables a resource adapter to establish security information while submitting a `Work` instance for execution to a `WorkManager` and while delivering messages-to-message endpoints residing in the Enterprise Server. This contract provides a mechanism to support the execution of a `Work` instance in the context of an established identity. It also supports the propagation of user information or Principal information from an EIS to a `MessageEndpoint` during message inflow.
- Transaction context — The transaction context contract between the resource adapter and the application server leverages the Generic Work Context mechanism by describing a standard `WorkContext`, the `TransactionContext`. It represents the standard interface a resource adapter can use to propagate transaction context information from the EIS to the application server.

Connector Architecture for JMS and JDBC

In the Administration Console, connector, JMS, and JDBC resources are handled differently, but they use the same underlying Connector architecture. In the Enterprise Server, all communication to an EIS, whether to a message provider or an RDBMS, happens through the

Connector architecture. To provide JMS infrastructure to clients, the Enterprise Server uses the Sun GlassFish Message Queue software. To provide JDBC infrastructure to clients, the Enterprise Server uses its own JDBC system resource adapters. The application server automatically makes these system resource adapters available to any client that requires them.

For more information about JMS in the Enterprise Server, see Chapter 17, "Using the Java Message Service." For more information about JDBC in the Enterprise Server, see Chapter 14, "Using the JDBC API for Database Access."

Connector Configuration

The Enterprise Server does not need to use `sun-ra.xml`, which previous Enterprise Server versions used, to store server-specific deployment information inside a Resource Adapter Archive (RAR) file. (However, the `sun-ra.xml` file is still supported for backward compatibility.) Instead, the information is stored in the server configuration. As a result, you can create multiple connector connection pools for a connection definition in a functional resource adapter instance, and you can create multiple user-accessible connector resources (that is, registering a resource with a JNDI name) for a connector connection pool. In addition, dynamic changes can be made to connector connection pools and the connector resource properties without restarting the Enterprise Server.

Advanced Connector Configuration Options

You can use these advanced connector configuration options:

Thread Associations

Connectors can submit work instances to the Enterprise Server for execution. By default, the Enterprise Server services work requests for all connectors from its default thread pool.

However, you can associate a specific user-created thread pool to service work requests from a connector. A thread pool can service work requests from multiple resource adapters. To create a thread pool:

- In the Administration Console, select Thread Pools under the relevant configuration. For details, click the Help button in the Administration Console.
- Use the `asadmin create-threadpool` command. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

To associate a connector with a thread pool:

- In the Administration Console, open the Applications component and select Resource Adapter Configs. Specify the name of the thread pool in the Thread Pool ID field. For details, click the Help button in the Administration Console.
- Use the `--threadpoolid` option of the `asadmin create-resource-adapter-config` command. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

If you create a resource adapter configuration for a connector module that is already deployed, the connector module deployment is restarted with the new configuration properties.

Security Maps

Create a security map for a connector connection pool to map an application principal or a user group to a back end EIS principal. The security map is usually used in situations where one or more EIS back end principals are used to execute operations (on the EIS) initiated by various principals or user groups in the application.

To create or update security maps for a connector connection pool:

- In the Administration Console, open the Resources component, select Connectors, select Connector Connection Pools, and select the Security Maps tab. For details, click the Help button in the Administration Console.
- Use the `asadmin create-connector-security-map` command. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

If a security map already exists for a connector connection pool, the new security map is appended to the previous one. The connector security map configuration supports the use of the wildcard asterisk (`*`) to indicate all users or all user groups.

When an application principal initiates a request to an EIS, the Enterprise Server first checks for an exact match to a mapped back end EIS principal using the security map defined for the connector connection pool. If there is no exact match, the Enterprise Server uses the wild card character specification, if any, to determined the mapped back end EIS principal.

Work Security Maps

A work security map for a resource adapter maps an EIS principal or group to a application principal or group. A work security map is useful in situations where one or more application principals execute operations initiated by principals or user groups in the EIS. A resource adapter can have multiple work security maps. A work security map can map either principals or groups, but not both.

To create a work security map, use the `asadmin create-connector-work-security-map` command. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

The work security map configuration supports the wildcard asterisk (`*`) character to indicate all users or all user groups. When an EIS principal sends a request to the Enterprise Server, the Enterprise Server first checks for an exact match to a mapped application principal using the work security map defined for the resource adapter. If there is no exact match, the Enterprise Server uses the wild card character specification, if any, to determine the application principal.

Overriding Configuration Properties

You can override the properties (`config-property` elements) specified in the `ra.xml` file of a resource adapter:

- In the Administration Console, open the Resources component and select Resource Adapter Configs. Create a new resource adapter configuration or select an existing one to edit. Then enter property names and values in the Additional Properties table. For details, click the Help button in the Administration Console.
- Use the `asadmin create-resource-adapter-config` command to create a configuration for a resource adapter. Use this command's `--property` option to specify a name-value pair for a resource adapter property. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

You can specify configuration properties either before or after resource adapter deployment. If you specify properties after deploying the resource adapter, the existing resource adapter is restarted with the new properties.

Testing a Connector Connection Pool

You can test a connector connection pool for usability in one of these ways:

- In the Administration Console, open the Resources component, open the Connector component, select Connection Pools, and select the connection pool you want to test. Then select the Ping button in the top right corner of the page. For details, click the Help button in the Administration Console.
- Use the `asadmin ping-connection-pool` command. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Both these commands fail and display an error message unless they successfully connect to the connection pool.

You can also specify that a connection pool is automatically tested when created or reconfigured by setting the Ping attribute to `true` (the default is `false`) in one of the following ways:

- Enter a Ping value in the Connector Connection Pools page in the Administration Console. For more information, click the Help button in the Administration Console.
- Specify the `--ping` option in the `asadmin create-connector-connection-pool` command. For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Flushing a Connector Connection Pool

Flushing a connector connection pool recreates all the connections in the pool and brings the pool to the steady pool size without the need for reconfiguring the pool. Connection pool reconfiguration can result in application redeployment, which is a time-consuming operation. Flushing destroys existing connections, and any existing transactions are lost and must be retired.

You can flush a connector connection pool in one of these ways:

- In the Administration Console, open the Resources component, open the Connector component, select Connection Pools, and select the connection pool you want to flush. Then select the Flush button in the top right corner of the page. For details, click the Help button in the Administration Console.
- Use the `asadmin flush-connection-pool` command. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Handling Invalid Connections

If a resource adapter generates a `ConnectionErrorOccured` event, the Enterprise Server considers the connection invalid and removes the connection from the connection pool. Typically, a resource adapter generates a `ConnectionErrorOccured` event when it finds a `ManagedConnection` object unusable. Reasons can be network failure with the EIS, EIS failure, fatal problems with the resource adapter, and so on.

If the `fail-all-connections` setting in the connection pool configuration is set to `true`, and a single connection fails, all connections are closed and recreated. If this setting is `false`, individual connections are recreated only when they are used. The default is `false`.

The `is-connection-validation-required` setting specifies whether connections have to be validated before being given to the application. If a resource's validation fails, it is destroyed, and a new resource is created and returned. The default is `false`.

The `prefer-validate-over-recreate` property specifies that validating idle connections is preferable to closing them. This property has no effect on non-idle connections. If set to `true`, idle connections are validated during pool resizing, and only those found to be invalid are destroyed and recreated. If `false`, all idle connections are destroyed and recreated during pool resizing. The default is `false`.

You can set the `fail-all-connections`, `is-connection-validation-required`, and `prefer-validate-over-recreate` configuration settings during creation of a connector connection pool. Or, you can use the `asadmin set` command to dynamically reconfigure a setting. For example:

```
asadmin set server.resources.connector-connection-pool.CCP1.fail-all-connections="true"
asadmin set server.resources.connector-connection-pool.CCP1.is-connection-validation-required="true"
asadmin set server.resources.connector-connection-pool.CCP1.property.prefer-validate-over-recreate="true"
```

For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

The interface `ValidatingManagedConnectionFactory` exposes the method `getInvalidConnections` to allow retrieval of the invalid connections. The Enterprise Server checks if the resource adapter implements this interface, and if it does, invalid connections are removed when the connection pool is resized.

Setting the Shutdown Timeout

According to the Connector specification, while an application server shuts down, all resource adapters should be stopped. A resource adapter might hang during shutdown, since shutdown is typically a resource intensive operation. To avoid such a situation, you can set a timeout that aborts resource adapter shutdown if exceeded. The default timeout is 30 seconds per resource adapter module. To configure this timeout:

- In the Administration Console, select Connector Service under the relevant configuration and edit the shutdown Timeout field. For details, click the Help button in the Administration Console.
- Use the following `asadmin set` command:

  ```
  asadmin set server.connector-service.shutdown-timeout-in-seconds="num-secs"
  ```

  For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

The Enterprise Server deactivates all message-driven bean deployments before stopping a resource adapter.

Specifying the Class Loading Policy

Use the `class-loading-policy` setting to determine which resource adapters accessible to applications. Allowed values are:

- `derived` — Applications access resource adapters based on references in their deployment descriptors. These references can be `resource-ref`, `resource-env-ref`, `resource-adapter-mid`, or equivalent annotations.
- `global` — All stand-alone resource adapters are available to all applications.

To configure this setting, use the `asadmin set` command. For example:

```
asadmin set server.connector-service.class-loading-policy="global"
```

For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Using Last Agent Optimization of Transactions

Transactions that involve multiple resources or multiple participant processes are *distributed* or *global* transactions. A global transaction can involve one non-XA resource if last agent optimization is enabled. Otherwise, all resources must be XA. For more information about transactions in the Enterprise Server, see Chapter 15, "Using the Transaction Service."

The Connector specification requires that if a resource adapter supports `XATransaction`, the `ManagedConnection` created from that resource adapter must support both distributed and local transactions. Therefore, even if a resource adapter supports `XATransaction`, you can configure its connector connection pools as non-XA or without transaction support for better performance. A non-XA resource adapter becomes the last agent in the transactions in which it participates.

The value of the connection pool configuration property `transaction-support` defaults to the value of the `transaction-support` property in the `ra.xml` file. The connection pool configuration property can override the `ra.xml` file property if the transaction level in the connection pool configuration property is lower. If the value in the connection pool configuration property is higher, it is ignored.

Disabling Pooling for a Connection

To disable connection pooling, set the Pooling attribute to false. The default is true. You can enable or disable connection pooling in one of the following ways:

- Enter a Pooling value in the Connector Connection Pools page in the Administration Console. For more information, click the Help button in the Administration Console.
- Specify the `--pooling` option in the `asadmin create-connector-connection-pool` command. For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Inbound Communication Support

The Connector specification defines the transaction and message inflow system contracts for achieving inbound connectivity from an EIS. The message inflow contract also serves as a standard message provider pluggability contract, thereby allowing various message providers to seamlessly plug in their products with any application server that supports the message inflow contract. In the inbound communication model, the EIS initiates all communication to an application. An application can be composed of enterprise beans (session, entity, or message-driven beans), which reside in an EJB container.

Incoming messages are received through a message endpoint, which is a message-driven bean. This message-driven bean asynchronously consumes messages from a message provider. An application can also synchronously send and receive messages directly using messaging style APIs.

A resource adapter supporting inbound communication provides an instance of an `ActivationSpec` JavaBean class for each supported message listener type. Each class contains a set of configurable properties that specify endpoint activation configuration information during message-driven bean deployment. The required `config-property` element in the `ra.xml` file provides a list of configuration property names required for each activation specification. An endpoint activation fails if the required property values are not specified. Values for the properties that are overridden in the message-driven bean's deployment descriptor are applied to the `ActivationSpec` JavaBean when the message-driven bean is deployed.

Administered objects can also be specified for a resource adapter, and these JavaBeans are specific to a messaging style or message provider. For example, some messaging styles may need applications to use special administered objects (such as `Queue` and `Topic` objects in JMS). Applications use these objects to send and synchronously receive messages using connection objects using messaging style APIs. For more information about administered objects, see Chapter 17, "Using the Java Message Service."

Outbound Communication Support

The Connector specification defines the system contracts for achieving outbound connectivity from an EIS. A resource adapter supporting outbound communication provides an instance of a `ManagedConnectionFactory` JavaBean class. A `ManagedConnectionFactory` JavaBean represents outbound connectivity information to an EIS instance from an application.

The 1.6 Connector specification introduces a mechanism through which the transaction level of a `ManagedConnectionFactory` can be detected at runtime. During the configuration of a `ManagedConnectionFactory` in the Connector Connection Pools page in the Administration Console, the Administration Console can instantiate the `ManagedConnectionFactory` and show the level of transaction support. The three levels are `no-tx`, `local-tx`, `xa-tx`. If a

`ManagedConnectionFactory` returns `local-tx` as the level it can support, it is assumed that `xa-tx` is not supported, and the Administration Console shows only `no-tx` and `local-tx` as the available support levels.

For more information, click the Help button in the Administration Console.

Configuring a Message Driven Bean to Use a Resource Adapter

The Connectors specification's message inflow contract provides a generic mechanism to plug in a wide-range of message providers, including JMS, into a Java-EE-compatible application server. Message providers use a resource adapter and dispatch messages to message endpoints, which are implemented as message-driven beans.

The message-driven bean developer provides activation configuration information in the message-driven bean's `ejb-jar.xml` file. Configuration information includes messaging-style-specific configuration details, and possibly message-provider-specific details as well. The message-driven bean deployer uses this configuration information to set up the activation specification JavaBean. The activation configuration properties specified in `ejb-jar.xml` override configuration properties in the activation specification definition in the `ra.xml` file.

According to the EJB specification, the messaging-style-specific descriptor elements contained within the activation configuration element are not specified because they are specific to a messaging provider. In the following sample message-driven bean `ejb-jar.xml`, a message-driven bean has the following activation configuration property names: `destinationType`, `SubscriptionDurability`, and `MessageSelector`.

```
<!--  A sample MDB that listens to a JMS Topic -->
<!-- message-driven bean deployment descriptor -->
...
 <activation-config>
   <activation-config-property>
     <activation-config-property-name>
       destinationType
     </activation-config-property-name>
     <activation-config-property-value>
       javax.jms.Topic
     </activation-config-property-value>
  </activation-config-property>
  <activation-config-property>
     <activation-config-property-name>
       SubscriptionDurability
     </activation-config-property-name>
     <activation-config-property-value>
       Durable
     </activation-config-property-value>
  </activation-config-property>
  <activation-config-property>
     <activation-config-property-name>
```

```
      MessageSelector
    </activation-config-property-name>
    <activation-config-property-value>
      JMSType = 'car' AND color = 'blue'
    </activation-config-property-value>
  </activation-config-property>
 ...
 </activation-config>
...
```

When the message-driven bean is deployed, the value for the `resource-adapter-mid` element in the `sun-ejb-jar.xml` file is set to the resource adapter module name that delivers messages to the message endpoint (to the message-driven bean). In the following example, the `jmsra` JMS resource adapter, which is the bundled resource adapter for the Sun GlassFish Message Queue message provider, is specified as the resource adapter module identifier for the `SampleMDB` bean.

```
<sun-ejb-jar>
<enterprise-beans>
    <unique-id>1</unique-id>
    <ejb>
       <ejb-name>SampleMDB</ejb-name>
       <jndi-name>SampleQueue</jndi-name>
    <!-- JNDI name of the destination from which messages would be
         delivered from MDB needs to listen to -->
    ...
    <mdb-resource-adapter>
       <resource-adapter-mid>jmsra</resource-adapter-mid>
       <!-- Resource Adapter Module Id that would deliver messages to
            this message endpoint -->
       </mdb-resource-adapter>
    ...
 </ejb>
 ...
</enterprise-beans>
...
</sun-ejb-jar>
```

When the message-driven bean is deployed, the Enterprise Server uses the `resourceadapter-mid` setting to associate the resource adapter with a message endpoint through the message inflow contract. This message inflow contract with the application server gives the resource adapter a handle to the `MessageEndpointFactory` and the `ActivationSpec` JavaBean, and the adapter uses this handle to deliver messages to the message endpoint instances (which are created by the `MessageEndpointFactory`).

When a message-driven bean first created for use on the Enterprise Server 7 is deployed, the Connector runtime transparently transforms the previous deployment style to the current connector-based deployment style. If the deployer specifies neither a `resource-adapter-mid` property nor the Message Queue resource adapter's activation configuration properties, the Connector runtime maps the message-driven bean to the `jmsra` system resource adapter and converts the JMS-specific configuration to the Message Queue resource adapter's activation configuration properties.

◆ ◆ ◆ CHAPTER 13

Developing Lifecycle Listeners

Lifecycle listener modules provide a means of running short or long duration Java-based tasks within the Sun GlassFish Enterprise Server environment, such as instantiation of singletons or RMI servers. These modules are automatically initiated at server startup and are notified at various phases of the server life cycle.

Note – Lifecycle listener modules are deprecated. Support for them is included for backward compatibility. Implementing the `org.glassfish.api.Startup` interface instead is recommended.

All lifecycle module classes and interfaces are in the *as-install*`/modules/glassfish-api.jar` file.

For Javadoc tool pages relevant to lifecycle modules, go to `https://glassfish.dev.java.net/nonav/docs/v3/api/` and click on the `com.sun.appserv.server` package.

The following sections describe how to create and use a lifecycle listener module:

Server Life Cycle Events

A lifecycle module listens for and performs its tasks in response to the following events in the server life cycle:

- After the `INIT_EVENT`, the server reads the configuration, initializes built-in subsystems (such as security and logging services), and creates the containers.
- After the `STARTUP_EVENT`, the server loads and initializes deployed applications.
- After the `READY_EVENT`, the server is ready to service requests.
- After the `SHUTDOWN_EVENT`, the server destroys loaded applications and stops.
- After the `TERMINATION_EVENT`, the server closes the containers, the built-in subsystems, and the server runtime environment.

These events are defined in the `LifecycleEvent` class.

The lifecycle modules that listen for these events implement the `LifecycleListener` interface.

The `LifecycleListener` Interface

To create a lifecycle module is to configure a customized class that implements the `com.sun.appserv.server.LifecycleListener` interface. You can create and simultaneously execute multiple lifecycle modules.

The `LifecycleListener` interface defines this method:

```
public void handleEvent(com.sun.appserv.server.LifecycleEvent event)
throws ServerLifecycleException
```

This method responds to a lifecycle event and throws a `com.sun.appserv.server.ServerLifecycleException` if an error occurs.

A sample implementation of the `LifecycleListener` interface is the `LifecycleListenerImpl.java` file, which you can use for testing lifecycle events.

The `LifecycleEvent` Class

The `com.sun.appserv.server.LifecycleEvent` class defines a server life cycle event. The following methods are associated with the event:

- `public java.lang.Object.getData()`

  This method returns an instance of `java.util.Properties` that contains the properties defined for the lifecycle module.
- `public int getEventType()`

This method returns the type of the last event, which is `INIT_EVENT`, `STARTUP_EVENT`, `READY_EVENT`, `SHUTDOWN_EVENT`, or `TERMINATION_EVENT`.

- `public com.sun.appserv.server.LifecycleEventContext.getLifecycleEventContext()`

  This method returns the lifecycle event context, described next.

A `LifecycleEvent` instance is passed to the `LifecycleListener.handleEvent` method.

The Server Lifecycle Event Context

The `com.sun.appserv.server.LifecycleEventContext` interface exposes runtime information about the server. The lifecycle event context is created when the `LifecycleEvent` class is instantiated at server initialization. The `LifecycleEventContext` interface defines these methods:

- `public java.lang.String[].getCmdLineArgs()`

  This method returns the server startup command-line arguments.

- `public java.lang.String.getInstallRoot()`

  This method returns the server installation root directory.

- `public java.lang.String.getInstanceName()`

  This method returns the server instance name.

- `public javax.naming.InitialContext.getInitialContext()`

  This method returns the initial JNDI naming context. The naming environment for lifecycle modules is installed after the `STARTUP_EVENT`. A lifecycle module can look up any resource by its `jndi-name` attribute after the `READY_EVENT`.

If a lifecycle module needs to look up resources, it can do so after the `READY_EVENT`. It can use the `getInitialContext()` method to get the initial context to which all the resources are bound.

Deploying a Lifecycle Module

For instructions on how to deploy a lifecycle module, see the *Sun GlassFish Enterprise Server v3 Application Deployment Guide*, or see the `asadmin create-lifecycle-module` command in the *Sun GlassFish Enterprise Server v3 Reference Manual*.

You do not need to specify a classpath for the lifecycle module if you place it in the *domain-dir*`/lib` or *domain-dir*`/lib/classes` directory for the Domain Administration Server.

Considerations for Lifecycle Modules

The resources allocated at initialization or startup should be freed at shutdown or termination. The lifecycle module classes are called synchronously from the main server thread, therefore it is important to ensure that these classes don't block the server. Lifecycle modules can create threads if appropriate, but these threads must be stopped in the shutdown and termination phases.

The LifeCycleModule class loader is the parent class loader for lifecycle modules. Each lifecycle module's classpath is used to construct its class loader. All the support classes needed by a lifecycle module must be available to the LifeCycleModule class loader or its parent, the Connector class loader.

You must ensure that the `server.policy` file is appropriately set up, or a lifecycle module trying to perform a `System.exec()` might cause a security access violation. For details, see "The `server.policy` File" on page 88.

The configured properties for a lifecycle module are passed as properties after the `INIT_EVENT`. The JNDI naming context is not available before the `STARTUP_EVENT`. If a lifecycle module requires the naming context, it can get this after the `STARTUP_EVENT`, `READY_EVENT`, or `SHUTDOWN_EVENT`.

PART III

Using Services and APIs

◆ ◆ ◆ CHAPTER 14

Using the JDBC API for Database Access

This chapter describes how to use the Java Database Connectivity (JDBC) API for database access with the Sun GlassFish Enterprise Server. This chapter also provides high level JDBC implementation instructions for servlets and EJB components using the Enterprise Server. If the JDK version 1.6 is used, the Enterprise Server supports the JDBC 4.0 API.

The JDBC specifications are available at `http://java.sun.com/products/jdbc/download.html`.

A useful JDBC tutorial is located at `http://java.sun.com/docs/books/tutorial/jdbc/index.html`.

Note – The Enterprise Server does not support connection pooling or transactions for an application's database access if it does not use standard Java EE `DataSource` objects.

This chapter discusses the following topics:

General Steps for Creating a JDBC Resource

To prepare a JDBC resource for use in Java EE applications deployed to the Enterprise Server, perform the following tasks:

- "Creating a JDBC Resource" on page 253

For information about how to configure some specific JDBC drivers, see "Configuration Specifics for JDBC Drivers" in *Sun GlassFish Enterprise Server v3 Administration Guide*.

Integrating the JDBC Driver

To use JDBC features, you must choose a JDBC driver to work with the Enterprise Server, then you must set up the driver. This section covers these topics:

- "Supported Database Drivers" on page 250
- "Making the JDBC Driver JAR Files Accessible" on page 250
- "Automatic Detection of Installed Drivers" on page 250

Supported Database Drivers

Supported JDBC drivers are those that have been fully tested by Sun. For a list of the JDBC drivers currently supported by the Enterprise Server, see the *Sun GlassFish Enterprise Server v3 Release Notes*. For configurations of supported and other drivers, see "Configuration Specifics for JDBC Drivers" in *Sun GlassFish Enterprise Server v3 Administration Guide*.

Note – Because the drivers and databases supported by the Enterprise Server are constantly being updated, and because database vendors continue to upgrade their products, always check with Sun technical support for the latest database support information.

Making the JDBC Driver JAR Files Accessible

To integrate the JDBC driver into an Enterprise Server domain, copy the JAR files into the *domain-dir*`/lib` directory, then restart the server. This makes classes accessible to all applications or modules deployed on servers that share the same configuration. For more information about Enterprise Server class loaders, see Chapter 2, "Class Loaders."

If you are using an Oracle database with EclipseLink extensions, copy the JAR files into the *domain-dir*`/lib/ext` directory, then restart the server. For details, see "Oracle Database Enhancements" on page 127.

Automatic Detection of Installed Drivers

The Administration Console detects installed JDBC Drivers automatically when you create a JDBC connection pool. To create a JDBC connection pool using the Administration Console, open the Resources component, open the JDBC component, select Connection Pools, and click on the New button. This displays the New JDBC Connection Pool page.

Based on the Resource Type and Database Vendor you select on the New JDBC Connection Pool page, data source or driver implementation class names are listed in the Datasource Classname or Driver Classname field when you click on the Next button. When you choose a specific implementation class name on the next page, additional properties relevant to the installed JDBC driver are displayed in the Additional Properties section.

Creating a JDBC Connection Pool

When you create a connection pool that uses JDBC technology (a *JDBC connection pool*) in the Enterprise Server, you can define many of the characteristics of your database connections.

You can create a JDBC connection pool in one of these ways:

- In the Administration Console, open the Resources component, open the JDBC component, select Connection Pools, and click on the New button. This displays the New JDBC Connection Pool page. For details, click the Help button in the Administration Console.
- Use the `asadmin create-jdbc-connection-pool` command. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

For a complete description of JDBC connection pool features, see the *Sun GlassFish Enterprise Server v3 Administration Guide*.

Modifying a JDBC Connection Pool

In the Administration Console, some JDBC connection pool attributes are advanced, and you cannot set them during JDBC connection pool creation. You can only set them when modifying an existing JDBC connection pool. You can also use the `asadmin set` command to set or reset a JDBC connection pool's attributes.

You can modify a JDBC connection pool in one of these ways:

- In the Administration Console, open the Resources component, open the JDBC component, select Connection Pools, and click on the name of the connection pool you want to modify. This displays the Edit Connection Pool page. To edit advanced attributes, click on the Advanced tab. This displays the Edit Connection Pool Advanced Attributes page. For details, click the Help button in the Administration Console.
- Use the `asadmin set` command. For example:

  ```
  asadmin set domain1.resources.jdbc-connection-pool.DerbyPool.pooling=false
  ```

  For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Testing a JDBC Connection Pool

You can test a JDBC connection pool for usability in one of these ways:

- In the Administration Console, open the Resources component, open the JDBC component, select Connection Pools, and select the connection pool you want to test. Then select the Ping button in the top left corner of the page. For details, click the Help button in the Administration Console.
- Use the `asadmin ping-connection-pool` command. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Both these commands fail and display an error message unless they successfully connect to the connection pool.

You can also specify that a connection pool is automatically tested when created or reconfigured by setting the Ping attribute to `true` (the default is `false`) in one of the following ways:

- Enter a Ping value in the New JDBC Connection Pool or Edit Connection Pool page in the Administration Console. For more information, click the Help button in the Administration Console.
- Specify the `--ping` option in the `asadmin create-jdbc-connection-pool` command. For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.
- Specify the `ping` option in the `asadmin set` command. For example:

  ```
  asadmin set domain1.resources.jdbc-connection-pool.DerbyPool.ping=true
  ```

  For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Flushing a JDBC Connection Pool

Flushing a JDBC connection pool recreates all the connections in the pool and brings the pool to the steady pool size without the need for reconfiguring the pool. Connection pool reconfiguration can result in application redeployment, which is a time-consuming operation. Flushing destroys existing connections, and any existing transactions are lost and must be retired.

You can flush a JDBC connection pool in one of these ways:

- In the Administration Console, open the Resources component, open the JDBC component, select Connection Pools, and select the connection pool you want to flush. Then select the Flush button in the top left corner of the page. For details, click the Help button in the Administration Console.
- Use the `asadmin flush-connection-pool` command. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Creating a JDBC Resource

A JDBC resource, also called a data source, lets you make connections to a database using `getConnection()`. Create a JDBC resource in one of these ways:

- In the Administration Console, open the Resources component, open the JDBC component, and select JDBC Resources. For details, click the Help button in the Administration Console.
- Use the `asadmin create-jdbc-resource` command. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Creating Applications That Use the JDBC API

An application that uses the JDBC API is an application that looks up and connects to one or more databases. This section covers these topics:

Statements

The following features pertain to statements:

Using an Initialization Statement

You can specify a statement that executes each time a physical connection to the database is created (not reused) from a JDBC connection pool. This is useful for setting request or session specific properties and is suited for homogeneous requests in a single application. Set the Init SQL attribute of the JDBC connection pool to the SQL string to be executed in one of the following ways:

- Enter an Init SQL value in the Edit Connection Pool Advanced Attributes page in the Administration Console. For more information, click the Help button in the Administration Console.
- Specify the `--initsql` option in the `asadmin create-jdbc-connection-pool` command. For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

- Specify the `init-sql` option in the `asadmin set` command. For example:

  ```
  asadmin set domain1.resources.jdbc-connection-pool.DerbyPool.init-sql="sql-string"
  ```

  For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Setting a Statement Timeout

An abnormally long running JDBC query executed by an application may leave it in a hanging state unless a timeout is explicitly set on the statement. Setting a statement timeout guarantees that all queries automatically time out if not completed within the specified period. When statements are created, the `queryTimeout` is set according to the statement timeout setting. This works only when the underlying JDBC driver supports `queryTimeout` for `Statement`, `PreparedStatement`, `CallableStatement`, and `ResultSet`.

You can specify a statement timeout in the following ways:

- Enter a Statement Timeout value in the Edit Connection Pool Advanced Attributes page in the Administration Console. For more information, click the Help button in the Administration Console.
- Specify the `--statementtimeout` option in the `asadmin create-jdbc-connection-pool` command. For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Statement Caching

Statement caching stores statements, prepared statements, and callable statements that are executed repeatedly by applications in a cache, thereby improving performance. Instead of the statement being prepared each time, the cache is searched for a match. The overhead of parsing and creating new statements each time is eliminated.

Statement caching is usually a feature of the JDBC driver. The Enterprise Server provides caching for drivers that do not support caching. To enable this feature, set the Statement Cache Size for the JDBC connection pool in one of the following ways:

- Enter a Statement Cache Size value in the Edit Connection Pool Advanced Attributes page in the Administration Console. For more information, click the Help button in the Administration Console.
- Specify the `--statementcachesize` option in the `asadmin create-jdbc-connection-pool` command. For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.
- Specify the `statement-cache-size` option in the `asadmin set` command. For example:

  ```
  asadmin set domain1.resources.jdbc-connection-pool.DerbyPool.statement-cache-size=10
  ```

  For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

By default, this attribute is set to zero and the statement caching is turned off. To enable statement caching, you can set any positive nonzero value. The built-in cache eviction strategy is LRU-based (Least Recently Used). When a connection pool is flushed, the connections in the statement cache are recreated.

Statement Tracing

You can trace the SQL statements executed by applications that use a JDBC connection pool. Set the SQL Trace Listeners attribute to a comma-separated list of trace listener implementation classes in one of the following ways:

- Enter an SQL Trace Listeners value in the Edit Connection Pool Advanced Attributes page in the Administration Console. For more information, click the Help button in the Administration Console.
- Specify the `--sqltracelisteners` option in the `asadmin create-jdbc-connection-pool` command. For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.
- Specify the `sql-trace-listeners` option in the `asadmin set` command. For example:

```
asadmin set domain1.resources.jdbc-connection-pool.DerbyPool.sql-trace-listeners=listeners
```

For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

The Enterprise Server provides a public interface, `org.glassfish.api.jdbc.SQLTraceListener`, that implements a means of recording `SQLTraceRecord` objects. To make custom implementations of this interface available to the Enterprise Server, place the implementation classes in *as-install*`/lib`.

The Enterprise Server provides an SQL tracing logger to log the SQL operations in the form of `SQLTraceRecord` objects in the `server.log` file. The module name under which the SQL operation is logged is `javax.enterprise.resource.sqltrace`. SQL traces are logged as FINE messages along with the module name to enable easy filtering of the SQL logs. A sample SQL trace record looks like this:

```
[#|2009-11-27T15:46:52.202+0530|FINE|glassfishv3.0|javax.enterprise.resource.sqltrace.com.sun.gjc.util
|_ThreadID=29;_ThreadName=Thread-1;ClassName=com.sun.gjc.util.SQLTraceLogger;MethodName=sqlTrace;
|ThreadID=77 | ThreadName=p: thread-pool-1; w: 6 | TimeStamp=1259317012202
| ClassName=com.sun.gjc.spi.jdbc40.PreparedStatementWrapper40 | MethodName=executeUpdate
| arg[0]=insert into table1(colName) values(100) | arg[1]=columnNames | |#]
```

This trace shows that an `executeUpdate(String sql, String columnNames)` operation is being done.

Connections

The following features pertain to connections:

Disabling Pooling

To disable connection pooling, set the Pooling attribute to false. The default is true. You can enable or disable connection pooling in one of the following ways:

- Enter a Pooling value in the Edit Connection Pool Advanced Attributes page in the Administration Console. For more information, click the Help button in the Administration Console.
- Specify the `--pooling` option in the `asadmin create-jdbc-connection-pool` command. For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.
- Specify the `pooling` option in the `asadmin set` command. For example:

  ```
  asadmin set domain1.resources.jdbc-connection-pool.DerbyPool.pooling=false
  ```

  For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

The `pooling` option and the system property `com.sun.enterprise.connectors.SwitchoffACCConnectionPooling`, which turns off connection pooling in the Application Client Container, do not affect each other.

An exception is thrown if `associate-with-thread` is set to `true` and pooling is disabled. An exception is thrown if you attempt to flush a connection pool when pooling is disabled. A warning is logged if the following attributes are used, because they are useful only in a pooled environment:

- `connection-validation`
- `validate-atmost-once-period`
- `match-connections`
- `max-connection-usage`
- `idle-timeout`

Associating Connections with Threads

To associate connections with a thread, set the Associate With Thread attribute to `true`. The default is `false`. A `true` setting allows connections to be saved as `ThreadLocal` in the calling thread. Connections get reclaimed only when the calling thread dies or when the calling thread

is not in use and the pool has run out of connections. If the setting is `false`, the thread must obtain a connection from the pool each time the thread requires a connection.

The Associate With Thread attribute associates connections with a thread such that when the same thread is in need of connections, it can reuse the connections already associated with that thread. In this case, the overhead of getting connections from the pool is avoided. However, when this value is set to `true`, you should verify that the value of the Max Pool Size attribute is comparable to the Max Thread Pool Size attribute of the thread pool. If the Max Thread Pool Size value is much higher than the Max Pool Size value, a lot of time is spent associating connections with a new thread after dissociating them from an older one. Use this attribute in cases where the thread pool should reuse connections to avoid this overhead.

You can set the Associate With Thread attribute in the following ways:

- Enter an Associate With Thread value in the Edit Connection Pool Advanced Attributes page in the Administration Console. For more information, click the Help button in the Administration Console.
- Specify the `--associatewiththread` option in the `asadmin create-jdbc-connection-pool` command. For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.
- Specify the `associate-with-thread` option in the `asadmin set` command. For example:

```
asadmin set domain1.resources.jdbc-connection-pool.DerbyPool.associate-with-thread=true
```

For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Custom Connection Validation

You can specify a custom implementation for Connection Validation that is faster or optimized for a specific database. Set the Validation Method attribute to the value `custom-validation`. (Other validation methods available are `table` (the default), `auto-commit`, and `meta-data`.) The Enterprise Server provides a public interface, `org.glassfish.api.jdbc.ConnectionValidation`, which you can implement to plug in your implementation. A new attribute, Validation Classname, specifies the fully qualified name of the class that implements the `ConnectionValidation` interface. The Validation Classname attribute is required if Connection Validation is enabled and Validation Method is set to Custom Validation.

To enable this feature, set Connection Validation, Validation Method, and Validation Classname for the JDBC connection pool in one of the following ways:

- Enter Connection Validation, Validation Method, and Validation Classname values in the Edit Connection Pool Advanced Attributes page in the Administration Console. You can select from among validation class names for common databases in the Validation Classname field. For more information, click the Help button in the Administration Console.

- Specify the `--isconnectionvalidatereq`, `--validationmethod`, and `--validationclassname` options in the `asadmin create-jdbc-connection-pool` command. For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.
- Specify the `is-connection-validation-required`, `connection-validation-method`, and `validation-classname` options in the `asadmin set` command. For example:

```
asadmin set domain1.resources.jdbc-connection-pool.MyPool.is-connection-validation-required=true
asadmin set domain1.resources.jdbc-connection-pool.MyPool.connection-validation-method=custom-validation
asadmin set domain1.resources.jdbc-connection-pool.MyPool.validation-classname=impl-class
```

  For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

By default, optimized validation mechanisms are provided for Java DB, MySQL, Oracle, and PostgreSQL databases. Additionally, for JDBC 4.0 compliant database drivers, a validation mechanism is provided that uses the `Connection.isValid(0)` implementation.

Sharing Connections

When multiple connections acquired by an application use the same JDBC resource, the connection pool provides connection sharing within the same transaction scope. For example, suppose Bean A starts a transaction and obtains a connection, then calls a method in Bean B. If Bean B acquires a connection to the same JDBC resource with the same sign-on information, and if Bean A completes the transaction, the connection can be shared.

Connections obtained through a resource are shared only if the resource reference declared by the Java EE component allows it to be shareable. This is specified in a component's deployment descriptor by setting the `res-sharing-scope` element to `Shareable` for the particular resource reference. To turn off connection sharing, set `res-sharing-scope` to `Unshareable`.

For general information about connections and JDBC URLs, see Chapter 14, "Administering Database Connectivity," in *Sun GlassFish Enterprise Server v3 Administration Guide*.

Marking Bad Connections

The `DataSource` implementation in the Enterprise Server provides a `markConnectionAsBad` method. A marked bad connection is removed from its connection pool when it is closed. The method signature is as follows:

```
public void markConnectionAsBad(java.sql.Connection con)
```

For example:

```
com.sun.appserv.jdbc.DataSource ds=
   (com.sun.appserv.jdbc.DataSource)context.lookup("dataSource");
Connection con = ds.getConnection();
Statement stmt = null;
try{
```

```
    stmt = con.createStatement();
    stmt.executeUpdate("Update");
}
catch (BadConnectionException e){
    ds.markConnectionAsBad(con) //marking it as bad for removal
}
finally{
    stmt.close();
    con.close(); //Connection will be destroyed during close.
}
```

Handling Invalid Connections

If a `ConnectionErrorOccured` event occurs, the Enterprise Server considers the connection invalid and removes the connection from the connection pool. Typically, a JDBC driver generates a `ConnectionErrorOccured` event when it finds a `ManagedConnection` object unusable. Reasons can be database failure, network failure with the database, fatal problems with the connection pool, and so on.

If the `fail-all-connections` setting in the connection pool configuration is set to `true`, and a single connection fails, all connections are closed and recreated. If this setting is `false`, individual connections are recreated only when they are used. The default is `false`.

The `is-connection-validation-required` setting specifies whether connections have to be validated before being given to the application. If a resource's validation fails, it is destroyed, and a new resource is created and returned. The default is `false`.

The `prefer-validate-over-recreate` property specifies that validating idle connections is preferable to closing them. This property has no effect on non-idle connections. If set to `true`, idle connections are validated during pool resizing, and only those found to be invalid are destroyed and recreated. If `false`, all idle connections are destroyed and recreated during pool resizing. The default is `false`.

You can set the `fail-all-connections`, `is-connection-validation-required`, and `prefer-validate-over-recreate` configuration settings during creation of a JDBC connection pool. Or, you can use the `asadmin set` command to dynamically reconfigure a setting. For example:

```
asadmin set server.resources.jdbc-connection-pool.JCPool1.fail-all-connections="true"
asadmin set server.resources.jdbc-connection-pool.JCPool1.is-connection-validation-required="true"
asadmin set server.resources.jdbc-connection-pool.JCPool1.property.prefer-validate-over-recreate="true"
```

For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

The interface `ValidatingManagedConnectionFactory` exposes the method `getInvalidConnections` to allow retrieval of the invalid connections. The Enterprise Server checks if the JDBC driver implements this interface, and if it does, invalid connections are removed when the connection pool is resized.

Connection Wrapping

The following features pertain to connection wrapping:

Wrapping Connections

If the Wrap JDBC Objects option is `true` (the default), wrapped JDBC objects are returned for `Statement`, `PreparedStatement`, `CallableStatement`, `ResultSet`, and `DatabaseMetaData`.

This option ensures that `Statement.getConnection()` is the same as `DataSource.getConnection()`. Therefore, this option should be `true` when both `Statement.getConnection()` and `DataSource.getConnection()` are done.

You can specify the Wrap JDBC Objects option in the following ways:

- Check or uncheck the Wrap JDBC Objects box on the Edit Connection Pool Advanced Attributes page in the Administration Console. For more information, click the Help button in the Administration Console.
- Specify the `--wrapjdbcobjects` option in the `asadmin create-jdbc-connection-pool` command. For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Obtaining a Physical Connection From a Wrapped Connection

The `DataSource` implementation in the Enterprise Server provides a `getConnection` method that retrieves the JDBC driver's `SQLConnection` from the Enterprise Server's `Connection` wrapper. The method signature is as follows:

```
public java.sql.Connection getConnection(java.sql.Connection con)
throws java.sql.SQLException
```

For example:

```
InitialContext ctx = new InitialContext();
com.sun.appserv.jdbc.DataSource ds = (com.sun.appserv.jdbc.DataSource)
   ctx.lookup("jdbc/MyBase");
Connection con = ds.getConnection();
Connection drivercon = ds.getConnection(con); //get physical connection from wrapper
// Do db operations.
// Do not close driver connection.
con.close(); // return wrapped connection to pool.
```

Using the `Connection.unwrap()` Method

If the JDK version 1.6 is used, the Enterprise Server supports JDBC 4.0 if the JDBC driver is JDBC 4.0 compliant. Using the `Connection.unwrap()` method on a vendor-provided interface returns an object or a wrapper object implementing the vendor-provided interface, which the application can make use of to do vendor-specific database operations. Use the `Connection.isWrapperFor()` method on a vendor-provided interface to check whether the connection can provide an implementation of the vendor-provided interface. Check the JDBC driver vendor's documentation for information on these interfaces.

Transactions

The following features pertain to transactions:

- "Using Non-Transactional Connections" on page 261
- "Using JDBC Transaction Isolation Levels" on page 262

Using Non-Transactional Connections

You can specify a non-transactional database connection in any of these ways:

- Check the Non-Transactional Connections box on the New JDBC Connection Pool or Edit Connection Pool page in the Administration Console. The default is unchecked. For more information, click the Help button in the Administration Console.
- Specify the `--nontransactionalconnections` option in the `asadmin create-jdbc-connection-pool` command. For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.
- Specify the `non-transactional-connections` option in the `asadmin set` command. For example:

```
asadmin set domain1.resources.jdbc-connection-pool.DerbyPool.non-transactional-connections=true
```

  For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.
- Use the `DataSource` implementation in the Enterprise Server, which provides a `getNonTxConnection` method. This method retrieves a JDBC connection that is not in the scope of any transaction. There are two variants.

```
public java.sql.Connection getNonTxConnection() throws java.sql.SQLException

public java.sql.Connection getNonTxConnection(String user, String password)
   throws java.sql.SQLException
```

- Create a resource with the JNDI name ending in `__nontx`. This forces all connections looked up using this resource to be non transactional.

Typically, a connection is enlisted in the context of the transaction in which a `getConnection` call is invoked. However, a non-transactional connection is not enlisted in a transaction context even if a transaction is in progress.

The main advantage of using non-transactional connections is that the overhead incurred in enlisting and delisting connections in transaction contexts is avoided. However, use such connections carefully. For example, if a non-transactional connection is used to query the database while a transaction is in progress that modifies the database, the query retrieves the unmodified data in the database. This is because the in-progress transaction hasn't committed. For another example, if a non-transactional connection modifies the database and a transaction that is running simultaneously rolls back, the changes made by the non-transactional connection are not rolled back.

Here is a typical use case for a non-transactional connection: a component that is updating a database in a transaction context spanning over several iterations of a loop can refresh cached data by using a non-transactional connection to read data before the transaction commits.

Using JDBC Transaction Isolation Levels

For general information about transactions, see Chapter 15, "Using the Transaction Service," and Chapter 21, "Administering Transactions," in *Sun GlassFish Enterprise Server v3 Administration Guide*. For information about last agent optimization, which can improve performance, see "Transaction Scope" on page 266.

Not all database vendors support all transaction isolation levels available in the JDBC API. The Enterprise Server permits specifying any isolation level your database supports. The following table defines transaction isolation levels.

TABLE 14–1 Transaction Isolation Levels

Transaction Isolation Level	Description
`read-uncommitted`	Dirty reads, non-repeatable reads, and phantom reads can occur.
`read-committed`	Dirty reads are prevented; non-repeatable reads and phantom reads can occur.
`repeatable-read`	Dirty reads and non-repeatable reads are prevented; phantom reads can occur.
`serializable`	Dirty reads, non-repeatable reads and phantom reads are prevented.

You can specify the transaction isolation level in the following ways:

- Select the value from the Transaction Isolation drop-down list on the New JDBC Connection Pool or Edit Connection Pool page in the Administration Console. For more information, click the Help button in the Administration Console.
- Specify the `--isolationlevel` option in the `asadmin create-jdbc-connection-pool` command. For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.
- Specify the `transaction-isolation-level` option in the `asadmin set` command. For example:

```
asadmin set domain1.resources.jdbc-connection-pool.DerbyPool.transaction-isolation-level=serializable
```

For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Note that you cannot call `setTransactionIsolation()` during a transaction.

You can set the default transaction isolation level for a JDBC connection pool. For details, see "Creating a JDBC Connection Pool" on page 251.

To verify that a level is supported by your database management system, test your database programmatically using the `supportsTransactionIsolationLevel()` method in `java.sql.DatabaseMetaData`, as shown in the following example:

```
InitialContext ctx = new InitialContext();
DataSource ds = (DataSource)
ctx.lookup("jdbc/MyBase");
Connection con = ds.getConnection();
DatabaseMetaData dbmd = con.getMetaData();
if (dbmd.supportsTransactionIsolationLevel(TRANSACTION_SERIALIZABLE)
{ Connection.setTransactionIsolation(TRANSACTION_SERIALIZABLE); }
```

For more information about these isolation levels and what they mean, see the JDBC API specification.

Note – Applications that change the isolation level on a pooled connection programmatically risk polluting the pool, which can lead to errors.

Other Features

The following additional features related to JDBC are provided:

- "Allowing Non-Component Callers" on page 263

Allowing Non-Component Callers

You can allow non-Java-EE components, such as servlet filters, lifecycle modules, and third party persistence managers, to use this JDBC connection pool. The returned connection is automatically enlisted with the transaction context obtained from the transaction manager. Standard Java EE components can also use such pools. Connections obtained by non-component callers are not automatically closed at the end of a transaction by the container. They must be explicitly closed by the caller.

You can enable non-component callers in the following ways:

- Check the Allow Non Component Callers box on the Edit Connection Pool Advanced Attributes page in the Administration Console. The default is `false`. For more information, click the Help button in the Administration Console.
- Specify the `--allownoncomponentcallers` option in the `asadmin create-jdbc-connection-pool` command. For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

- Specify the `allow-non-component-callers` option in the `asadmin set` command. For example:

```
asadmin set domain1.resources.jdbc-connection-pool.DerbyPool.allow-non-component-callers=true
```

  For more information, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

- Create a JDBC resource with a `__pm` suffix.

Accessing a `DataSource` using the `Synchronization.beforeCompletion()` method requires setting Allow Non Component Callers to `true`. For more information about the Transaction Synchronization Registry, see "The Transaction Manager, the Transaction Synchronization Registry, and `UserTransaction`" on page 267.

Restrictions and Optimizations

This section discusses restrictions and performance optimizations that affect using the JDBC API.

Disabling Stored Procedure Creation on Sybase

By default, DataDirect and Sun GlassFish JDBC drivers for Sybase databases create a stored procedure for each parameterized `PreparedStatement`. On the Enterprise Server, exceptions are thrown when primary key identity generation is attempted. To disable the creation of these stored procedures, set the property `PrepareMethod=direct` for the JDBC connection pool.

◆ ◆ ◆ CHAPTER 15

Using the Transaction Service

The Java EE platform provides several abstractions that simplify development of dependable transaction processing for applications. This chapter discusses Java EE transactions and transaction support in the Sun GlassFish Enterprise Server.

This chapter contains the following sections:

For more information about the Java Transaction API (JTA) and Java Transaction Service (JTS), see Chapter 21, "Administering Transactions," in *Sun GlassFish Enterprise Server v3 Administration Guide* and the following sites: `http://java.sun.com/products/jta/` and `http://java.sun.com/products/jts/`.

You might also want to read Chapter 27, "Transactions," in *The Java EE 6 Tutorial, Volume I*.

Transaction Resource Managers

There are three types of transaction resource managers:

- Databases - Use of transactions prevents databases from being left in inconsistent states due to incomplete updates. For information about JDBC transaction isolation levels, see "Using JDBC Transaction Isolation Levels" on page 262.

The Enterprise Server supports a variety of JDBC XA drivers. For a list of the JDBC drivers currently supported by the Enterprise Server, see the *Sun GlassFish Enterprise Server v3 Release Notes*. For configurations of supported and other drivers, see "Configuration Specifics for JDBC Drivers" in *Sun GlassFish Enterprise Server v3 Administration Guide*.

- Java Message Service (JMS) Providers - Use of transactions ensures that messages are reliably delivered. The Enterprise Server is integrated with Sun GlassFish Message Queue, a fully capable JMS provider. For more information about transactions and the JMS API, see Chapter 17, "Using the Java Message Service."
- J2EE Connector Architecture (CA) components - Use of transactions prevents legacy EIS systems from being left in inconsistent states due to incomplete updates. For more information about connectors, see Chapter 12, "Developing Connectors."

For details about how transaction resource managers, the transaction service, and applications interact, see Chapter 21, "Administering Transactions," in *Sun GlassFish Enterprise Server v3 Administration Guide*.

Transaction Scope

A *local* transaction involves only one non-XA resource and requires that all participating application components execute within one process. Local transaction optimization is specific to the resource manager and is transparent to the Java EE application.

In the Enterprise Server, a JDBC resource is non-XA if it meets either of the following criteria:

- In the JDBC connection pool configuration, the DataSource class does not implement the `javax.sql.XADataSource` interface.
- The Resource Type setting is not set to `javax.sql.XADataSource`.

A transaction remains local if the following conditions remain true:

- One and only one non-XA resource is used. If any additional non-XA resource is used, the transaction is aborted.
- No transaction importing or exporting occurs.

Transactions that involve multiple resources or multiple participant processes are *distributed* or *global* transactions. A global transaction can involve one non-XA resource if last agent optimization is enabled. Otherwise, all resourced must be XA. The `use-last-agent-optimization` property is set to `true` by default. For details about how to set this property, see "Configuring the Transaction Service" on page 267.

If only one XA resource is used in a transaction, one-phase commit occurs, otherwise the transaction is coordinated with a two-phase commit protocol.

A two-phase commit protocol between the transaction manager and all the resources enlisted for a transaction ensures that either all the resource managers commit the transaction or they all

abort. When the application requests the commitment of a transaction, the transaction manager issues a PREPARE_TO_COMMIT request to all the resource managers involved. Each of these resources can in turn send a reply indicating whether it is ready for commit (PREPARED) or not (NO). Only when all the resource managers are ready for a commit does the transaction manager issue a commit request (COMMIT) to all the resource managers. Otherwise, the transaction manager issues a rollback request (ABORT) and the transaction is rolled back.

Configuring the Transaction Service

You can configure the transaction service in the Enterprise Server in the following ways:

- To configure the transaction service using the Administration Console, open the Transaction Service component under the relevant configuration. For details, click the Help button in the Administration Console.
- To configure the transaction service, use the asadmin set command to set the following attributes.

```
server-config.transaction-service.automatic-recovery = false
server-config.transaction-service.heuristic-decision = rollback
server-config.transaction-service.keypoint-interval = 2048
server-config.transaction-service.retry-timeout-in-seconds = 600
server-config.transaction-service.timeout-in-seconds = 0
server-config.transaction-service.tx-log-dir = domain-dir/logs
```

You can also set these properties:

```
server-config.transaction-service.property.oracle-xa-recovery-workaround = false
server-config.transaction-service.property.disable-distributed-transaction-logging = false
server-config.transaction-service.property.xaresource-txn-timeout = 600
server-config.transaction-service.property.pending-txn-cleanup-interval = 60
server-config.transaction-service.property.use-last-agent-optimization = true
server-config.transaction-service.property.db-logging-resource = jdbc/TxnDS
server-config.transaction-service.property.xa-servername = myserver
```

You can use the asadmin get command to list all the transaction service attributes and properties. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Changing keypoint-interval, retry-timeout-in-seconds, or timeout-in-seconds does not require a server restart. Changing other attributes or properties requires a server restart.

The Transaction Manager, the Transaction Synchronization Registry, and UserTransaction

To access a UserTransaction instance, you can either look it up using the java:comp/UserTransaction JNDI name or inject it using the @Resource annotation.

If you need to access the javax.transaction.TransactionManager implementation, you can look up the Enterprise Server implementation of this interface using the JNDI name

`java:appserver/TransactionManager`. If possible, you should use the `javax.transaction.TransactionSynchronizationRegistry` interface instead, for portability. You can look up the implementation of this interface by using the JNDI name `java:comp/TransactionSynchronizationRegistry`. For details, see the Javadoc page for Interface TransactionSynchronizationRegistry (`http://java.sun.com/javaee/5/docs/api/javax/transaction/TransactionSynchronizationRegistry.html`) and Java Specification Request (JSR) 907 (`http://www.jcp.org/en/jsr/detail?id=907`).

Accessing a `DataSource` using the `Synchronization.beforeCompletion()` method requires setting Allow Non Component Callers to `true`. The default is `false`. For more information about non-component callers, see "Allowing Non-Component Callers" on page 263.

Transaction Logging

The transaction service writes transactional activity into transaction logs so that transactions can be recovered. You can control transaction logging in these ways:

- Set the location of the transaction log files using the Transaction Log Location setting in the Administration Console, or set the `tx-log-dir` attribute using the `asadmin set` command.
- Turn off transaction logging by setting the `disable-distributed-transaction-logging` property to `true` and the `automatic-recovery` attribute to `false`. Do this *only* if performance is more important than transaction recovery.

Storing Transaction Logs in a Database

For multi-core machines, logging transactions to a database may be more efficient.

To log transactions to a database, follow these steps:

1. Create a JDBC connection Pool, and set the `non-transactional-connections` attribute to `true`.
2. Create a JDBC resource that uses the connection pool and note the JNDI name of the JDBC resource.
3. Create a table named `txn_log_table` with the schema shown in Table 15–1.
4. Add the `db-logging-resource` property to the transaction service. For example:

```
asadmin set server-config.transaction-service.property.db-logging-resource="jdbc/TxnDS"
```

   The property's value should be the JNDI name of the JDBC resource configured previously.

5. To disable file synchronization, use the following `asadmin create-jvm-options` command:

   ```
   asadmin create-jvm-options -Dcom.sun.appserv.transaction.nofdsync
   ```

6. Restart the server.

For information about JDBC connection pools and resources, see Chapter 14, "Using the JDBC API for Database Access." For more information about the `asadmin create-jvm-options` command, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

TABLE 15–1 Schema for `txn_log_table`

Column Name	JDBC Type
`LOCALTID`	`BIGINT`
`SERVERNAME`	`VARCHAR(n)`
`GTRID`	`VARBINARY`

The size of the `SERVERNAME` column should be at least the length of the Enterprise Server host name plus 10 characters.

The size of the `GTRID` column should be at least 64 bytes.

To define the SQL used by the transaction manager when it is storing its transaction logs in the database, use the following flags:

```
-Dcom.sun.jts.dblogging.insertquery=sql statement
-Dcom.sun.jts.dblogging.deletequery=sql statement
```

The default statements are as follows:

```
-Dcom.sun.jts.dblogging.insertquery=insert into txn_log_table values ( ?, ? , ? )
-Dcom.sun.jts.dblogging.deletequery=delete from txn_log_table where localtid = ? and servername = ?
```

To set one of these flags using the `asadmin create-jvm-options` command, you must quote the statement. For example:

```
create-jvm-options '-Dcom.sun.jts.dblogging.deletequery=delete from txn_log_table where gtrid = ?'
```

You can also set JVM options in the Administration Console. Select the Application Server component and the JVM Settings tab. These flags and their statements must also be quoted in the Administration Console. For example:

```
'-Dcom.sun.jts.dblogging.deletequery=delete from txn_log_table where gtrid = ?'
```

Recovery Workarounds and Limitations

The Enterprise Server provides workarounds for some known issues with transaction recovery implementations.

Note – These workarounds do not imply support for any particular JDBC driver.

Oracle Thin Driver

In the Oracle thin driver, the `XAResource.recover` method repeatedly returns the same set of in-doubt Xids regardless of the input flag. According to the XA specifications, the Transaction Manager initially calls this method with TMSTARTSCAN and then with TMNOFLAGS repeatedly until no Xids are returned. The `XAResource.commit` method also has some issues.

To disable the Enterprise Server workaround, set the `oracle-xa-recovery-workaround` property value to `false`. For details about how to set this property, see "Configuring the Transaction Service" on page 267. This workaround is used unless explicitly disabled.

Manual Transaction Recovery Limitation

Manual transaction recovery cannot recover transactions after a server crash. Manual operations are intended for cases when a resource dies unexpectedly while the server is running. In case of a server crash, only start-up recovery can recover in-doubt transactions.

◆ ◆ ◆ CHAPTER 16

Using the Java Naming and Directory Interface

A *naming service* maintains a set of bindings, which relate names to objects. The Java EE naming service is based on the Java Naming and Directory Interface (JNDI) API. The JNDI API allows application components and clients to look up distributed resources, services, and EJB components. For general information about the JNDI API, see `http://java.sun.com/products/jndi/`.

You can also see the JNDI tutorial at `http://java.sun.com/products/jndi/tutorial/`.

This chapter contains the following sections:

Note – The Web Profile of the Enterprise Server supports the EJB 3.1 Lite specification, which allows enterprise beans within web applications, among other features. The full Enterprise Server supports the entire EJB 3.1 specification. For details, see JSR 318 (`http://jcp.org/en/jsr/detail?id=318`).

Accessing the Naming Context

The Sun GlassFish Enterprise Server provides a naming environment, or *context*, which is compliant with standard Java EE requirements. A `Context` object provides the methods for binding names to objects, unbinding names from objects, renaming objects, and listing the bindings. The `InitialContext` is the handle to the Java EE naming service that application components and clients use for lookups.

The JNDI API also provides subcontext functionality. Much like a directory in a file system, a subcontext is a context within a context. This hierarchical structure permits better organization

of information. For naming services that support subcontexts, the `Context` class also provides methods for creating and destroying subcontexts.

The rest of this section covers these topics:

Note – Each resource within the server must have a unique name.

Global JNDI Names

Global JNDI names are assigned according to the following precedence rules:

1. A global JNDI name assigned in the `sun-ejb-jar.xml`, `sun-web.xml`, or `sun-application-client.xml` deployment descriptor file has the highest precedence. See "Mapping References" on page 278.
2. A global JNDI name assigned in a `mapped-name` element in the `ejb-jar.xml`, `web.xml`, or `application-client.xml` deployment descriptor file has the second highest precedence. The following elements have `mapped-name` subelements: `resource-ref`, `resource-env-ref`, `ejb-ref`, `message-destination`, `message-destination-ref`, `session`, `message-driven`, and `entity`.
3. A global JNDI name assigned in a `mappedName` attribute of an annotation has the third highest precedence. The following annotations have `mappedName` attributes: `@javax.annotation.Resource`, `@javax.ejb.EJB`, `@javax.ejb.Stateless`, `@javax.ejb.Stateful`, and `@javax.ejb.MessageDriven`.
4. A default global JNDI name is assigned in some cases if no name is assigned in deployment descriptors or annotations.
   - For an EJB 2.x dependency or a session or entity bean with a remote interface, the default is the fully qualified name of the home interface.
   - For an EJB 3.0 dependency or a session bean with a remote interface, the default is the fully qualified name of the remote business interface.
   - If both EJB 2.x and EJB 3.0 remote interfaces are specified, or if more than one 3.0 remote interface is specified, there is no default, and the global JNDI name must be specified.
   - For all other component dependencies that must be mapped to global JNDI names, the default is the name of the dependency relative to `java:comp/env`. For example, in the `@Resource(name="jdbc/Foo") DataSource ds;` annotation, the global JNDI name is `jdbc/Foo`.

Accessing EJB Components Using the `CosNaming` Naming Context

The preferred way of accessing the naming service, even in code that runs outside of a Java EE container, is to use the no-argument `InitialContext` constructor. However, if EJB client code explicitly instantiates an `InitialContext` that points to the CosNaming naming service, it is necessary to set the `java.naming.factory.initial` property to `com.sun.jndi.cosnaming.CNCtxFactory` in the client JVM software when accessing EJB components. You can set this property as a command-line argument, as follows:

```
-Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
```

Or you can set this property in the code, as follows:

```
Properties properties = null;
      try {
            properties = new Properties();
            properties.put("java.naming.factory.initial",
                  "com.sun.jndi.cosnaming.CNCtxFactory");
      ...
```

Accessing EJB Components in a Remote Enterprise Server

The recommended approach for looking up an EJB component in a remote Enterprise Server from a client that is a servlet or EJB component is to use the Interoperable Naming Service syntax. Host and port information is prepended to any global JNDI names and is automatically resolved during the lookup. The syntax for an interoperable global name is as follows:

`corbaname:iiop:`*host*`:`*port*`#a/b/`*name*

This makes the programming model for accessing EJB components in another Enterprise Server exactly the same as accessing them in the same server. The deployer can change the way the EJB components are physically distributed without having to change the code.

For Java EE components, the code still performs a `java:comp/env` lookup on an EJB reference. The only difference is that the deployer maps the `ejb-reference` element to an interoperable name in an Enterprise Server deployment descriptor file instead of to a simple global JNDI name.

For example, suppose a servlet looks up an EJB reference using `java:comp/env/ejb/Foo`, and the target EJB component has a global JNDI name of `a/b/Foo`.

The `ejb-ref` element in `sun-web.xml` looks like this:

```
<ejb-ref>
   <ejb-ref-name>ejb/Foo</ejb-ref-name>
   <jndi-name>corbaname:iiop:host:port#a/b/Foo</jndi-name>
<ejb-ref>
```

The code looks like this:

```
Context ic = new InitialContext();
Object o = ic.lookup("java:comp/env/ejb/Foo");
```

For a client that doesn't run within a Java EE container, the code just uses the interoperable global name instead of the simple global JNDI name. For example:

```
Context ic = new InitialContext();
Object o = ic.lookup("corbaname:iiop:host:port#a/b/Foo");
```

Objects stored in the interoperable naming context and component-specific (`java:comp/env`) naming contexts are transient. On each server startup or application reloading, all relevant objects are re-bound to the namespace.

Naming Environment for Lifecycle Modules

Lifecycle listener modules provide a means of running short or long duration tasks based on Java technology within the application server environment, such as instantiation of singletons or RMI servers. These modules are automatically initiated at server startup and are notified at various phases of the server life cycle. For details about lifecycle modules, see Chapter 13, "Developing Lifecycle Listeners."

The configured properties for a lifecycle module are passed as properties during server initialization (the `INIT_EVENT`). The initial JNDI naming context is not available until server initialization is complete. A lifecycle module can get the `InitialContext` for lookups using the method `LifecycleEventContext.getInitialContext()` during, and only during, the `STARTUP_EVENT`, `READY_EVENT`, or `SHUTDOWN_EVENT` server life cycle events.

Configuring Resources

The Enterprise Server exposes the following special resources in the naming environment. Full administration details are provided in the following sections:

External JNDI Resources

An external JNDI resource defines custom JNDI contexts and implements the `javax.naming.spi.InitialContextFactory` interface. There is no specific JNDI parent context for external JNDI resources, except for the standard `java:comp/env/`.

Create an external JNDI resource in one of these ways:

- To create an external JNDI resource using the Administration Console, open the Resources component, open the JNDI component, and select External Resources. For details, click the Help button in the Administration Console.
- To create an external JNDI resource, use the `asadmin create-jndi-resource` command. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Custom Resources

A custom resource specifies a custom server-wide resource object factory that implements the `javax.naming.spi.ObjectFactory` interface. There is no specific JNDI parent context for external JNDI resources, except for the standard `java:comp/env/`.

Create a custom resource in one of these ways:

- To create a custom resource using the Administration Console, open the Resources component, open the JNDI component, and select Custom Resources. For details, click the Help button in the Administration Console.
- To create a custom resource, use the `asadmin create-custom-resource` command. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Built- in Factories for Custom Resources

The Enterprise Server provides built-in factories for the following types of custom resources:

- "JavaBeanFactory" on page 275
- "PropertiesFactory" on page 276
- "PrimitiviesAndStringFactory" on page 276
- "URLFactory" on page 277

Template `sun-resources.xml` files for these built-in factories and a `README` file are available at *as-install*`/lib/install/templates/resources/custom/`. For more information about the `sun-resources.xml` file, see the *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

JavaBeanFactory

To create a custom resource that provides instances of a JavaBean class, follow these steps:

1. Set the custom resource's factory class to `org.glassfish.resources.custom.factory.JavaBeanFactory`.
2. Create a property in the custom resource for each setter method in the JavaBean class.

   For example, if the JavaBean class has a method named `setAccount`, specify a property named `account` and give it a value.
3. Make sure the JavaBean class is accessible to the Enterprise Server.

   For example, you can place the JavaBean class in the *as-install*`/lib` directory.

PropertiesFactory

To create a custom resource that provides properties to applications, set the custom resource's factory class to `org.glassfish.resources.custom.factory.PropertiesFactory`, then specify one or both of the following:

- Create a property in the custom resource named `file-location` and specify as its value the path to a properties file or an XML file.

  The path can be absolute or relative to *as-install*. The file must be accessible to the Enterprise Server.

  If an XML file is specified, it must match the document type definition (DTD) specified in the API definition of java.util.Properties (`http://java.sun.com/javase/6/docs/api/java/util/Properties.html`).
- Create the desired properties directly as properties of the custom resource.

  If both the `file-location` property and other properties are specified, the resulting property set is the union. If the same property is defined in the file and directly in the custom resource, the value of the latter takes precedence.

PrimitivesAndStringFactory

To create a custom resource that provides Java primitives to applications, follow these steps:

1. Set the custom resource's factory class to `org.glassfish.resources.custom.factory.PrimitivesAndStringFactory`.
2. Set the custom resource's resource type to one of the following or its fully qualified wrapper class name equivalent:
   - `int`
   - `integer`
   - `long`
   - `double`
   - `float`
   - `char`
   - `character`
   - `short`

- byte
- boolean
- String

3. Create a property in the custom resource named value and give it the value needed by the application.

   For example, If the application requires a double of value 22.1, create a property with the name value and the value 22.1.

URLFactory

To create a custom resource that provides URL instances to applications, follow these steps:

1. Set the custom resource's factory class to org.glassfish.resources.custom.factory.URLFactory.
2. Choose which of the following constructors to use:
   - URL(protocol, host, port, file)
   - URL(protocol, host, file)
   - URL(spec)
3. Define properties according to the chosen constructor.

   For example, for the first constructor, define properties named protocol, host, port, and file. Example values might be http, localhost, 8085, and index.html, respectively.

   For the third constructor, define a property named spec and assign it the value of the entire URL.

Using a Custom jndi.properties File

To use a custom jndi.properties file, place the file in the *domain-dir*/lib/classes directory or JAR it and place it in the *domain-dir*/lib directory. This adds the custom jndi.properties file to the Common class loader. For more information about class loading, see Chapter 2, "Class Loaders."

For each property found in more than one jndi.properties file, the Java EE naming service either uses the first value found or concatenates all of the values, whichever makes sense.

Mapping References

The following XML elements in the Enterprise Server deployment descriptors map resource references in application client, EJB, and web application components to JNDI names configured in the Enterprise Server:

- `resource-env-ref` - Maps the `@Resource` or `@Resources` annotation (or the `resource-env-ref` element in the corresponding Java EE XML file) to the absolute JNDI name configured in the Enterprise Server.
- `resource-ref` - Maps the `@Resource` or `@Resources` annotation (or the `resource-ref` element in the corresponding Java EE XML file) to the absolute JNDI name configured in the Enterprise Server.
- `ejb-ref` - Maps the `@EJB` annotation (or the `ejb-ref` element in the corresponding Java EE XML file) to the absolute JNDI name configured in the Enterprise Server.

  JNDI names for EJB components must be unique. For example, appending the application name and the module name to the EJB name is one way to guarantee unique names. In this case, `mycompany.pkging.pkgingEJB.MyEJB` would be the JNDI name for an EJB in the module `pkgingEJB.jar`, which is packaged in the `pkging.ear` application.

These elements are part of the `sun-web.xml`, `sun-application-client.xml`, and `sun-ejb-ref.xml` deployment descriptor files. For more information about how these elements behave in each of the deployment descriptor files, see Appendix C, "Elements of the Enterprise Server Deployment Descriptors," in *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

The rest of this section uses an example of a JDBC resource lookup to describe how to reference resource factories. The same principle is applicable to all resources (such as JMS destinations, JavaMail sessions, and so on).

The `@Resource` annotation in the application code looks like this:

```
@Resource(name="jdbc/helloDbDs") javax.sql.DataSource ds;
```

This references a resource with the JNDI name of `java:comp/env/jdbc/helloDbDs`. If this is the JNDI name of the JDBC resource configured in the Enterprise Server, the annotation alone is enough to reference the resource.

However, you can use an Enterprise Server specific deployment descriptor to override the annotation. For example, the `resource-ref` element in the `sun-web.xml` file maps the `res-ref-name` (the name specified in the annotation) to the JNDI name of another JDBC resource configured in the Enterprise Server.

```
<resource-ref>
   <res-ref-name>jdbc/helloDbDs</res-ref-name>
   <jndi-name>jdbc/helloDbDataSource</jndi-name>
</resource-ref>
```

◆ ◆ ◆ CHAPTER 17

Using the Java Message Service

This chapter describes how to use the Java Message Service (JMS) API. The Sun GlassFish Enterprise Server has a fully integrated JMS provider: the Sun GlassFish Message Queue software.

Note – JMS resources are supported only in the full Enterprise Server, not in the Web Profile.

For detailed information about JMS concepts and JMS support in the Enterprise Server, see Chapter 19, "Administering the Java Message Service (JMS) ," in *Sun GlassFish Enterprise Server v3 Administration Guide*.

For more information about Message Queue software, see the *Sun GlassFish Message Queue 4.4 Administration Guide*.

This chapter contains the following sections:

The JMS Provider

The Enterprise Server support for JMS messaging, in general, and for message-driven beans, in particular, requires messaging middleware that implements the JMS specification: a JMS provider. The Enterprise Server uses the Sun GlassFish Message Queue software as its native JMS provider. The Message Queue software is tightly integrated into theEnterprise Server, providing transparent JMS messaging support. This support is known within Enterprise Server as the *JMS Service*. The JMS Service requires only minimal administration.

The relationship of the Message Queue software to the Enterprise Server can be one of these types: `EMBEDDED`, `LOCAL`, or `REMOTE`. The effects of these choices on the Message Queue broker life cycle are as follows:

- If the type is `EMBEDDED`, the Enterprise Server and Message Queue software run in the same JVM, and the networking stack is bypassed. The Message Queue broker is started and stopped automatically by the Enterprise Server. This is the default for the Domain Administration Server (DAS).

  Lazy initialization starts the default embedded broker on the first access of JMS services rather than at Enterprise Server startup.
- If the type is `LOCAL`, the Message Queue broker starts when the Enterprise Server starts. This is the default for all Enterprise Server instances except the DAS.

  The `LOCAL` setting implicitly sets up a 1:1 relationship between an Enterprise Server instance and a Message Queue broker.
- If the type is `REMOTE`, the Message Queue broker must be started separately. For information about starting the broker, see the *Sun GlassFish Message Queue 4.4 Administration Guide*.

For more information about setting the type and the default JMS host, see "Configuring the JMS Service" on page 281.

For more information about the Message Queue software, refer to the documentation at `http://docs.sun.com/coll/1343.9`.

For general information about the JMS API, see the JMS web page at `http://java.sun.com/products/jms/index.html`.

Message Queue Resource Adapter

The Sun GlassFish Message Queue software is integrated into the Enterprise Server using a resource adapter that is compliant with the Connector specification. The module name of this system resource adapter is `jmsra`. Every JMS resource is converted to a corresponding connector resource of this resource adapter as follows:

- **Connection Factory** – A connector connection pool with a `max-pool-size` of `250` and a corresponding connector resource

- **Destination (Topic or Queue)** – A connector administered object

You use connector configuration tools to manage JMS resources. For more information, see Chapter 12, "Developing Connectors."

Generic Resource Adapter

The Enterprise Server provides a generic resource adapter for JMS, for those who want to use a JMS provider other than Sun GlassFish Message Queue. For details, see `http://genericjmsra.dev.java.net/` and "Configuring Resource Adapters for JMS" in *Sun GlassFish Enterprise Server v3 Administration Guide*.

Administration of the JMS Service

To configure the JMS Service and prepare JMS resources for use in applications deployed to the Enterprise Server, you must perform these tasks:

- "Configuring the JMS Service" on page 281
- "The Default JMS Host" on page 282
- "Creating JMS Hosts" on page 282
- "Checking Whether the JMS Provider Is Running" on page 283
- "Creating Physical Destinations" on page 283
- "Creating JMS Resources: Destinations and Connection Factories" on page 284

For more information about JMS administration tasks, see Chapter 19, "Administering the Java Message Service (JMS)," in *Sun GlassFish Enterprise Server v3 Administration Guide* and the *Sun GlassFish Message Queue 4.4 Administration Guide*.

Configuring the JMS Service

The JMS Service configuration is available to all inbound and outbound connections pertaining to the Enterprise Server instance. You can edit the JMS Service configuration in the following ways:

- To edit the JMS Service configuration using the Administration Console, open the Java Message Service component under the relevant configuration. For details, click the Help button in the Administration Console.
- To configure the JMS service, use the `asadmin set` command to set the following attributes:

```
server.jms-service.init-timeout-in-seconds = 60
server.jms-service.type = EMBEDDED
server.jms-service.start-args =
server.jms-service.default-jms-host = default_JMS_host
```

```
server.jms-service.reconnect-interval-in-seconds = 60
server.jms-service.reconnect-attempts = 3
server.jms-service.reconnect-enabled = true
server.jms-service.addresslist-behavior = random
server.jms-service.addresslist-iterations = 3
server.jms-service.mq-scheme = mq
server.jms-service.mq-service = jms
```

You can also set these properties:

```
server.jms-service.property.instance-name = imqbroker
server.jms-service.property.instance-name-suffix =
server.jms-service.property.append-version = false
server.jms-service.property.user-name =
server.jms-service.property.password =
```

You can use the `asadmin get` command to list all the JMS service attributes and properties. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

You can override the JMS Service configuration using JMS connection factory settings. For details, see Chapter 19, "Administering the Java Message Service (JMS)," in *Sun GlassFish Enterprise Server v3 Administration Guide*.

Note – The Enterprise Server instance must be restarted after configuration of the JMS Service.

The Default JMS Host

A JMS host refers to a Sun GlassFish Message Queue broker. A default JMS host for the JMS service is provided, named `default_JMS_host`. This is the JMS host that the Enterprise Server uses for performing all Message Queue broker administrative operations, such as creating and deleting JMS destinations.

If you have created a multi-broker cluster in the Message Queue software, delete the default JMS host, then add the Message Queue cluster's brokers as JMS hosts. In this case, the default JMS host becomes the first JMS host in the `AddressList`. For more information about the `AddressList`, see "JMS Connection Features" on page 284. You can also explicitly set the default JMS host; see "Configuring the JMS Service" on page 281.

When the Enterprise Server uses a Message Queue cluster, it executes Message Queue specific commands on the default JMS host. For example, when a physical destination is created for a Message Queue cluster of three brokers, the command to create the physical destination is executed on the default JMS host, but the physical destination is used by all three brokers in the cluster.

Creating JMS Hosts

You can create additional JMS hosts in the following ways:

- Use the Administration Console. Open the Java Message Service component under the relevant configuration, then select the JMS Hosts component. For details, click the Help button in the Administration Console.
- Use the `asadmin create-jms-host` command. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

For machines having more than one host, use the Host field in the Administration Console or the `--mqhost` option of `create-jms-host` to specify the address to which the broker binds.

Checking Whether the JMS Provider Is Running

You can use the `asadmin jms-ping` command to check whether a Sun GlassFish Message Queue instance is running. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

Creating Physical Destinations

Produced messages are delivered for routing and subsequent delivery to consumers using *physical destinations* in the JMS provider. A physical destination is identified and encapsulated by an administered object (a `Topic` or `Queue` destination resource) that an application component uses to specify the destination of messages it is producing and the source of messages it is consuming.

If a message-driven bean is deployed and the `Queue` physical destination it listens to doesn't exist, the Enterprise Server automatically creates the physical destination. However, it is good practice to create the `Queue` physical destination beforehand.

You can create a JMS physical destination in the following ways:

- Use the Administration Console. Open the Resources component, open the JMS Resources component, then select Physical Destinations. For details, click the Help button in the Administration Console.
- Use the `asadmin create-jmsdest` command. This command acts on the default JMS host of its target. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

To purge all messages currently queued at a physical destination, use the `asadmin flush-jmsdest` command. This deletes the messages before they reach any message consumers. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

To create a destination resource, see "Creating JMS Resources: Destinations and Connection Factories" on page 284.

Creating JMS Resources: Destinations and Connection Factories

You can create two kinds of JMS resources in the Enterprise Server:

- **Connection Factories** – administered objects that implement the `ConnectionFactory`, `QueueConnectionFactory`, or `TopicConnectionFactory` interfaces.
- **Destination Resources** – administered objects that implement the `Queue` or `Topic` interfaces.

In either case, the steps for creating a JMS resource are the same. You can create a JMS resource in the following ways:

- To create a JMS resource using the Administration Console, open the Resources component, then open the JMS Resources component. Click Connection Factories to create a connection factory, or click Destination Resources to create a queue or topic. For details, click the Help button in the Administration Console.
- A JMS resource is a type of connector. For more information about connectors, see Chapter 12, "Developing Connectors."

Note – All JMS resource properties that used to work with version 7 of the Enterprise Server are supported for backward compatibility.

Restarting the JMS Client After JMS Configuration

When a JMS client accesses a JMS administered object for the first time, the client JVM retrieves the JMS service configuration from the Enterprise Server. Further changes to the configuration are not available to the client JVM until the client is restarted.

JMS Connection Features

The Sun GlassFish Message Queue software supports the following JMS connection features:

- "Connection Pooling" on page 285
- "Connection Failover" on page 285

Both these features use the `AddressList` configuration, which is populated with the hosts and ports of the JMS hosts defined in the Enterprise Server. The `AddressList` is updated whenever a JMS host configuration changes. The `AddressList` is inherited by any JMS resource when it is created and by any MDB when it is deployed.

Note – In the Sun GlassFish Message Queue software, the `AddressList` property is called `imqAddressList`.

Connection Pooling

The Enterprise Server pools JMS connections automatically.

To dynamically modify connection pool properties using the Administration Console, go to either the Connection Factories page (see "Creating JMS Resources: Destinations and Connection Factories" on page 284) or the Connector Connection Pools page.

To use the command line, use the `asadmin create-connector-connection-pool` command to manage the pool.

The `addresslist-behavior` JMS service attribute is set to `random` by default. This means that each `ManagedConnection` (physical connection) created from the `ManagedConnectionFactory` selects its primary broker in a random way from the `AddressList`.

The `addresslist-behavior` JMS service attribute can be set to `priority`. This means that the first broker in the `AddressList` is selected first.

When a JMS connection pool is created, there is one `ManagedConnectionFactory` instance associated with it. If you configure the `AddressList` as a `ManagedConnectionFactory` property, the `AddressList` configuration in the `ManagedConnectionFactory` takes precedence over the one defined in the Enterprise Server.

Connection Failover

To specify whether the Enterprise Server tries to reconnect to the primary broker if the connection is lost, set the `reconnect-enabled` attribute in the JMS service. To specify the number of retries and the time between retries, set the `reconnect-attempts` and `reconnect-interval-in-seconds` attributes, respectively.

If reconnection is enabled and the primary broker goes down, the Enterprise Server tries to reconnect to another broker in the `AddressList`. The `AddressList` is updated whenever a JMS host configuration changes. The logic for scanning is decided by two JMS service attributes, `addresslist-behavior` and `addresslist-iterations`.

You can override these settings using JMS connection factory settings. For details, see Chapter 19, "Administering the Java Message Service (JMS)," in *Sun GlassFish Enterprise Server v3 Administration Guide*.

The Sun GlassFish Message Queue software transparently transfers the load to another broker when the failover occurs. JMS semantics are maintained during failover.

Transactions and Non-Persistent Messages

During transaction recovery, non-persistent messages might be lost. If the broker fails between the transaction manager's prepare and commit operations, any non-persistent message in the transaction is lost and cannot be delivered. A message that is not saved to a persistent store is not available for transaction recovery.

Using the `ConfigurableTransactionSupport` Interface

The Java EE Connector 1.6 specification allows a resource adapter to use the `transaction-support` attribute to specify the level of transaction support that the resource adapter handles. However, the resource adapter vendor does not have a mechanism to figure out the current transactional context in which a `ManagedConnectionFactory` is used.

If a `ManagedConnectionFactory` implements an optional interface called `com.sun.appserv.connectors.spi.ConfigurableTransactionSupport`, the Enterprise Server notifies the `ManagedConnectionFactory` of the `transaction-support` configured for the connector connection pool when the `ManagedConnectionFactory` instance is created for the pool. Connections obtained from the pool can then be used with a transaction level at or lower than the configured value. For example, a connection obtained from a pool that is set to `XA_TRANSACTION` could be used as a LOCAL resource in a last-agent-optimized transaction or in a non-transactional context.

Authentication With `ConnectionFactory`

If your web, EJB, or client module has `res-auth` set to `Container`, but you use the `ConnectionFactory.createConnection("user","password")` method to get a connection, the Enterprise Server searches the container for authentication information before using the supplied user and password. Version 7 of the Enterprise Server threw an exception in this situation.

Message Queue `varhome` Directory

The Sun GlassFish Message Queue software uses a default directory for storing data such as persistent messages and its log file. This directory is called `varhome`. The Enterprise Server uses *domain-dir*`/imq` as the `varhome` directory if the type of relationship between the Enterprise Server and the Message Queue software is `LOCAL` or `EMBEDDED`. If the relationship type is `REMOTE`, the Message Queue software determines the `varhome` location. For more information about the types of relationships between the Enterprise Server and Message Queue, see "The JMS Provider" on page 280.

When executing Message Queue scripts such as *as-install*`/imq/bin/imqusermgr`, use the `-varhome` option to point the scripts to the Message Queue data if the relationship type is `LOCAL` or `EMBEDDED`. For example:

```
imqusermgr -varhome $AS_INSTALL/domains/domain1/imq add -u testuser
```

For more information about the Message Queue software, refer to the documentation at `http://docs.sun.com/coll/1343.9`.

Delivering SOAP Messages Using the JMS API

Web service clients use the Simple Object Access Protocol (SOAP) to communicate with web services. SOAP uses a combination of XML-based data structuring and Hyper Text Transfer Protocol (HTTP) to define a standardized way of invoking methods in objects distributed in diverse operating environments across the Internet.

For more information about SOAP, see the Apache SOAP web site at `http://xml.apache.org/soap/index.html`.

You can take advantage of the JMS provider's reliable messaging when delivering SOAP messages. You can convert a SOAP message into a JMS message, send the JMS message, then convert the JMS message back into a SOAP message. The following sections explain how to do these conversions:

- "To Send SOAP Messages Using the JMS API" on page 287
- "To Receive SOAP Messages Using the JMS API" on page 288

▼ To Send SOAP Messages Using the JMS API

1 **Import the `MessageTransformer` library.**

```
import com.sun.messaging.xml.MessageTransformer;
```

This is the utility whose methods you use to convert SOAP messages to JMS messages and the reverse. You can then send a JMS message containing a SOAP payload as if it were a normal JMS message.

2 **Initialize the `TopicConnectionFactory`, `TopicConnection`, `TopicSession`, and publisher.**

```
tcf = new TopicConnectionFactory();
tc = tcf.createTopicConnection();
session = tc.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);
topic = session.createTopic(topicName);
publisher = session.createPublisher(topic);
```

3 **Construct a SOAP message using the SOAP with Attachments API for Java (SAAJ).**

```
/*construct a default soap MessageFactory */
MessageFactory mf = MessageFactory.newInstance();
* Create a SOAP message object.*/
SOAPMessage soapMessage = mf.createMessage();
/** Get SOAP part.*/
SOAPPart soapPart = soapMessage.getSOAPPart();
/* Get SOAP envelope. */
SOAPEnvelope soapEnvelope = soapPart.getEnvelope();
/* Get SOAP body.*/
SOAPBody soapBody = soapEnvelope.getBody();
/* Create a name object. with name space */
/* http://www.sun.com/imq. */
Name name = soapEnvelope.createName("HelloWorld", "hw",
 "http://www.sun.com/imq");
* Add child element with the above name. */
SOAPElement element = soapBody.addChildElement(name)
/* Add another child element.*/
element.addTextNode( "Welcome to Sun GlassFish Web Services." );
/* Create an atachment with activation API.*/
URL url = new URL ("http://java.sun.com/webservices/");
DataHandler dh = new DataHandler (url);
AttachmentPart ap = soapMessage.createAttachmentPart(dh);
/*set content type/ID. */
ap.setContentType("text/html");
ap.setContentId("cid-001");
/**  add the attachment to the SOAP message.*/
soapMessage.addAttachmentPart(ap);
soapMessage.saveChanges();
```

4 **Convert the SOAP message to a JMS message by calling the `MessageTransformer.SOAPMessageintoJMSMessage()` method.**

```
Message m = MessageTransformer.SOAPMessageIntoJMSMessage (soapMessage,
session );
```

5 **Publish the JMS message.**

```
publisher.publish(m);
```

6 **Close the JMS connection.**

```
tc.close();
```

▼ To Receive SOAP Messages Using the JMS API

1 **Import the `MessageTransformer` library.**

```
import com.sun.messaging.xml.MessageTransformer;
```

This is the utility whose methods you use to convert SOAP messages to JMS messages and the reverse. The JMS message containing the SOAP payload is received as if it were a normal JMS message.

2 **Initialize the** `TopicConnectionFactory`, `TopicConnection`, `TopicSession`, `TopicSubscriber`, **and** `Topic`.

```
messageFactory = MessageFactory.newInstance();
tcf = new com.sun.messaging.TopicConnectionFactory();
tc = tcf.createTopicConnection();
session = tc.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
topic = session.createTopic(topicName);
subscriber = session.createSubscriber(topic);
subscriber.setMessageListener(this);
tc.start();
```

3 **Use the** `OnMessage` **method to receive the message. Use the** `SOAPMessageFromJMSMessage` **method to convert the JMS message to a SOAP message.**

```
public void onMessage (Message message) {
SOAPMessage soapMessage =
 MessageTransformer.SOAPMessageFromJMSMessage( message,
 messageFactory ); }
```

4 **Retrieve the content of the SOAP message.**

◆ ◆ ◆ CHAPTER 18

Using the JavaMail API

This chapter describes how to use the JavaMail API, which provides a set of abstract classes defining objects that comprise a mail system.

This chapter contains the following sections:

Note – JavaMail resources are supported only in the full Sun GlassFishEnterprise Server, not in the Web Profile.

Introducing JavaMail

The JavaMail API defines classes such as `Message`, `Store`, and `Transport`. The API can be extended and can be subclassed to provide new protocols and to add functionality when necessary. In addition, the API provides concrete subclasses of the abstract classes. These subclasses, including `MimeMessage` and `MimeBodyPart`, implement widely used Internet mail protocols and conform to the RFC822 and RFC2045 specifications. The JavaMail API includes support for the IMAP4, POP3, and SMTP protocols.

The JavaMail architectural components are as follows:

- The *abstract layer* declares classes, interfaces, and abstract methods intended to support mail handling functions that all mail systems support.
- The *internet implementation layer* implements part of the abstract layer using the RFC822 and MIME internet standards.

- JavaMail uses the *JavaBeans Activation Framework* (JAF) to encapsulate message data and to handle commands intended to interact with that data.

For more information, see Chapter 18, "Administering the JavaMail Service," in *Sun GlassFish Enterprise Server v3 Administration Guide* and the JavaMail specification at `http://java.sun.com/products/javamail/`. A useful JavaMail tutorial is located at `http://java.sun.com/developer/onlineTraining/JavaMail/`.

Creating a JavaMail Session

You can create a JavaMail session in the following ways:

- In the Administration Console, open the Resources component and select JavaMail Sessions. For details, click the Help button in the Administration Console.
- Use the `asadmin create-javamail-resource` command. For details, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

JavaMail Session Properties

You can set properties for a JavaMail `Session` object. Every property name must start with a `mail-` prefix. The Enterprise Server changes the dash (`-`) character to a period (`.`) in the name of the property and saves the property to the `MailConfiguration` and JavaMail `Session` objects. If the name of the property doesn't start with `mail-`, the property is ignored.

For example, if you want to define the property `mail.from` in a JavaMail `Session` object, first define the property as follows:

- Name – `mail-from`
- Value – `john.doe@sun.com`

Looking Up a JavaMail Session

The standard Java Naming and Directory Interface (JNDI) subcontext for JavaMail sessions is `java:comp/env/mail`.

Registering JavaMail sessions in the `mail` naming subcontext of a JNDI namespace, or in one of its child subcontexts, is standard. The JNDI namespace is hierarchical, like a file system's directory structure, so it is easy to find and nest references. A JavaMail session is bound to a logical JNDI name. The name identifies a subcontext, `mail`, of the root context, and a logical name. To change the JavaMail session, you can change its entry in the JNDI namespace without having to modify the application.

The resource lookup in the application code looks like this:

```
InitialContext ic = new InitialContext();
String snName = "java:comp/env/mail/MyMailSession";
Session session = (Session)ic.lookup(snName);
```

For more information about the JNDI API, see Chapter 16, "Using the Java Naming and Directory Interface."

Sending and Reading Messages Using JavaMail

The following sections describe how to send and read messages using the JavaMail API:

- "To Send a Message Using JavaMail" on page 293
- "To Read a Message Using JavaMail" on page 294

▼ To Send a Message Using JavaMail

1 Import the packages that you need.

```
import java.util.*;
import javax.activation.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.naming.*;
```

2 Look up the JavaMail session.

```
InitialContext ic = new InitialContext();
String snName = "java:comp/env/mail/MyMailSession";
Session session = (Session)ic.lookup(snName);
```

For more information, see "Looking Up a JavaMail Session" on page 292.

3 Override the JavaMail session properties if necessary.

For example:

```
Properties props = session.getProperties();
props.put("mail.from", "user2@mailserver.com");
```

4 Create a `MimeMessage`.

The `msgRecipient`, `msgSubject`, and `msgTxt` variables in the following example contain input from the user:

```
Message msg = new MimeMessage(session);
msg.setSubject(msgSubject);
msg.setSentDate(new Date());
msg.setFrom();
msg.setRecipients(Message.RecipientType.TO,
   InternetAddress.parse(msgRecipient, false));
msg.setText(msgTxt);
```

5 **Send the message.**

```
Transport.send(msg);
```

▼ To Read a Message Using JavaMail

1 **Import the packages that you need.**

```
import java.util.*;
import javax.activation.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.naming.*;
```

2 **Look up the JavaMail session.**

```
InitialContext ic = new InitialContext();
String snName = "java:comp/env/mail/MyMailSession";
Session session = (javax.mail.Session)ic.lookup(snName);
```

For more information, see "Looking Up a JavaMail Session" on page 292.

3 **Override the JavaMail session properties if necessary.**

For example:

```
Properties props = session.getProperties();
props.put("mail.from", "user2@mailserver.com");
```

4 **Get a `Store` object from the `Session`, then connect to the mail server using the Store object's `connect()` method.**

You must supply a mail server name, a mail user name, and a password.

```
Store store = session.getStore();
store.connect("MailServer", "MailUser", "secret");
```

5 **Get the INBOX folder.**

```
Folder folder = store.getFolder("INBOX");
```

6 **It is efficient to read the `Message` objects (which represent messages on the server) into an array.**

```
Message[] messages = folder.getMessages();
```

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T