

Eclipse GlassFish Application Development Guide, Release 8

Chapter 1. Eclipse GlassFish

Application Development Guide

Release 8

Contributed 2018 - 2026

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

Eclipse GlassFish Application Development Guide, Release 8

Copyright (c) 2025 Contributors to the Eclipse Foundation. All rights reserved.

Copyright © 2019,2026 Contributors to the Eclipse Foundation. Copyright © 2013, 2019 Oracle and/or its affiliates. All rights reserved.

This program and the accompanying materials are made available under the terms of the Eclipse Public License v. 2.0, which is available at <http://www.eclipse.org/legal/epl-2.0>.

SPDX-License-Identifier: EPL-2.0

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.



Chapter 2. Preface

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

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

Eclipse GlassFish 8 is developed through the GlassFish project open-source community at <https://github.com/eclipse-ee4j/glassfish>. The GlassFish project provides a structured process for developing the Eclipse GlassFish platform that makes the new features of the Jakarta EE platform available faster, while maintaining the most important feature of Jakarta EE: compatibility. It enables Java developers to access the Eclipse GlassFish source code and to contribute to the development of the Eclipse GlassFish.

The following topics are addressed here:

- [Eclipse GlassFish Documentation Set](#)
- [Related Documentation](#)
- [Typographic Conventions](#)
- [Symbol Conventions](#)
- [Default Paths and File Names](#)

Eclipse GlassFish Documentation Set

The Eclipse GlassFish documentation set is organized into the following groups.

Common Eclipse GlassFish Guides

These guides apply to all Eclipse GlassFish variants.

Book Title	Description
Release Notes	Provides late-breaking information about the software and the documentation and includes a comprehensive, table-based summary of the supported hardware, operating system, Java Development Kit (JDK), and database drivers.
Application Development Guide	Explains how to create and implement Java Platform, Enterprise Edition (Jakarta EE platform) applications that are intended to run on the Eclipse GlassFish. These applications follow the open Java standards model for Jakarta EE components and application programmer interfaces (APIs). This guide provides information about developer tools, security, and debugging.

Book Title	Description
Error Message Reference	Describes error messages that you might encounter when using Eclipse GlassFish.
Performance Tuning Guide	Explains how to optimize the performance of Eclipse GlassFish.
Reference Manual	Provides reference information in man page format for Eclipse GlassFish administration commands, utility commands, and related concepts.

Eclipse GlassFish Server Guides

These guides are specific to the Eclipse GlassFish Server installation.

Book Title	Description
Server Quick Start Guide	Explains how to get started with the Eclipse GlassFish Server.
Installation Guide	Explains how to install the software and its components.
Upgrade Guide	Explains how to upgrade to the latest version of Eclipse GlassFish. This guide also describes differences between adjacent product releases and configuration options that can result in incompatibility with the product specifications.
Deployment Planning Guide	Explains how to build a production deployment of Eclipse GlassFish that meets the requirements of your system and enterprise.
Administration Guide	Explains how to configure, monitor, and manage Eclipse GlassFish subsystems and components from the command line by using the <code>asadmin</code> utility. Instructions for performing these tasks from the Administration Console are provided in the Administration Console online help.
Server Security Guide	Provides instructions for configuring and administering Eclipse GlassFish Server security.
Application Deployment Guide	Explains how to assemble and deploy applications to the Eclipse GlassFish and provides information about deployment descriptors.
Add-On Component Development Guide	Explains how to use published interfaces of Eclipse GlassFish to develop add-on components for Eclipse GlassFish. This document explains how to perform only those tasks that ensure that the add-on component is suitable for Eclipse GlassFish.
High Availability Administration Guide	Explains how to configure Eclipse GlassFish to provide higher availability and scalability through failover and load balancing.
Server Troubleshooting Guide	Describes common problems that you might encounter when using Eclipse GlassFish Server and explains how to solve them.

Embedded Eclipse GlassFish Guides

These guides cover running Eclipse GlassFish as a self-contained executable JAR for cloud

deployments, containers, and embedding in applications.

Book Title	Description
Embedded Server Guide	Explains how to run applications with Embedded Eclipse GlassFish from the command line or as a library, suitable for cloud deployments, containers, and integration testing.

Eclipse Open MQ Documentation

Book Title	Description
Message Queue Release Notes	Describes new features, compatibility issues, and existing bugs for Open Message Queue.
Message Queue Technical Overview	Provides an introduction to the technology, concepts, architecture, capabilities, and features of the Message Queue messaging service.
Message Queue Administration Guide	Explains how to set up and manage a Message Queue messaging system.
Message Queue Developer's Guide for JMX Clients	Describes the application programming interface in Message Queue for programmatically configuring and monitoring Message Queue resources in conformance with the Java Management Extensions (JMX).
Message Queue Developer's Guide for Java Clients	Provides information about concepts and procedures for developing Java messaging applications (Java clients) that work with Eclipse GlassFish.
Message Queue Developer's Guide for C Clients	Provides programming and reference information for developers working with Message Queue who want to use the C language binding to the Message Queue messaging service to send, receive, and process Message Queue messages.

Related Documentation

The following tutorials explain how to develop Jakarta EE applications:

- [Your First Cup: An Introduction to the Jakarta EE Platform](#). For beginning Jakarta EE programmers, this short tutorial 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.
- [The Jakarta EE Tutorial](#). This comprehensive tutorial explains how to use Jakarta EE platform technologies and APIs to develop Jakarta EE applications.

Javadoc tool reference documentation for packages that are provided with Eclipse GlassFish is available as follows.

- The Jakarta EE specifications and API specification is located at <https://jakarta.ee/specifications/>.

- The API specification for Eclipse GlassFish 8, including Jakarta EE platform packages and nonplatform packages that are specific to the Eclipse GlassFish product, is located at <https://glassfish.org/docs/>.

For information about creating enterprise applications in the NetBeans Integrated Development Environment (IDE), see the [NetBeans Documentation, Training & Support page](#).

For information about the Derby database for use with the Eclipse GlassFish, see the [Derby page](#).

The Jakarta EE Samples project is a collection of sample applications that demonstrate a broad range of Jakarta EE technologies. The Jakarta EE Samples are bundled with the Jakarta EE Software Development Kit (SDK) and are also available from the repository (<https://github.com/eclipse-ee4j/glassfish-samples>).

Typographic Conventions

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

Typeface	Meaning	Example
AaBbCc123	The names of commands, files, and directories, and onscreen computer output	Edit your <code>.login</code> file. Use <code>ls a</code> to list all files. <code>machine_name% you have mail.</code>
AaBbCc123	What you type, contrasted with onscreen computer output	<code>machine_name% su</code> <code>Password:</code>
AaBbCc123	A placeholder to be replaced with a real name or value	The command to remove a file is <code>rm</code> 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.

Symbol	Description	Example	Meaning
[]	Contains optional arguments and command options.	<code>ls [-l]</code>	The <code>-l</code> option is not required.
{ }	Contains a set of choices for a required command option.	<code>-d {y n}</code>	The <code>-d</code> option requires that you use either the <code>y</code> argument or the <code>n</code> argument.

Symbol	Description	Example	Meaning
<code>\${ }</code>	Indicates a variable reference.	<code>\${com.sun.javaRoot}</code>	References the value of the <code>com.sun.javaRoot</code> 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.

Placeholder	Description	Default Value
as-install	Represents the base installation directory for Eclipse GlassFish. In configuration files, as-install is represented as follows: <code>\${com.sun.aas.installRoot}</code>	<ul style="list-style-type: none"> Installations on the Oracle Solaris operating system, Linux operating system, and Mac OS operating system: user's-home-directory/<code>glassfish8/glassfish</code> Installations on the Windows operating system: SystemDrive:\code>glassfish8\glassfish
as-install-parent	Represents the parent of the base installation directory for Eclipse GlassFish.	<ul style="list-style-type: none"> Installations on the Oracle Solaris operating system, Linux operating system, and Mac operating system: user's-home-directory/<code>glassfish8</code> Installations on the Windows operating system: SystemDrive:\code>glassfish8
domain-root-dir	Represents the directory in which a domain is created by default.	as-install/ <code>domains/</code>

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: <code>\${com.sun.aas.instanceRoot}</code>	domain-root-dir/domain-name
instance-dir	Represents the directory for a server instance.	domain-dir/instance-name

Part I

Chapter 3. Development Tasks and Tools

Chapter 4. Setting Up a Development Environment

This chapter gives guidelines for setting up an application development environment in the Eclipse GlassFish. Setting up an environment for creating, assembling, deploying, and debugging your code involves installing the mainstream version of the Eclipse GlassFish and making use of development tools. In addition, sample applications are available.

The following topics are addressed here:

- [Installing and Preparing the Server for Development](#)
- [High Availability Features](#)
- [Development Tools](#)
- [Sample Applications](#)

Installing and Preparing the Server for Development

For more information about Eclipse GlassFish installation, see the [Eclipse GlassFish Installation Guide](#).

The following components are included in the full installation.

- JDK
- Eclipse GlassFish core
 - Java Platform, Standard Edition (Java SE) 11 or newer
 - Jakarta EE 10 compliant application server
 - Administration Console
 - **asadmin** utility
 - Other development and deployment tools
 - Open Message Queue software
 - Apache [Derby database](#)
 - Load balancer plug-ins for web servers

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

After you have installed Eclipse GlassFish, 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](#).
- Set up debugging. For more information, see [Debugging Applications](#).

- Configure the Virtual Machine for the Java platform (JVM software). For more information, see "[Administering the Virtual Machine for the Java Platform](#)" in Eclipse GlassFish Administration Guide.

High Availability Features

High availability features such as load balancing and session failover are discussed in detail in the [Eclipse GlassFish High Availability Administration Guide](#). This book describes the following features in the following sections:

- For information about HTTP session persistence, see [Distributed Sessions and Persistence](#).
- For information about checkpointing of the stateful session bean state, see [Stateful Session Bean Failover](#).
- For information about failover and load balancing for Java clients, see [Developing Java Clients](#).
- For information about load balancing for message-driven beans, see [Load-Balanced Message Inflow](#).

Development Tools

The following general tools are provided with the Eclipse GlassFish:

- [The `asadmin` Command](#)
- [The Administration Console](#)

The following development tools are provided with the Eclipse GlassFish or downloadable from Oracle:

- [The Migration Tool](#)
- [The NetBeans IDE](#)

The following third-party tools might also be useful:

- [Debugging Tools](#)
- [Profiling Tools](#)

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 [Eclipse GlassFish 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 Eclipse GlassFish 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 Eclipse GlassFish is running. By default, the host is `localhost`. For example:

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

The Migration Tool

The Migration Tool converts and reassembles Jakarta 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>.

Code Editors

There is several advanced editors with an integration with GlassFish or generic Jakarta EE servers. Sometimes you have to install additional extensions, sometimes all you need is just some configuration.

- [IntelliJ IDEA](#)
- [Eclipse IDE](#)
- [Apache NetBeans](#)
- [Microsoft Visual Studio Code](#)

Debugging Tools

You can use several debugging tools with the Eclipse GlassFish. For more information, see [Debugging Applications](#).

Profiling Tools

You can use several profilers with the Eclipse GlassFish. For more information, see [Profiling Tools](#).

Sample Applications

The samples are available from <https://github.com/eclipse-ee4j/glassfish-samples>.

Most Eclipse GlassFish samples have the following directory structure:

- The `docs` directory contains instructions for how to use the sample.

- The `pom.xml` file defines Maven targets for the sample.
- The `src/` directory contains source code for the sample.

Chapter 5. Class Loaders

Understanding Eclipse GlassFish class loaders can help you determine where to place supporting JAR and resource files for your modules and applications.

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.

The following topics are addressed here:

- [The Class Loader Hierarchy](#)
- [Delegation](#)
- [Using the Java Optional Package Mechanism](#)
- [Class Loader Universes](#)
- [Application-Specific Class Loading](#)
- [Circumventing Class Loader Isolation](#)



The Web Profile of the Eclipse GlassFish supports the EJB 3.1 Lite specification, which allows enterprise beans within web applications, among other features. The full Eclipse GlassFish supports the entire EJB 3.1 specification. For details, see [JSR 318](#) (<http://jcp.org/en/jsr/detail?id=318>).

For information about class loader debugging, see [Class Loader Debugging](#).

The Class Loader Hierarchy

Class loaders in the Eclipse GlassFish runtime follow a delegation hierarchy that is illustrated in the following figure and fully described in [Table 2-1](#).

The following table describes the class loaders in the Eclipse GlassFish.

Table 2-1 Eclipse GlassFish 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, <code>domain-dir/lib/ext</code> . It is parent to the Public API class loader. See Using the Java Optional Package Mechanism .
Public API	The Public API class loader makes available all classes specifically exported by the Eclipse GlassFish runtime for use by deployed applications. This includes, but is not limited to, Jakarta EE APIs and other Oracle APIs. It is parent to the Common class loader.

Class Loader	Description
Common	The Common class loader loads JAR files in the as-install/ lib directory, followed by JAR files in the domain-dir/ lib directory. Using 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 .
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 Developing Lifecycle Listeners .
Applib	<p>The Applib class loader loads the library classes, specified during deployment, for a specific enabled module or Jakarta EE application; see Application-Specific Class Loading. One instance of this class loader is present in each class loader universe; see Class Loader Universes. It is parent to the Archive class loader.</p> <p>When multiple deployed applications use the same library, they share the same instance of the library. One library cannot reference classes from another library.</p>
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 Eclipse GlassFish. This class loader also loads any application-specific classes generated by the Eclipse GlassFish runtime, such as stub classes or servlets generated by JSP pages.

In previous Eclipse GlassFish versions, the JVM options provided **classpath-prefix** and **classpath-suffix** attributes that made it possible to add JAR files or directories either in front of, or after the application server's system **classpath**. These options are not present in Eclipse GlassFish 6.0.

The **classpath-prefix** was typically used to substitute another package for one of the Eclipse GlassFish packages, for example if a newer one was available. This same result can be achieved on a per-application basis with the **--libraries** option for the **deploy** subcommand. For more information, see the **deploy(1)** help page. The Java Optional Package Mechanism does what **classpath-suffix** used to do. For more information, see [Using the Java Optional Package Mechanism](#).

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 `glassfish-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 Eclipse GlassFish 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 `glassfish-web.xml`, see the [Eclipse GlassFish 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 Jakarta 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, or use the `asadmin add-library` command with the `--type ext` option, then restart the server. For more information about the `asadmin add-library` command, see the Eclipse GlassFish Reference Manual.

For more information, see [Optional Packages - An Overview](http://docs.oracle.com/javase/8/docs/technotes/guides/extensions/extensions.html) (<http://docs.oracle.com/javase/8/docs/technotes/guides/extensions/extensions.html>) and [Understanding Extension Class Loading](http://docs.oracle.com/javase/tutorial/ext/basics/load.html) (<http://docs.oracle.com/javase/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://docs.oracle.com/javase/8/docs/technotes/guides/standards/) (<http://docs.oracle.com/javase/8/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 Jakarta 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 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 [Eclipse GlassFish Reference Manual](#).
- Use the `asadmin add-library` command with the `--type app` option, then restart the server. For details, see the [Eclipse GlassFish Reference Manual](#).



None of these alternatives apply to application clients. For more information, see [Using Libraries with Application Clients](#).

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 can 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). The Eclipse GlassFish automatically synchronizes these libraries to all remote cluster instances when the cluster is restarted. However, libraries specified by absolute paths are not guaranteed to be synchronized.



You can use application-specific class loading to specify a different XML parser

than the default Eclipse GlassFish XML parser.

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](#).

One library cannot reference classes from another library.

For general information about deployment, including dynamic reloading, see the [Eclipse GlassFish Application Deployment Guide](#).



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](#).

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](#)
- [Sharing Libraries Across a Cluster](#)
- [Packaging the Client JAR for One Application in Another Application](#)

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 use the `asadmin add-library` command with the `--type common` option, then restart the server. For more information about the `asadmin add-library` command, see the Eclipse GlassFish Reference Manual.

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](#).

For example, using the Common class loader is the recommended way of adding JDBC drivers to the Eclipse GlassFish. For a list of the JDBC drivers currently supported by the Eclipse GlassFish, see the [Eclipse GlassFish Release Notes](#). For configurations of supported and other drivers, see

"[Configuration Specifics for JDBC Drivers](#)" in Eclipse GlassFish Administration Guide.

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

Sharing Libraries Across a Cluster

To share libraries across a specific cluster, copy the JAR files to the domain-dir/**config**/cluster-config-name/**lib** directory.

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](#). 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 Jakarta 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 **glassfish-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 **glassfish-ejb-jar.xml** or **glassfish-**

`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 6. Debugging Applications

This chapter gives guidelines for debugging applications in the Eclipse GlassFish.

The following topics are addressed here:

- [Enabling Debugging](#)
- [JPDA Options](#)
- [Generating a Stack Trace for Debugging](#)
- [Application Client Debugging](#)
- [Open Message Queue Debugging](#)
- [Enabling Verbose Mode](#)
- [Class Loader Debugging](#)
- [Eclipse GlassFish Logging](#)
- [Profiling Tools](#)

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

- Java Platform Debugger Architecture - The Java Debugger: <https://docs.oracle.com/en/java/javase/21/docs/specs/jpda/architecture.html> <https://docs.oracle.com/en/java/javase/21/docs/specs/jpda/jpda.html>
- Java Platform Debugger Architecture - Connecting with JDB: <https://docs.oracle.com/en/java/javase/21/docs/specs/man/jdb.html>

Eclipse GlassFish debugging is based on the JPDA. For more information, see [JPDA Options](#).

You can attach to the Eclipse GlassFish using any JPDA compliant debugger.

You can enable debugging even when the Eclipse GlassFish is started without the `--debug` option. This is useful if you start the Eclipse GlassFish 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 JVM Settings component under the relevant configuration.
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](#).

See Also

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

JPDA Options

The default JPDA options in Eclipse GlassFish are as follows:

```
-Xdebug -agentlib: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/javase/technologies/core/toolsapis/jpda/>.

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 [Eclipse GlassFish 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=-agentlib: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](#).

You can also include JVM options in the `appclient` script directly. For information about the `appclient` script, see the [Eclipse GlassFish Reference Manual](#).



The Application Client Container is supported only in the full Eclipse GlassFish, not in the Web Profile. See [Developing Java Clients](#).

Open Message Queue Debugging

Open 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 [Open Message Queue Administration Guide](#).



JMS resources are supported only in the full Eclipse GlassFish, not in the Web Profile. See [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://docs.oracle.com/javase/8/docs/technotes/tools/share/jstack.html>) or the command `kill -QUIT process_id`.

Class Loader Debugging

To generate class loading messages, use the following `asadmin create-jvm-options` command:


```
asadmin create-jvm-options -verbose\:class
```

To send the JVM messages to a special JVM log file instead of `stdout`, use the following `asadmin create-jvm-options` commands:

```
asadmin create-jvm-options -XX\:+LogVMOutput  
asadmin create-jvm-options -XX\:LogFile=${com.sun.aas.instanceRoot}/logs/jvm.log
```



These `-XX` options are specific to the OpenJDK (or Hotspot) JVM and do not work with the JRockit JVM.

To send the Eclipse GlassFish messages to the Administration Console instead of `stderr`, start the domain in verbose mode as described in [Enabling Verbose Mode](#).

Eclipse GlassFish Logging

You can use the Eclipse GlassFish's log files to help debug your applications. Use the Administration Console. Select the Stand-Alone Instances component, select the instance from the table, then click the View Log Files button in the General Information page. Or select the Cluster component, select the cluster from the table, select the Instances tab, select the instance from the table, then click the View Log Files button in the General Information page.

To change logging settings, select Logger Settings under the relevant configuration.

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 Eclipse GlassFish to discover bottlenecks in server-side performance. This section describes how to configure profilers for use with Eclipse GlassFish.

The following topics are addressed here:

- [The NetBeans Profiler](#)
- [The HPROF Profiler](#)
- [The JProbe Profiler](#)

Information about comprehensive monitoring and management support in the Java 2 Platform, Standard Edition (J2SE platform) is available at <http://docs.oracle.com/javase/8/docs/technotes/guides/management/index.html>.

The NetBeans Profiler

For information on how to use the NetBeans profiler, see <http://profiler.netbeans.org/index.html>.

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 JVM Settings component under the relevant configuration. 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



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 Eclipse GlassFish.

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 Eclipse GlassFish using the Administration Console:
 1. Select the JVM Settings component under the relevant configuration.
 2. Then select the Profiler tab.
 3. 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
```



If any of the configuration options are missing or incorrect, the profiler might experience problems that affect the performance of the Eclipse GlassFish.

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 instance.
 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:
 1. Select Server Side for the type of application.
 2. 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 as an input to **JPROBE_ARGS_0**.

Part II

Chapter 7. Developing Applications and Application Components

Chapter 8. API for development

Eclipse GlassFish provides several APIs to build applications and components:

- [Jakarta EE API](#) - Platform or Web profile, depends on the Eclipse GlassFish distribution
- [MicroProfile API](#) - selected specifications, only available in Eclipse GlassFish Full
- GlassFish API - to access other functionality provided by Eclipse GlassFish

GlassFish API is composed of a few sets of APIs:

- GlassFish API
- GlassFish EE API
- Simple GlassFish API

GlassFish API

Most of the functionality specific to Eclipse GlassFish is available in the GlassFish API. To compile applications or components, add the `glassfish-api.jar` to the compile classpath. You may also need `scattered-archive-api.jar`.

The `glassfish-api.jar` is located in the Eclipse GlassFish installation in `as-install/modules/glassfish-api.jar`.

In Maven project, you can add it as the following dependency:

```
<dependency>
  <groupId>org.glassfish.main.common</groupId>
  <artifactId>glassfish-api</artifactId>
</dependency>
```

This will already add `scattered-archive-api.jar` as a transitive dependency.

MicroProfile JWT Integration

The GlassFish API includes CDI qualifiers for integrating with built-in security mechanisms.

@MicroProfileJwtAuthenticationMechanism Qualifier

The `@MicroProfileJwtAuthenticationMechanism` qualifier allows direct injection of the GlassFish built-in MicroProfile JWT authentication mechanism:

```
import jakarta.inject.Inject;
import
jakarta.security.enterprise.authentication.mechanism.http.HttpAuthenticationMechanism;
import org.glassfish.api.security.jwt.MicroProfileJwtAuthenticationMechanism;
```

```

@ApplicationScoped
public class SecurityService {

    @Inject
    @MicroProfileJwtAuthenticationMechanism
    private HttpAuthenticationMechanism jwtAuthMechanism;

    // Use the JWT authentication mechanism directly
}

```

This qualifier provides:

- **Direct Access:** Direct access to the GlassFish JWT authentication mechanism
- **CDI Integration:** Seamless integration with CDI dependency injection
- **Type Safety:** Uses the standard `HttpAuthenticationMechanism` interface

The `@MicroProfileJwtAuthenticationMechanism` qualifier is complementary to the qualifiers for the standard Jakarta Security mechanisms like the Basic authentication mechanism. It can be combined with the other standard mechanisms in the same `HttpAuthenticationMechanismHandler`.

Example:

```

@ApplicationScoped
@Alternative
@Priority(Interceptor.Priority.APPLICATION)
public class CustomAuthenticationHandler implements HttpAuthenticationMechanismHandler
{

    @Inject
    @MicroProfileJwtAuthenticationMechanism
    HttpAuthenticationMechanism jwtAuthentication;

    @Inject
    @BasicAuthenticationMechanismDefinition.BasicAuthenticationMechanism
    HttpAuthenticationMechanism basicAuthentication;

    @Override
    public AuthenticationStatus validateRequest(HttpServletRequest request,
        HttpServletResponse response, HttpContext messageContext) throws
        AuthenticationException {
        // delegate to one of the two mechanisms
    }

}

```


GlassFish EE API

GlassFish EE API provides functionality related to Jakarta EE. To compile applications or components, add the `glassfish-ee-api.jar` to the compile classpath.

The `glassfish-ee-api.jar` is located in the Eclipse GlassFish installation in `as-install/modules/glassfish-ee-api.jar`.

In Maven project, you can add it as the following dependency:

```
<dependency>
  <groupId>org.glassfish.main.common</groupId>
  <artifactId>glassfish-ee-api</artifactId>
</dependency>
```

Simple GlassFish API

Simple GlassFish API provides basic functionality to deploy applications and run admin commands. Mostly to use embedded Eclipse GlassFish programmatically. To compile applications or components, add the `simple-glassfish-api.jar` to the compile classpath.

The `simple-glassfish-api.jar` is located in the Eclipse GlassFish installation in `as-install/modules/simple-glassfish-api.jar`.

In Maven project, you can add it as the following dependency:

```
<dependency>
  <groupId>org.glassfish.main.common</groupId>
  <artifactId>simple-glassfish-api</artifactId>
</dependency>
```

Chapter 9. Securing Applications

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

For information about administrative security for the Eclipse GlassFish, see the [Eclipse GlassFish Server Security Guide](#).

For general information about Jakarta EE security, see [Security](#) in The Jakarta EE Tutorial.

The following topics are addressed here:

- [Security Goals](#)
- [Eclipse GlassFish Specific Security Features](#)
- [Container Security](#)
- [Roles, Principals, and Principal to Role Mapping](#)
- [Realm Configuration](#)
- [Jakarta EE Security API Support](#)
- [JACC Support](#)
- [Pluggable Audit Module Support](#)
- [The `server.policy` File](#)
- [Configuring Message Security for Web Services](#)
- [Programmatic Login Using the ProgrammaticLogin Class](#)
- [User Authentication for Single Sign-on](#)
- [Adding Authentication Mechanisms to the Servlet Container](#)



The Web Profile of the Eclipse GlassFish supports the EJB 3.1 Lite specification, which allows enterprise beans within web applications, among other features. The full Eclipse GlassFish 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 Eclipse GlassFish is to provide highly secure, interoperable, and distributed component computing based on the Jakarta EE security model. Security goals include:

- Full compliance with the Jakarta EE security model. This includes EJB and servlet role-based authorization.
- Support for single sign-on across all Eclipse GlassFish 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 Eclipse GlassFish specific XML-based role mapping.
- Support for Java Authorization Contract for Containers (JACC) pluggable authorization as included in the Jakarta EE specification and defined by [Java Specification Request \(JSR\) 115](http://www.jcp.org/en/jsr/detail?id=115) (<http://www.jcp.org/en/jsr/detail?id=115>).
- Support for Java Authentication Service Provider Interface for Containers as included in the Jakarta EE specification and defined by [JSR 196](http://www.jcp.org/en/jsr/detail?id=196) (<http://www.jcp.org/en/jsr/detail?id=196>).
- Support for Web Services Interoperability Technologies (WSIT) as described in [Jakarta EE Tutorial](#).
- Support for the Jakarta EE Security API as included in the Jakarta EE specification and defined by [JSR 375](https://jcp.org/en/jsr/detail?id=375) (<https://jcp.org/en/jsr/detail?id=375>)

Eclipse GlassFish Specific Security Features

The Eclipse GlassFish supports the Jakarta EE security model, as well as the following features which are specific to the Eclipse GlassFish:

- Message security; see [Configuring Message Security for Web Services](#)
- Single sign-on across all Eclipse GlassFish applications within a single security domain; see [User Authentication for Single Sign-on](#)
- Programmatic login; see [Programmatic Login Using the ProgrammaticLogin Class](#)

Container Security

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

- [Declarative Security](#)
- [Programmatic Security](#)

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 Jakarta EE application's security

structure, including security roles, access control, and authentication requirements.

The Eclipse GlassFish supports the deployment descriptors specified by Jakarta EE and has additional security elements included in its own deployment descriptors. Declarative security is the application deployer's responsibility. For more information about Eclipse GlassFish deployment descriptors, see the [Eclipse GlassFish Application Deployment Guide](#).

There are two levels of declarative security, as follows:

- [Application Level Security](#)
- [Component Level Security](#)

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 `glassfish-ejb-jar.xml` files) and to the servlet XML deployment descriptors (`web.xml` and `glassfish-web.xml` files). For an individually deployed web or EJB module, you define roles using `@DeclareRoles` annotations or `role-name` elements in the Jakarta 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 `glassfish-application.xml`, `glassfish-ejb-jar.xml`, or `glassfish-web.xml` file for each `role-name` used by the application. By default, group principal names are mapped to roles of the same name. Accordingly, the Default Principal To Role Mapping setting is enabled by default on the Security page of the Eclipse GlassFish Administration Console. This default role mapping definition is in effect if you do not define your own mapping in the deployment descriptor for your application as described in this section. For more information, see [Roles, Principals, and Principal to Role Mapping](#).

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 `glassfish-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 `glassfish-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 Jakarta 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 API for programmatic security consists of methods of the Jakarta EE Security API `SecurityContext` interface, and methods of the EJB `EJBContext` interface and the servlet

`HttpServletRequest` interface. The Eclipse GlassFish supports these interfaces as specified in the Java EE specification.

There is also a proprietary Glassfish API for programmatic login. See [Programmatic Login Using the ProgrammaticLogin Class](#).

For more information about programmatic security, see [Using Programmatic Security](#) in the The Jakarta EE Tutorial.

Roles, Principals, and Principal to Role Mapping

By default, any groups that an authenticated user belongs to will be mapped to roles with the same names. Therefore, the Default Principal To Role Mapping setting is enabled by default on the Security page of the GlassFish Administration Console. To change the default mapping you can clear this setting. For applications, you define roles in `@DeclareRoles` annotations or the Jakarta EE deployment descriptor file `application.xml`. You define the corresponding role mappings in the Eclipse GlassFish deployment descriptor file `glassfish-application.xml`. For individually deployed web or EJB modules, you define roles in `@DeclareRoles` annotations or the Jakarta EE deployment descriptor files `web.xml` or `ejb-jar.xml`. You define the corresponding role mappings in the Eclipse GlassFish deployment descriptor files `glassfish-web.xml` or `glassfish-ejb-jar.xml`.

For more information regarding Jakarta EE deployment descriptors, see the Jakarta EE Specification. For more information regarding Eclipse GlassFish deployment descriptors, see "[Elements of the Eclipse GlassFish Deployment Descriptors](#)" in Eclipse GlassFish Application Deployment Guide.

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

```
<glassfish-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>
</glassfish-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 Eclipse GlassFish 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 Eclipse GlassFish instance. 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 [Eclipse GlassFish 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
```

Default principal to role mapping is enabled by default. To disable it, set the default principal to role mapping property to false.

Realm Configuration

The following topics are addressed here:

- [Supported Realms](#)
- [How to Configure a Realm](#)
- [How to Set a Realm for an Application or Module](#)

- [Creating a Custom Realm](#)

Supported Realms

The following realms are supported in the current release of the Eclipse GlassFish:

- **file** - Stores user information in a file. This is the default realm when you first install the Eclipse GlassFish.
- **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 Eclipse GlassFish uses the database information and the enabled JDBC realm option in the configuration file. For digest authentication, a JDBC realm should be created with **digestRealm** as the JAAS context.

- **certificate** - Sets up the user identity in the Eclipse GlassFish 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](#).

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 [Eclipse GlassFish 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:

- **glassfish-application** element in **glassfish-application.xml**
- **web-app** element in **web.xml**
- **as-context** element in **glassfish-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 **glassfish-application.xml** is used, otherwise the domain's default realm is used.

For example, a realm is specified in **glassfish-application.xml** as follows:

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

For more information about the deployment descriptor files and elements, see "[Elements of the Eclipse GlassFish Deployment Descriptors](#)" in Eclipse GlassFish 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 Eclipse GlassFish.

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 Eclipse GlassFish, see [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 Jakarta EE security mechanisms. For more information about JAAS, refer to the specification, available at <https://jakarta.ee/specifications/authentication/> and <https://jakarta.ee/specifications/authorization/>.

For general information about realms and login modules, see the section about working with realms, users, groups, and roles in [Introduction to Security in the Jakarta EE Platform](#) in The Jakarta EE Tutorial.

For Javadoc tool pages relevant to custom realms, see 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 Eclipse GlassFish 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(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 Eclipse GlassFish 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.



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.

Jakarta EE Security API Support

JSR-375 defines several authentication-related plugin SPIs, such as, `HttpAuthenticationMechanism` interface, the `IdentityStore` and `IdentityStoreHandler` interfaces:

- `HttpAuthenticationMechanism`: An interface for modules that authenticate callers to a web application. An application can supply its own `HttpAuthenticationMechanism`, or use one of the default implementations provided by the container.
- `IdentityStore`: This interface defines methods for validating a caller's credentials (such as user name and password) and returning group membership information. An application can provide its own `IdentityStore`, or use the built in LDAP or Database store.
- `RememberMeIdentityStore`: This interface is a variation on the `IdentityStore` interface, intended to address cases where an authenticated user's identity should be remembered for an extended period of time, so that the caller can return to the application periodically without needing to present primary authentication credentials each time.

In addition to these authentication plugin SPIs, the Jakarta EE Security API specification defines the `SecurityContext` API for use by application code to query and interact with the current security context. The `SecurityContext` interface defines methods that allow an application to access security information about a caller, authenticate a caller, and authorize a caller. These methods include `getCallerPrincipal()`, `getPrincipalsByType()`, `isCallerInRole()`, `authenticate()`, and `hasAccessToWebResource()`.

JACC Support

JACC (Java Authorization Contract for Containers) is part of the Jakarta 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 Jakarta 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 Eclipse GlassFish 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.

The following topics are addressed here:

- [Configuring an Audit Module](#)
- [The AuditModule Class](#)

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 [Eclipse GlassFish 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, see 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 Eclipse GlassFish 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 Eclipse GlassFish domain has its own global J2SE policy file, located in `domain-dir/config`. The file is named `server.policy`.

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

The following topics are addressed here:

- [Default Permissions](#)
- [System Properties](#)
- [Changing Permissions for an Application](#)
- [Enabling and Disabling the Security Manager](#)

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 Eclipse GlassFish 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. Jakarta 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 [Eclipse GlassFish Reference Manual](#).

Table 4-1 Predefined System Properties

Property	Default	Description
<code>com.sun.aas.installRoot</code>	depends on operating system	Specifies the directory where the Eclipse GlassFish is installed.
<code>com.sun.aas.instanceRoot</code>	depends on operating system	Specifies the top level directory for a server instance.
<code>com.sun.aas.hostName</code>	none	Specifies the name of the host (machine).
<code>com.sun.aas.javaRoot</code>	depends on operating system	Specifies the installation directory for the Java runtime.
<code>com.sun.aas.imqLib</code>	depends on operating system	Specifies the library directory for the Open Message Queue software.
<code>com.sun.aas.configName</code>	<code>server-config</code>	Specifies the name of the configuration used by a server instance.
<code>com.sun.aas.instanceName</code>	<code>server1</code>	Specifies the name of the server instance. This property is not used in the default configuration, but can be used to customize configuration.
<code>com.sun.aas.clusterName</code>	<code>cluster1</code>	Specifies the name of the cluster. This property is only set on clustered server instances. This property is not used in the default configuration, but can be used to customize configuration.
<code>com.sun.aas.domainName</code>	<code>domain1</code>	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 Jakarta 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/-" {  
    ...  
}
```



Deployment directories may change between Eclipse GlassFish 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 Eclipse GlassFish 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";
}
```



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 Jakarta 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 [Eclipse GlassFish Reference Manual](#).

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

For detailed information about policy file syntax, see <http://docs.oracle.com/javase/8/docs/technotes/guides/security/PolicyFiles.html>.

For information about using system properties in the `server.policy` file, see <http://docs.oracle.com/javase/8/docs/technotes/guides/security/PolicyFiles.html>.

For detailed information about the permissions you can set in the `server.policy` file, see <http://docs.oracle.com/javase/8/docs/technotes/guides/security/permissions.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 enable the security manager, do one of the following:

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

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

To disable the security manager, uncheck the Security Manager Enabled box or use the corresponding `asadmin delete-jvm-options` command. For more information about `create-jvm-options` and `delete-jvm-options`, see the [Eclipse GlassFish Reference Manual](#).

If the security manager is enabled and you are using the Java Persistence API by calling `Persistence.createEMF()`, the EclipseLink persistence provider requires that you set the `eclipselink.security.usedoprivileged` JVM option to `true` as follows:

```
asadmin create-jvm-options -Declipselink.security.usedoprivileged=true
```

If the security manager is enabled and you are using the Java Persistence API by injecting or looking up an entity manager or entity manager factory, the EJB container sets this JVM option for you.

You must grant additional permissions to CDI-enabled Jakarta EE applications that are deployed in a Eclipse GlassFish 8 domain or cluster for which security manager is enabled. These additional permissions are not required when security manager is disabled.

To deploy CDI-enabled Jakarta EE applications in a Eclipse GlassFish 8 domain or cluster for which

security manager is enabled, add the following permissions to the applications:

```
grant codeBase "file:${com.sun.aas.instanceRoot}/applications/[ApplicationName]" {  
    permission java.lang.reflect.ReflectPermission "suppressAccessChecks";  
};
```

For example, for a CDI application named `foo.war`, add the following permissions to the `server.policy` file, restart the domain or cluster, and then deploy and use the application.

```
grant codeBase "file:${com.sun.aas.instanceRoot}/applications/foo" {  
    permission java.lang.reflect.ReflectPermission "suppressAccessChecks";  
};
```

For more information about modifying application permissions, see [Changing Permissions for an Application](#).

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.



Message security (JSR 196) is supported only in the full Eclipse GlassFish, not in the Web Profile.



In this release of the Eclipse GlassFish, message layer annotations are not supported.

For more information about web services, see [Developing Web Services](#).

For more information about message security, see the following:

- ["Introduction to Security in the Jakarta EE Platform"](#) in The Jakarta EE Tutorial
- [Eclipse GlassFish Server Security 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?f=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 <http://xwss.java.net/>
- The WSIT page at <http://wsit.java.net/>

The following topics are addressed here:

- [Message Security Providers](#)
- [Message Security Responsibilities](#)
- [Application-Specific Message Protection](#)
- [Understanding and Running the Sample Application](#)

Message Security Providers

When you first install the Eclipse GlassFish, 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 [Eclipse GlassFish Reference Manual](#).

The example described in [Understanding and Running the Sample Application](#) uses the `ClientProvider` and `ServerProvider` providers, which are enabled when the Ant targets are run. You don't need to enable these on the Eclipse GlassFish prior to running the example.

If you install the OpenSSO, 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 `glassfish-web.xml`, `glassfish-ejb-jar.xml`, and `glassfish-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 OpenSSO for single sign-on. To use this provider, specify it in the **httpServlet-security-provider** attribute of the **glassfish-web-app** element in the **glassfish-web.xml** file.

Liberty specifications can be viewed at <http://www.projectliberty.org/resources/specifications.php?f=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 Eclipse GlassFish deployment descriptor files, see the [Eclipse GlassFish Application Deployment Guide](#).

For information about configuring these providers in the Eclipse GlassFish, see the [Eclipse GlassFish Server Security Guide](#). For additional information about overriding provider settings, see [Application-Specific Message Protection](#).

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 [Eclipse GlassFish 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 4-2 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 .

Property	Default	Description
<code>debug</code>	<code>false</code>	If <code>true</code> , enables dumping of server provider debug messages to the server log.
<code>dynamic.username.password</code>	<code>false</code>	If <code>true</code> , signals the provider runtime to collect the user name and password from the <code>CallbackHandler</code> for each request. If <code>false</code> , the user name and password for <code>wsse:UsernameToken(s)</code> is collected once, during module initialization. This property is only applicable for a <code>ClientAuthModule</code> .
<code>encryption.key.alias</code>	<code>s1as</code>	Specifies the encryption key used by the provider. The key is identified by its <code>keystore</code> alias.
<code>signature.key.alias</code>	<code>s1as</code>	Specifies the signature key used by the provider. The key is identified by its <code>keystore</code> alias.

Message Security Responsibilities

In the Eclipse GlassFish, 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 following topics are addressed here:

- [Application Developer Responsibilities](#)
- [Application Deployer Responsibilities](#)
- [System Administrator Responsibilities](#)

Application Developer Responsibilities

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 Eclipse GlassFish level. If so, ensuring that this need is communicated to the system administrator, or taking care of implementing message security at the Eclipse GlassFish level.

Application Deployer Responsibilities

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 Eclipse GlassFish 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](#). A sample application using message security is discussed in [Understanding and Running the Sample Application](#).

System Administrator Responsibilities

The system administrator is responsible for the following:

- Configuring message security providers on the Eclipse GlassFish.
- 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`. If Network Security Services (NSS) is installed, certificates and private keys are stored in an NSS database, where they are managed using `certutil`. System administrator tasks are discussed in the [Eclipse GlassFish Server Security Guide](#).

Application-Specific Message Protection

When the Eclipse GlassFish 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 Eclipse GlassFish 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 <http://wsit.java.net/>.

For more information about message security providers, see [Message Security Providers](#).

For more details on message security binding for EJB web services, servlet web services, and clients, see the XML file descriptions in "[Elements of the Eclipse GlassFish Deployment Descriptors](#)" in Eclipse GlassFish Application Deployment Guide.

- For `glassfish-ejb-jar.xml`, see "[The glassfish-ejb-jar.xml File](#)" in Eclipse GlassFish Application Deployment Guide.
- For `glassfish-web.xml`, see "[The glassfish-web.xml File](#)" in Eclipse GlassFish Application Deployment Guide.
- For `glassfish-application-client.xml`, see "[The glassfish-application-client.xml file](#)" in Eclipse GlassFish Application Deployment Guide.

The following topics are addressed here:

- [Using a Signature to Enable Message Protection for All Methods](#)
- [Configuring Message Protection for a Specific Method Based on Digital Signatures](#)

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 `glassfish-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 the [Eclipse GlassFish Server Security 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](#) discusses listing specific methods or using wildcard characters.

This section uses the sample application discussed in [Understanding and Running the Sample Application](#) 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

Follow this procedure.

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

For the `xms` example, this file is located in the directory `app-dir\src\conf`, where `app-dir` is defined in [To Set Up the Sample Application](#).

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

```
<glassfish-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>
</glassfish-ejb-jar>

```

3. Compile, deploy, and run the application as described in [To Run the Sample Application](#).

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](#), to enable message protection for a specific method, update the `message-security-binding` element for the EJB web service endpoint in the application's `glassfish-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 the [Eclipse GlassFish Server Security 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](#) 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

Follow this procedure.

1. In a text editor, open the application's `glassfish-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](#).

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

```

<glassfish-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>
</glassfish-ejb-jar>

```

3. Compile, deploy, and run the application as described in [To Run the Sample Application](#).

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](#).

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 Jakarta 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](#).

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 Eclipse GlassFish level. For information about using message security at the application level, see [Application-Specific Message Protection](#). 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>.

The following topics are addressed here:

- [To Set Up the Sample Application](#)

- [To Run the Sample Application](#)

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](https://www.oracle.com/java/technologies/java-archive-eesdk-downloads.html) (<https://www.oracle.com/java/technologies/java-archive-eesdk-downloads.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 Eclipse GlassFish is running.
Message security providers are set up when the Ant targets are run, so you do not need to configure these on the Eclipse GlassFish 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`](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:

1. To compile samples: `ant`
2. To deploy samples: `ant deploy`
3. 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 Using the ProgrammaticLogin Class

Programmatic login allows a deployed Jakarta 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 Jakarta 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 Jakarta EE standard authentication mechanisms.

This section describes a proprietary GlassFish mechanism, but see also the standard security APIs in the Jakarta EE tutorial.



The `com.sun.appserv.security.ProgrammaticLogin` class in Eclipse GlassFish is not a Jakarta EE API; therefore, it is not portable to other application servers.

The following topics are addressed here:

- [Programmatic Login Precautions](#)
- [Granting Programmatic Login Permission](#)
- [The ProgrammaticLogin Class](#)

Programmatic Login Precautions

The Eclipse GlassFish 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 Eclipse GlassFish-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](#). This permission is not granted by default to deployed applications because this is not a standard Jakarta 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, see 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 Eclipse GlassFish 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 "[Administering Internet Connectivity](#)" in Eclipse GlassFish 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 [Eclipse GlassFish 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 Eclipse GlassFish 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.

The following topics are addressed here:

- [The Eclipse GlassFish and JSR-375](#)
- [The Eclipse GlassFish and JSR 196](#)
- [Writing a Server Authentication Module](#)
- [Sample Server Authentication Module](#)
- [Compiling and Installing a Server Authentication Module](#)
- [Configuring a Server Authentication Module](#)
- [Binding a Server Authentication Module to Your Application](#)

The Eclipse GlassFish and JSR-375

The Eclipse GlassFish implements JSR-375 to provide built-in support for BASIC, FORM and Custom FORM authentication mechanisms. JSR-375 also defines plug-in interfaces for authentication and identity stores, that is, the `HttpAuthenticationMechanism` interface and the `IdentityStore` interface, respectively. Though `HttpAuthenticationMechanism` implementations can authenticate users in any

manner they choose, the `IdentityStore` interface provides a convenient mechanism. A significant advantage of using `HttpAuthenticationMechanism` and `IdentityStore` over the declarative mechanisms defined by the Servlet specification is that it allows an application to control the identity stores that it authenticates against, in a standard, portable way. You can use the built-in implementations of these APIs, or define custom implementations.

Jakarta EE Security API defines several annotations, with names that end with `Definition`, which when used makes the corresponding built-in mechanism available as a CDI bean. Jakarta EE Security API also supports the use of Expression Language 3.0 in these annotations to allow dynamic configuration.

Built-in Authentication Mechanisms

An application packages its own `HttpAuthenticationMechanism` by including in a bean archive that is a part of the application. Alternatively, it may select and configure one of the container's built-in mechanisms using the corresponding annotation, as listed below:

- `BasicAuthenticationMechanismDefinition`—implements BASIC authentication that conforms to the behavior of the servlet container when BASIC `<auth-method>` is declared in `web.xml`.
- `CustomFormAuthenticationMechanismDefinition`—implements FORM authentication that conforms to the behavior of the servlet container when the FORM `<auth-method>` is declared in `web.xml`.
- `FormAuthenticationMechanismDefinition`—implements a modified version of FORM authentication in which custom handling replaces the POST to `j_security_check`.

In Eclipse GlassFish, all built-in authentication mechanisms need to be authenticated using an identity store. The `IdentityStore` interface, included in the Jakarta EE Security API, defines an SPI for interacting with identity stores, which are directories or databases containing user account information. The `IdentityStore` interface has four methods: `validate(Credential)`, `getCallerGroups(CredentialValidationResult)`, `validationTypes()` and `priority()`. Developers can provide their own implementation of this interface, or use one of the built-in Identity Stores. The `RememberMeIdentityStore` interface, which is a variation on the `IdentityStore` interface, can be used when an application wants to "remember" a user's authenticated session for an extended period, so that the caller can return to the application periodically without needing to present primary authentication credentials each time.

There are two built-in implementations of `IdentityStore`: an LDAP identity store, and a Database identity store. The following snippet shows the usage of `DatabaseIdentityStoreDefinition`, which makes `DatabaseIdentityStore` available as CDI bean.

```
@DatabaseIdentityStoreDefinition(  
    callerQuery = "#{select password from caller where name = '?'}",  
    groupsQuery = "select group_name from caller_groups where caller_name = ?",  
    hashAlgorithm = Pbkdf2PasswordHash.class,  
    priorityExpression = "#{100}",  
    hashAlgorithmParameters = {  
        "Pbkdf2PasswordHash.Iterations=3072",  
        "${applicationConfig.dyna}"  
    }  
)
```


)

Since Jakarta EE Security API provides support for Expression Language 3.0, regular expressions can be used to set value of annotation attributes.

The Eclipse GlassFish provides out of the box implementation of `Pbkdf2PasswordHash` that supports PBKDF2 password hashing. It is suggested that you use `Pbkdf2PasswordHash` for generating and validating passwords, unless there are specific requirements which cannot be met any other way.

Custom Authentication Mechanism

An application provider can choose to provide its own custom authentication mechanism, apart from built-in authentication mechanism.

A custom authentication mechanism implements the `HttpAuthenticationMechanism` interface, introduced in Jakarta EE Security API. This interface defines the following three methods.

```
AuthenticationStatus validateRequest(HttpServletRequest request,
                                    HttpServletResponse response,
                                    HttpContext httpMessageContext
                                    ) throws AuthenticationException;

AuthenticationStatus secureResponse(HttpServletRequest request,
                                    HttpServletResponse response,
                                    HttpContext httpMessageContext
                                    ) throws AuthenticationException;

void cleanSubject(HttpServletRequest request,
                  HttpServletResponse response,
                  HttpContext httpMessageContext);
```

`HttpAuthenticationMechanism` returns `AuthenticationStatus` to indicate the status of authentication request. Internally, it gets translated to corresponding JASPIC `AuthStatus` as shown below:

- `AuthenticationStatus.NOT_DONE` to `AuthStatus.SUCCESS`
- `AuthenticationStatus.SEND_CONTINUE` to `AuthStatus.SEND_CONTINUE`
- `AuthenticationStatus.SUCCESS` to `AuthStatus.SUCCESS`
- `AuthenticationStatus.SEND_FAILURE` to `AuthStatus.SEND_FAILURE`

Each method of the `HttpAuthenticationMechanism` interface performs the same function as the corresponding `ServerAuth` methods. Unlike JASPIC, `HttpAuthenticationMechanism` is specified for the servlet container only. Only the `validateRequest()` must be implemented, for other two methods, default behaviors are specified.

`validateRequest` allows a caller to authenticate. The request gets inspected inside `validateRequest` to read credential or any other information, or it can write to standard response with status of the authentication request or redirect the caller to an OAuth provider. Once the credential is validated, the result of the validation is communicated to the container using the `HttpContext`

parameter.

Sample Http Authentication Mechanism

The class `MyAuthenticationMechanism.java` is a sample `HttpAuthenticationMechanism` implementation. Note that only `validateRequest` method has been implemented, since Jakarta EE Security API provides default implementation of other two methods. An application provider may choose to override the default implementation depending on the requirement.

```
import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import javax.security.enterprise.AuthenticationException;
import javax.security.enterprise.AuthenticationStatus;
import
javax.security.enterprise.authentication.mechanism.http.HttpAuthenticationMechanism;
import javax.security.enterprise.authentication.mechanism.http.HttpMessageContext;
import javax.security.enterprise.credential.UsernamePasswordCredential;
import javax.security.enterprise.identitystore.CredentialValidationResult;
import javax.security.enterprise.identitystore.IdentityStoreHandler;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import static
javax.security.enterprise.identitystore.CredentialValidationResult.Status.VALID;

@RequestScoped
public class MyAuthenticationMechanism implements HttpAuthenticationMechanism {

    @Inject
    private IdentityStoreHandler identityStoreHandler;

    @Override
    public AuthenticationStatus validateRequest(HttpServletRequest request,
        HttpServletResponse response, HttpMessageContext httpMessageContext) throws
        AuthenticationException {

        // Get the (caller) name and password from the request
        // NOTE: This is for the smallest possible example only. In practice
        // putting the password in a request query parameter is highly
        // insecure and is discouraged.
        String name = request.getParameter("name");
        String password = request.getParameter("password");

        if (name != null && password != null) {

            // Delegate the {credentials in -> identity data out} function to
            // the Identity Store
            CredentialValidationResult result = identityStoreHandler.validate(
                new UsernamePasswordCredential(name, password));
```

```

        if (result.getStatus() == VALID) {
            // Communicate the details of the authenticated user to the
            // container.
            response.addHeader("Authentication Mechanism",
                "MyAuthenticationMechanism");
            return httpMessageContext.notifyContainerAboutLogin(
                result.getCallerPrincipal(), result.getCallerGroups());
        } else {
            return httpMessageContext.responseUnauthorized();
        }
    }

    return httpMessageContext.doNothing();
}
}

```

The Eclipse GlassFish and JSR 196

The Eclipse GlassFish 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 Eclipse GlassFish 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 Eclipse GlassFish 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;
private 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 Eclipse GlassFish 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 [Eclipse GlassFish 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 `glassfish-web.xml` file. Set the value of the `httpServlet-security-provider` attribute of the `glassfish-web-app` element to the SAM's configured provider ID, for example, `MySAM`. For more information about the `glassfish-web.xml` file, see the [Eclipse GlassFish Application Deployment Guide](#). This option leverages the native `AuthConfigProvider` implementation that ships with the Eclipse GlassFish.
- Another approach is to develop your own `AuthConfigProvider` and register it with the Eclipse GlassFish `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 10. Developing Web Services

This chapter describes Eclipse GlassFish support for web services. The following topics are addressed here:

- [Creating Portable Web Service Artifacts](#)
- [Deploying a Web Service](#)
- [The Web Service URI, WSDL File, and Test Page](#)
- [The Databinding Provider](#)



If you installed the Web Profile, web services are not supported. Without the Metro add-on component, a servlet or EJB component cannot be a web service endpoint, and the `glassfish-web.xml` and `glassfish-ejb-jar.xml` elements related to web services are ignored.

In addition, the `as-install/modules/webservices-osgi.jar` and `as-install/modules/webservices-api-osgi.jar` files must be in the classpath of your web services application. IDEs such as NetBeans and Eclipse do this automatically.

"[Web Services](#)" in The Jakarta EE Tutorial shows how to deploy simple web services to Eclipse GlassFish.

For additional information about JAXB (Java Architecture for XML Binding), see [Java Specification Request \(JSR\) 222](#) and [The Databinding Provider](#).

For additional information about JAX-WS (Java API for XML-Based Web Services), see [Java Specification Request \(JSR\) 224](#) and [Java Specification Request \(JSR\) 109](#).

For information about web services security, see [Configuring Message Security for Web Services](#).

The Fast Infoset standard specifies a binary format based on the XML Information Set. This format is an efficient alternative to XML. For more information about using Fast Infoset, see the [Metro WSIT Guide](#)

Creating Portable Web Service Artifacts

For a tutorial that shows how to use the `wsimport` and `wsgen` commands, see "[Web Services](#)" in The Jakarta EE Tutorial.

Deploying a Web Service

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



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

The Eclipse GlassFish deployment descriptor files `glassfish-web.xml` and `glassfish-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](#).

For more information about deployment, autodeployment, and deployment descriptors, see the [Eclipse GlassFish Application Deployment Guide](#). For more information about the `asadmin deploy` command, see the [Eclipse GlassFish 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 `glassfish-application.xml` or `glassfish-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.



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 `glassfish-web.xml` and `glassfish-ejb-jar.xml` deployment descriptor to `false`. For more information, see the [Eclipse GlassFish Application Deployment Guide](#).

The Databinding Provider

The JAX-WS reference implementation (RI) used to be dependent on the JAXB RI for databinding. JAXB and JAX-WS implementations have been decoupled, and databinding is modular. JAXB and JAX-WS are no longer Jakarta EE APIs.

The EclipseLink JAXB implementation, plus EclipseLink extensions, is called MOXy. The `org.eclipse.persistence.moxy.jar` file is bundled with Eclipse GlassFish, which supports the JAXB RI and MOXy as databinding providers.

To specify the databinding provider for the JVM, set the `com.sun.xml.ws.spi.db.BindingContextFactory` JVM property to one of the following values:

`com.sun.xml.ws.db.glassfish.JAXBRIContextFactory`

Specifies the JAXB reference implementation. This is the default.

`com.sun.xml.ws.db.toplink.JAXBContextFactory`

Specifies EclipseLink MOXy JAXB binding.

For example:

```
asadmin create-jvm-options
-Dcom.sun.xml.ws.spi.db.BindingContextFactory=com.sun.xml.ws.db.toplink.JAXBContextFactory
```

To specify the databinding provider for a web service endpoint:

- Set the `org.jvnet.ws.databinding.DatabindingModeFeature` feature during `WebServiceFeature` initialization or using the `add` method.

Allowed values are as follows

`org.jvnet.ws.databinding.DatabindingModeFeature.GLASSFISH_JAXB`

Specifies the JAXB reference implementation. This is the default.

`com.sun.xml.ws.db.toplink.JAXBContextFactory.ECLIPSELINK_JAXB`

Specifies EclipseLink MOXy JAXB binding.

For example:

```
import jakarta.xml.ws.WebServiceFeature;
import org.jvnet.ws.databinding.DatabindingModeFeature;
import com.sun.xml.ws.db.toplink.JAXBContextFactory;
...
WebServiceFeature[] features = {new DatabindingModeFeature(
```

```
JAXBContextFactory.ECLIPSELINK_JAXB));  
...
```

- Set the `org.jvnet.ws.databinding.DatabindingModeFeature` feature using the `@DatabindingMode` annotation. For example:

```
import jakarta.jws.WebService;  
import org.jvnet.ws.databinding.DatabindingMode;  
import com.sun.xml.ws.db.toplink.JAXBContextFactory;  
...  
@WebService()  
@DatabindingMode(JAXBContextFactory.ECLIPSELINK_JAXB);  
...
```

- Set the `databinding` attribute of the `endpoint` element in the `sun-jaxws.xml` file. Allowed values are `glassfish.jaxb` or `eclipselink.jaxb`. For example:

```
<endpoint name='hello'  
  implementation='hello.HelloImpl'  
  url-pattern='/hello'  
  databinding='eclipselink.jaxb'  
>
```

The EclipseLink JAXB compiler is not included but can be used with Eclipse GlassFish. Download the EclipseLink zip file at <http://www.eclipse.org/eclipselink/downloads/> and unzip it. The compiler files are located here:

```
bin/jaxb-compiler.cmd  
bin/jaxb-compiler.sh
```

Chapter 11. Configuring the Java Persistence Provider

This chapter describes Oracle TopLink, the default persistence provider in Eclipse GlassFish, and introduces how to use it. This chapter also tells how to set the default persistence provider in Eclipse GlassFish and how to use persistence-related features specific to Eclipse GlassFish such as automatic schema generation.

The following topics are addressed here:

- [Overview of Oracle TopLink](#)
- [Using Oracle TopLink in Eclipse GlassFish](#)
- [Specifying the Database for an Application](#)
- [Specifying the Persistence Provider for an Application](#)
- [Primary Key Generation Defaults](#)
- [Automatic Schema Generation](#)
- [Restrictions and Optimizations](#)

Overview of Oracle TopLink

Oracle TopLink is the default persistence provider in Eclipse GlassFish. It is a comprehensive standards-based object-persistence and object-transformation framework that provides APIs, schemas, and run-time services for the persistence layer of an application.

TopLink includes all of EclipseLink, from the Eclipse Foundation. EclipseLink is the default persistence provider in Eclipse GlassFish. EclipseLink is the open source implementation of the development framework and the runtime provided in TopLink. EclipseLink implements the following specifications, plus value-added extensions:

- Java Persistence Architecture (JPA) 2.0.

JPA 2.0 is part of Java Platform, Enterprise Edition 6 (Jakarta EE 6). It includes improvements and enhancements to domain modeling, object/relational mapping, `EntityManager` and `Query` interfaces, and the Java Persistence Query Language (JPQL). It includes an API for criteria queries, a metamodel API, and support for validation. The Java Persistence API can be used with non-EJB components outside the EJB container.

For the JPA 2.0 Specification, see [Java Specification Request \(JSR\) 317](#). For basic information about the Java Persistence API, see [Persistence](#) in The Jakarta EE Tutorial.

- Java Architecture for XML Binding (JAXB) 2.0. The EclipseLink JAXB implementation, plus EclipseLink extensions, is called MOXy. The `org.eclipse.persistence.moxy.jar` file is bundled with Eclipse GlassFish. For more information about MOXy support in Eclipse GlassFish, see [The Databinding Provider](#).

For the JAXB 2.0 specification, see [Java Specification Request \(JSR\) 222](#).

- EclipseLink utilities are not included but can be used with Eclipse GlassFish. Download the EclipseLink zip file at <http://www.eclipse.org/eclipselink/downloads/> and unzip it. The utility files are located here:

```
bin/jaxb-compiler.cmd  
bin/jaxb-compiler.sh
```

In addition to all of EclipseLink, Oracle TopLink includes TopLink Grid, an integration between TopLink and Oracle Coherence that allows TopLink to use Oracle Coherence as a level 2 (L2) cache and persistence layer for entities. The `toplink-grid.jar` file is bundled with Eclipse GlassFish.



You must have a license for Oracle Coherence to be able to use TopLink Grid.

For information about developing, deploying, and configuring Oracle TopLink, EclipseLink, and TopLink Grid applications, see the following:

- [Oracle Fusion Middleware Solution Guide for Oracle TopLink](#)
- EclipseLink project home at <http://wiki.eclipse.org/EclipseLink>
- EclipseLink Documentation Center at <http://wiki.eclipse.org/EclipseLink/UserGuide>
- Java API Reference for EclipseLink at <http://www.eclipse.org/eclipselink/api/latest/index.html>
- EclipseLink examples at <http://wiki.eclipse.org/EclipseLink/Examples>
- [Oracle Coherence Developer's Guide](#)
- [Oracle Fusion Middleware Integration Guide for Oracle TopLink with Coherence Grid](#)

Using Oracle TopLink in Eclipse GlassFish

To run TopLink JPA applications in Eclipse GlassFish, you must configure the server and coordinate certain server and application settings. These are described in the following steps. For a summary of these steps, see "Using TopLink with WebLogic Server" in Oracle Fusion Middleware Solution Guide for Oracle TopLink. For more detailed explanations of these steps, see the links in the steps.

1. Set up the datasource. See " [Administering Database Connectivity](#)" in Eclipse GlassFish Administration Guide.
2. Create the application. For guidance in writing your application, see [Persistence](#) in The Jakarta EE Tutorial.
3. Create the `persistence.xml` file. See [Specifying the Database for an Application](#) for considerations specific to Eclipse GlassFish.

If you are using the Java Persistence API by calling `Persistence.createEMF()`, see [Specifying the Persistence Provider for an Application](#).

4. If the security manager is enabled and you are using the Java Persistence API by calling

`Persistence.createEMF()`, see [Enabling and Disabling the Security Manager](#).

5. Deploy the application. See the Eclipse GlassFish Application Deployment Guide.
6. Run the application. See "Application Client Launch" and "To Launch an Application" in Administration Console online help.
7. Monitor the application. See " [Administering the Monitoring Service](#)" in Eclipse GlassFish Administration Guide.

Specifying the Database for an Application

Eclipse GlassFish uses the bundled Apache Derby database by default, named `jdbc/__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, Apache Derby 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, Apache Derby 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 Eclipse GlassFish and are found to be Jakarta 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](http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_for_Session,_Target_Database_and_Target_Application_Server) ([http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_\(ELUG\)#Using_EclipseLink_JPA_Extensions_for_Session.2C_Target_Database_and_Target_Application_Server](http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_(ELUG)#Using_EclipseLink_JPA_Extensions_for_Session.2C_Target_Database_and_Target_Application_Server)).

If you are using the Java Persistence API by calling `Persistence.createEMF()`, 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 a list of the JDBC drivers currently supported by the Eclipse GlassFish, see the [Eclipse GlassFish Release Notes](#). For configurations of supported and other drivers, see "[Configuration Specifics for JDBC Drivers](#)" in Eclipse GlassFish Administration Guide.

Specifying the Persistence Provider for an Application

If you are using the default persistence provider in an application that uses the Java Persistence API by injecting or looking up an entity manager or entity manager factory, you do not need to specify

the provider.

If you are using the Java Persistence API by calling `Persistence.createEMF()`, you should always specify the persistence provider for specification compliance. To specify the default provider, set the `provider` element of the `persistence.xml` file to `org.eclipse.persistence.jpa.PersistenceProvider`.

You can specify a non-default persistence provider for an application in the manner described in the Java Persistence API Specification:

1. Install the provider. Copy the provider JAR files to the domain-dir/`lib` directory, and restart the Eclipse GlassFish. For more information about the domain-dir/`lib` directory, see [Using the Common Class Loader](#). 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, Oracle TopLink or EclipseLink.

2. 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 specify the provider programmatically instead of in the `persistence.xml` file, set the `javax.persistence.provider` property and pass it to the `Map` parameter of the following method:

```
javax.persistence.Persistence.createEntityManagerFactory(String, Map)
```

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 Eclipse GlassFish 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.

The following topics are addressed here:

- [Annotations](#)
- [Generation Options](#)



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



Before using these options, make sure you have a properly configured database. See [Specifying the Database for an Application](#).

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](#) ([http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_\(ELUG\)#Using_EclipseLink_JPA_Extensions_for_Schema_Generation](http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_(ELUG)#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 6-1 The `asadmin deploy` and `asadmin deploydir` Generation Options

Option	Default	Description
<code>--createtables</code>	none	If <code>true</code> , causes database tables to be created for entities that need them. No unique constraints are created. If <code>false</code> , does not create tables. If not specified, the value of the <code>eclipseLink.ddl-generation</code> property in <code>persistence.xml</code> is used.

Option	Default	Description
<code>--dropandcreatetables</code>	none	<p>If <code>true</code>, and if tables were automatically created when this application was last deployed, tables from the earlier deployment are dropped and fresh ones are created.</p> <p>If <code>true</code>, 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.</p> <p>If <code>false</code>, the <code>eclipselink.ddl-generation</code> property setting in <code>persistence.xml</code> is overridden.</p>

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

Table 6-2 The `asadmin undeploy` Generation Options

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

For more information about the `asadmin deploy`, `asadmin deploydir`, and `asadmin undeploy` commands, see the [Eclipse GlassFish Reference Manual](#).

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

Restrictions and Optimizations

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

The following topics are addressed here:

- [Oracle Database Enhancements](#)
- [Extended Persistence Context](#)
- [Using @OrderBy with a Shared Session Cache](#)
- [Using BLOB or CLOB Types with the Inet Oraxo JDBC Driver](#)
- [Database Case Sensitivity](#)
- [Sybase Finder Limitation](#)
- [MySQL Database Restrictions](#)

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 Eclipse GlassFish 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 for an Application](#).

Extended Persistence Context

The Java Persistence API specification does not specify how the container and persistence provider should work together to serialize an extended persistence context. This also prevents successful serialization of a reference to an extended persistence context in a stateful session bean.

Even in a single-instance environment, if a stateful session bean is passivated, its extended persistence context could be lost when the stateful session bean is activated.

Therefore, in Eclipse GlassFish, a stateful session bean with an extended persistence context is never passivated and cannot be failed over.

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 Eclipse GlassFish 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. 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 Eclipse GlassFish 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 Eclipse GlassFish 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://dev.mysql.com/doc/mysql/en/innodb-foreign-key-constraints.html>.

Chapter 12. Using Jakarta Data Repositories

This chapter describes how to use Jakarta Data repositories in Eclipse GlassFish applications. Jakarta Data provides a standard API for data access that simplifies database operations through repository interfaces and automatic query generation.

The following topics are addressed here:

- [Overview of Jakarta Data](#)
- [Choosing Between JPA and NoSQL](#)
- [Basic Repository Example](#)
- [Defining Repository Interfaces](#)
- [Query Methods](#)
- [Custom Queries](#)
- [Method Name-Based Queries \(Deprecated\)](#)
- [Pagination and Sorting](#)
- [Configuring JPA Data Repositories](#)
- [Configuring NoSQL Data Repositories](#)
- [Transaction Management](#)
- [Jakarta Validation Support](#)
- [Limitations and Considerations](#)

For more information about Jakarta Data 1.0.1, see the official [Jakarta Data 1.0.1 documentation](#).

Overview of Jakarta Data

Jakarta Data is a specification that provides a standard API for data access in Jakarta EE applications. It offers a repository-based programming model that reduces boilerplate code and provides type-safe database operations.

Key features of Jakarta Data include:

- Repository interfaces with automatic implementation generation
- Query derivation from method names and annotations
- Support for custom queries using JDQL (Jakarta Data Query Language) or JPQL
- Built-in pagination and sorting capabilities
- Integration with Jakarta Persistence (JPA) for relational databases
- Integration with Jakarta NoSQL for NoSQL databases
- Type-safe query building
- Support for both SQL and NoSQL databases in the same application

Jakarta Data repositories work by defining interfaces that may extend base repository types. The Jakarta Data provider automatically generates implementations of these interfaces at build time or runtime.

In Eclipse GlassFish, Jakarta Data is implemented using Eclipse JNoSQL, which provides:

- **Unified API** - The same repository interfaces for both SQL and NoSQL databases
- **Multi-database support** - Support for SQL, Document, Key-Value, Wide-Column, and Graph databases
- **JPA integration** - Seamless integration with existing JPA entities for relational databases
- **NoSQL entities** - Support for Jakarta NoSQL entities for NoSQL databases
- **Automatic routing** - Automatic database detection and query creation based on entity type

Choosing Between JPA and NoSQL

When deciding between JPA entities and NoSQL entities for your Jakarta Data repositories, consider the following factors:

Factor	JPA Entities	NoSQL Entities
Database Type	Relational databases (PostgreSQL, MySQL, Oracle, etc.)	NoSQL databases (MongoDB, Redis, Cassandra, Neo4j, etc.)
Data Structure	Structured data with relationships	Flexible schemas, nested documents, key-value pairs
Transactions	Full JTA transaction support	Limited transaction support (database-dependent)
Validation	Jakarta Validation supported	Manual validation required
Relationships	Full relationship mapping (@OneToMany, @ManyToOne, etc.)	No relationship mapping (embed or reference manually)
Query Language	JPQL and JDQL	JDQL only
Schema Evolution	Requires database migrations	Flexible schema changes
Scalability	Vertical scaling (with some horizontal options)	Horizontal scaling
ACID Properties	Full ACID compliance	Eventual consistency (varies by database)

Use JPA entities when:

- You need strong consistency and ACID transactions
- Your data has complex relationships
- You require Jakarta Validation
- You're working with existing relational database schemas

Use NoSQL entities when:

- You need flexible schemas and rapid development
- You're working with large-scale, distributed data
- Your data is document-oriented or graph-based
- You need horizontal scalability

Basic Repository Example

This example shows a simple repository interface that extends `CrudRepository` for a `Product` JPA entity. It demonstrates both the modern annotated approach and the deprecated method name-based approach.

JPA Entity Example

```
import jakarta.persistence.*;
import jakarta.validation.constraints.*;

@Entity
@Table(name = "products")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(unique = true)
    @NotBlank(message = "Product code is required")
    @Size(max = 20, message = "Product code must not exceed 20 characters")
    private String code;

    @NotBlank(message = "Product name is required")
    @Size(max = 100, message = "Product name must not exceed 100 characters")
    private String name;

    private String category;

    @NotNull(message = "Price is required")
    @DecimalMin(value = "0.0", inclusive = false, message = "Price must be greater than 0")
    private BigDecimal price;

    // Constructors, getters, and setters
}
```

Repository Interface (Recommended Approach)

```
import jakarta.data.repository.CrudRepository;
import jakarta.data.repository.Repository;
import jakarta.data.repository.Find;
import jakarta.data.repository.By;
import jakarta.data.repository.OrderBy;
import jakarta.data.repository.Query;

@Repository
public interface ProductRepository extends CrudRepository<Product, Long> {

    // Annotated queries using @Find and @By annotations (RECOMMENDED)
    @Find
    List<Product> findProducts(@By("name") String name);

    @Find
    Optional<Product> findProduct(@By("code") String code);

    @Find
    @OrderBy("price")
    List<Product> findProducts(@By("category") String category);

    // Custom query using JSQL
    @Query("WHERE price BETWEEN :minPrice AND :maxPrice ORDER BY price")
    List<Product> findProductsByPriceRange(BigDecimal minPrice, BigDecimal maxPrice);
}
```

Defining Repository Interfaces

Jakarta Data repositories are defined as interfaces annotated with `@Repository` annotation (`@jakarta.data.Repository`). The interface can extend one of the base repository interfaces, which provide ready-to-use methods:

- `Repository<T, K>` - Marker interface with no predefined methods
- `CrudRepository<T, K>` - Adds basic CRUD operations (save, findById, findAll, delete, etc.)
- `PageableRepository<T, K>` - Extends `CrudRepository` and adds pagination support

Query Methods

Jakarta Data supports automatic query generation using annotated methods with query annotations like `@Find`, `@Count`, `@Exists`, and `@Delete` combined with parameter annotations like `@By`. This is the recommended approach for defining repository methods.

Supported Query Annotations

- `@Find` - Find operations that return entities

- **@Count** - Count operations that return the number of matching entities
- **@Exists** - Existence checks that return boolean values
- **@Delete** - Delete operations that remove entities
- **@Insert** - Insert operations that create new entities
- **@Update** - Update operations that modify existing entities
- **@Save** - Save operations that insert or update entities

Parameter Annotations

- **@By("propertyName")** - Specifies the entity property to query by. Can be nested to specify properties of referenced entities (e.g., **@By("customer.address.city")**).



If the application is compiled with parameter names preserved in bytecode (e.g., with `javac -parameters`), **@By** is not required. The property name will be derived from the parameter name if the **@By** annotation is not present.

Annotated Query Examples

```
import jakarta.data.repository.*;

@Repository
public interface CustomerRepository extends CrudRepository<Customer, Long> {

    // Find by single property
    @Find
    List<Customer> findCustomers(@By("lastName") String lastName);

    // Multiple conditions (implicit AND)
    @Find
    List<Customer> findCustomers(@By("firstName") String firstName, @By("lastName")
String lastName);

    // Nested property access
    @Find
    List<Customer> findCustomers(@By("address.city") String city);

    // Counting
    @Count
    long countCustomers(@By("status") String status);

    // Existence checks
    @Exists
    boolean customerExists(@By("email") String email);

    // Delete operations
    @Delete
    void deleteCustomers(@By("status") String status);
}
```



```

// Insert operations
@Insert
Customer insertCustomer(Customer customer);

@Insert
List<Customer> insertCustomers(List<Customer> customers);

// Update operations
@update
Customer updateCustomer(Customer customer);

// Save operations (insert or update)
@Save
Customer saveCustomer(Customer customer);

// Null value handling
@Find
List<Customer> findCustomers(@By("middleName") String middleName);

default List<Customer> findCustomersWithNullMiddleName() {
    return findCustomers((String) null);
}
}

```

Custom Queries

For complex queries that cannot be expressed through annotated methods, Jakarta Data supports custom queries using the `@Query` annotation. Jakarta Data specifies its own query language called JDQL (Jakarta Data Query Language), which is similar to JPQL (Java Persistence Query Language). Eclipse GlassFish also supports full JPQL queries for JPA entities.

JDQL vs JPQL

JDQL (Jakarta Data Query Language):

- Simplified syntax that allows omitting the SELECT clause
- Automatically infers the entity type from the method return type
- Works with both JPA and NoSQL entities
- Example: `WHERE customer.id = :customerId AND status = :status`

JPQL (Java Persistence Query Language):

- Full SQL-like syntax with explicit SELECT clauses
- Only works with JPA entities
- More powerful for complex joins and projections
- Example: `SELECT o FROM Order o WHERE o.customer.id = :customerId AND o.status = :status`

Custom Query Examples

```
import jakarta.data.repository.Query;

@Repository
public interface OrderRepository extends CrudRepository<Order, Long> {

    // JDQL query - simplified syntax
    @Query("WHERE customer.id = :customerId AND status = :status")
    List<Order> findOrdersByCustomerAndStatus(Long customerId, String status);

    // JDQL query with range conditions
    @Query("WHERE orderDate BETWEEN :startDate AND :endDate ORDER BY orderDate DESC")
    List<Order> findOrdersByDateRange(LocalDate startDate, LocalDate endDate);

    // JDQL query with pattern matching
    @Query("WHERE customer.email LIKE :pattern")
    List<Order> findOrdersByCustomerEmailPattern(String pattern);

    // JDQL query with collection operations
    @Query("WHERE status IN :statuses")
    List<Order> findOrdersByStatuses(Collection<String> statuses);

    // JPQL query with explicit SELECT and JOIN (JPA entities only)
    @Query("SELECT o FROM Order o JOIN o.items i WHERE i.product.category = :category")
    List<Order> findOrdersWithProductCategory(String category);

    // JPQL query with aggregation (JPA entities only)
    @Query("SELECT COUNT(o) FROM Order o WHERE o.orderDate >= :startDate")
    Long countOrdersSince(LocalDate startDate);

    // JPQL query with projection (JPA entities only)
    @Query("SELECT NEW com.example.dto.OrderSummary(o.id, o.customer.name, o.total) FROM Order o WHERE o.status = :status")
    List<OrderSummary> findOrderSummariesByStatus(String status);
}
```

Method Name-Based Queries (Deprecated)



Method name-based query derivation is deprecated and may be removed in future versions. It's provided for migrating existing code to Data repositories. For new code, use the annotated approach shown in the previous sections instead.

Eclipse GlassFish also supports method name-based query derivation using a standardized naming convention. While this approach is deprecated, it's still supported for backward compatibility and migration purposes.

Supported Keywords

- `findBy`, `getBy`, `queryBy`, `readBy` - Find operations
- `countBy` - Count operations
- `existsBy` - Existence checks
- `deleteBy`, `removeBy` - Delete operations

Property Expressions

- `And`, `Or` - Logical operators
- `Between` - Range queries
- `LessThan`, `GreaterThan`, `LessThanEqual`, `GreaterThanEqual` - Comparison operators
- `Like`, `NotLike` - Pattern matching
- `In`, `NotIn` - Collection membership
- `IsNull`, `IsNotNull` - Null checks
- `True`, `False` - Boolean values
- `OrderBy` - Sorting (can be combined with `Asc` or `Desc`)

Method Name-Based Query Examples

```
@Repository
public interface CustomerRepository extends CrudRepository<Customer, Long> {

    // Find by single property
    List<Customer> findByLastName(String lastName);

    // Multiple conditions with And
    List<Customer> findByFirstNameAndLastName(String firstName, String lastName);

    // Or conditions
    List<Customer> findByFirstNameOrLastName(String name);

    // Comparison operators
    List<Customer> findByAgeGreaterThan(int age);
    List<Customer> findByAgeLessThan(int age);
    List<Customer> findByAgeGreaterThanEqual(int age);
    List<Customer> findByAgeLessThanEqual(int age);
    List<Customer> findByAgeBetween(int minAge, int maxAge);

    // Pattern matching
    List<Customer> findByEmailLike(String pattern);
    List<Customer> findByEmailNotLike(String pattern);

    // Collection operations
    List<Customer> findByStatusIn(Collection<String> statuses);
}
```

```

List<Customer> findByStatusNotIn(Collection<String> statuses);

// Null checks
List<Customer> findByMiddleNameIsNull();
List<Customer> findByMiddleNameIsNotNull();

// Boolean values
List<Customer> findByActiveTrue();
List<Customer> findByActiveFalse();

// Sorting
List<Customer> findByLastNameOrderByFirstNameAsc(String lastName);
List<Customer> findByLastNameOrderByFirstNameDesc(String lastName);
List<Customer> findByStatusOrderByLastNameAscFirstNameAsc(String status);

// Counting
long countByStatus(String status);
long countByAgeGreaterThan(int age);

// Existence checks
boolean existsByEmail(String email);
boolean existsByLastNameAndFirstName(String lastName, String firstName);

// Delete operations
void deleteByStatus(String status);
long removeByAgeGreaterThan(int age);

// Complex combinations
List<Customer> findByLastNameAndAgeGreaterThanAndActiveTrue(String lastName, int
age);
List<Customer> findByEmailLikeOrPhoneLike(String emailPattern, String
phonePattern);
}

```

Nested Property Access

Method names can also access nested properties using camel case:

```

@Repository
public interface OrderRepository extends CrudRepository<Order, Long> {

    // Access nested properties
    List<Order> findByCustomerLastName(String lastName);
    List<Order> findByCustomerAddressCity(String city);
    List<Order> findByCustomerAddressCountry(String country);

    // Complex nested queries
    List<Order> findByCustomerLastNameAndShippingAddressCity(String lastName, String
city);
    List<Order> findByCustomerActiveAndOrderDateGreaterThan(boolean active, LocalDate

```

```
date);  
}
```

Pagination and Sorting

Jakarta Data provides built-in support for pagination and sorting through the `Pageable` and `Sort` interfaces.

Using Pageable

```
import jakarta.data.repository.Pageable;  
import jakarta.data.repository.Page;  
import jakarta.data.repository.Sort;  
import jakarta.data.repository.Find;  
import jakarta.data.repository.By;  
  
@Repository  
public interface ProductRepository extends PageableRepository<Product, Long> {  
  
    @Find  
    Page<Product> findProducts(@By("category") String category, Pageable pageable);  
  
    @Query("WHERE price > :price")  
    List<Product> findProductsAbovePrice(BigDecimal price, Sort sort);  
}
```

Usage Example

```
@Inject  
private ProductRepository productRepository;  
  
public void demonstratePagination() {  
    // Create pageable request for page 0, size 10, sorted by name  
    Pageable pageable = Pageable.of(0, 10, Sort.by("name").ascending());  
  
    Page<Product> page = productRepository.findProducts("Electronics", pageable);  
  
    List<Product> products = page.getContent();  
    long totalElements = page.getTotalElements();  
    int totalPages = page.getTotalPages();  
    boolean hasNext = page.hasNext();  
}
```

Configuring JPA Data Repositories

To use Jakarta Data repositories with JPA entities in your Eclipse GlassFish application, you need to:

1. **Add Jakarta Data API dependency** to your project
2. **Configure a Jakarta Persistence persistence unit**
3. **Set up database connection** (optional - uses default Derby database if not specified)

Adding Dependencies

Add the Jakarta Data API dependency to your `pom.xml`:

```
<dependency>
  <groupId>jakarta.data</groupId>
  <artifactId>jakarta.data-api</artifactId>
  <version>1.0.1</version>
  <scope>provided</scope>
</dependency>
```

Alternatively, if you're using the full Jakarta EE platform:

```
<dependency>
  <groupId>jakarta.platform</groupId>
  <artifactId>jakarta.jakartaee-api</artifactId>
  <version>11.0.0</version>
  <scope>provided</scope>
</dependency>
```

Jakarta Persistence Configuration

Configure your persistence unit in `persistence.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence" version="3.0">
  <persistence-unit name="default" transaction-type="JTA">
    <properties>
      <property name="jakarta.persistence.schema-generation.database.action"
value="create"/>
    </properties>
  </persistence-unit>
</persistence>
```

This configuration uses the default data source in Eclipse GlassFish, which points to a Derby database. Make sure Derby is running using the `start-database` command.

Custom Data Source Configuration (Recommended)

For production applications, configure a custom data source using the Eclipse GlassFish administration console or CLI commands:

```
# Create connection pool for PostgreSQL
asadmin create-jdbc-connection-pool \
  --datasourceclassname org.postgresql.ds.PGSimpleDataSource \
  --restype javax.sql.DataSource \
  --property
user=myuser:password=mypassword:databaseName=mydb:serverName=localhost:port=5432 \
  MyPool

# Create JDBC resource
asadmin create-jdbc-resource --connectionpoolid MyPool jdbc/MyDataSource
```

Adjust the `datasourceclassname` and properties according to your database and JDBC driver.



For information on how to install JDBC drivers in Eclipse GlassFish, see [Administering Database Connectivity](#).



For more information about these commands, see `create-jdbc-connection-pool` and `create-jdbc-resource` in the Reference Manual.

Then reference the custom data source in your `persistence.xml`:

```
<persistence-unit name="myPU" transaction-type="JTA">
  <jta-data-source>jdbc/MyDataSource</jta-data-source>
  <properties>
    <property name="jakarta.persistence.schema-generation.database.action" value=
"create"/>
  </properties>
</persistence-unit>
```

EntityManager Integration

You can access the EntityManager used by the repository for custom operations:

```
@Repository
public interface ProductRepository extends CrudRepository<Product, Long> {

    // Method to access the repository's EntityManager
    EntityManager getEntityManager();

    // Custom method using the repository's EntityManager
    default List<Product> findProductsWithComplexLogic(String searchTerm) {
        EntityManager em = getEntityManager();
        return em.createQuery(
            "SELECT p FROM Product p WHERE p.name LIKE :term OR p.description LIKE
:term",
            Product.class)
            .setParameter("term", "%" + searchTerm + "%")
    }
}
```

```

        .getResultList();
    }
}

```

EntityManager Configuration Options

By default, Jakarta Data repositories use the default `EntityManager`. You can customize this behavior:

Option 1: Specify persistence unit name

Use the `dataStore` attribute of the `@Repository` annotation to specify which persistence unit the repository should use.

```

@Repository(dataStore = "myPU")
public interface ProductRepository extends CrudRepository<Product, Long> {
    // Repository methods
}

```

Option 2: Use CDI qualifiers

Define a custom CDI qualifier and use it to inject a specific `EntityManager`.

```

@Repository
public interface ProductRepository extends CrudRepository<Product, Long> {

    @CustomDB
    EntityManager getEntityManager();
}

```

Option 3: Programmatic EntityManager selection

Define a default method in the repository interface that returns an `EntityManager`. This approach allows you to programmatically select the appropriate `EntityManager` using CDI lookup, JNDI lookup, or any other mechanism.

```

@Repository
public interface ProductRepository extends CrudRepository<Product, Long> {

    default EntityManager getEntityManager() {
        return CDI.current().select(EntityManager.class, new CustomDB.Literal()).get();
    }
}

```


Mixing Repository and EntityManager Operations

Repository methods and direct EntityManager operations can be used together in the same transaction when they use the same persistence unit:

```
@ApplicationScoped
@Transactional
public class CustomerService {

    @Inject
    private EntityManager entityManager;

    @Inject
    private CustomerRepository customerRepository;

    public Customer updateCustomerWithAudit(Long customerId, CustomerUpdate update) {

        EntityManager repositoryEntityManager = customerRepository.getEntityManager();

        // Use repository for simple operations
        Customer customer = customerRepository.findById(customerId)
            .orElseThrow(() -> new EntityNotFoundException("Customer not found"));

        // Use EntityManager for complex operations
        AuditLog audit = new AuditLog();
        audit.setAction("UPDATE_CUSTOMER");
        audit.setEntityId(customerId);
        audit.setTimestamp(LocalDateTime.now());
        entityManager.persist(audit);

        // Alternatively, use the repository's EntityManager to ensure the same
        persistence context
        repositoryEntityManager.flush();

        // Update using repository
        customer.setName(update.getName());
        customer.setEmail(update.getEmail());

        return customerRepository.save(customer);
    }
}
```

Configuring NoSQL Data Repositories

To use Jakarta Data repositories with NoSQL entities in your Eclipse GlassFish application, you need to:

1. **Add Jakarta Data and NoSQL API dependencies** to your project
2. **Add NoSQL database driver dependency**

3. **Configure database connection properties**
4. **Define NoSQL entities** using Jakarta NoSQL annotations

Adding Dependencies

Add the Jakarta Data API dependency (as provided - only compile-time, not included in the application):

```
<dependency>
  <groupId>jakarta.data</groupId>
  <artifactId>jakarta.data-api</artifactId>
  <version>1.0.1</version>
  <scope>provided</scope>
</dependency>
```

Add the Jakarta NoSQL API dependency (as provided - only compile-time, not included in the application):

```
<dependency>
  <groupId>jakarta.nosql</groupId>
  <artifactId>jakarta.nosql-api</artifactId>
  <version>1.0.1</version>
  <scope>provided</scope>
</dependency>
```

Add the appropriate Eclipse JNoSQL database driver (the driver and its dependencies should be added to the application). For example, for MongoDB:

```
<dependency>
  <groupId>org.eclipse.jnosql.databases</groupId>
  <artifactId>jnosql-mongodb</artifactId>
  <version>{jnosql-version}</version>
</dependency>
```



Eclipse GlassFish does not bundle NoSQL database drivers. You must add the appropriate Eclipse JNoSQL database dependency and its transitive dependencies to your application. If you use Maven or a similar build tool, it will pull all transitive dependencies automatically.

NoSQL Database Categories

Eclipse JNoSQL supports four main categories of NoSQL databases:

Category	Examples	Use Cases
Document	MongoDB, CouchDB, CouchBase, ArangoDB	Content management, catalogs, user profiles with nested data
Key-Value	Redis, Hazelcast, Memcached, Riak	Caching, session storage, shopping carts
Wide-Column	Cassandra, HBase, DynamoDB	Time-series data, IoT applications, analytics
Graph	Neo4j, ArangoDB, OrientDB	Social networks, recommendation engines, fraud detection

NoSQL Entity Definition

NoSQL entities use Jakarta NoSQL annotations:

Annotation	Description
<code>@jakarta.nosql.Entity</code>	Specifies that the class is a NoSQL entity
<code>@jakarta.nosql.Id</code>	Specifies the primary key of the entity
<code>@jakarta.nosql.Column</code>	Maps a field to a database column/attribute
<code>@jakarta.nosql.Convert</code>	Specifies a converter for the field
<code>@jakarta.nosql.Embeddable</code>	Specifies that the class is embeddable
<code>@jakarta.nosql.Inheritance</code>	Specifies inheritance mapping strategy for entities
<code>@jakarta.nosql.DiscriminatorColumn</code>	Specifies the discriminator column for the inheritance mapping strategy
<code>@jakarta.nosql.DiscriminatorValue</code>	Specifies the discriminator value for the inheritance mapping strategy
<code>@jakarta.nosql.MappedSuperclass</code>	Specifies a class whose mapping information is applied to entities that inherit from it

Key differences from JPA annotations:

- No `@Table` annotation - collection/table name is specified in `@Entity`
- No relationship annotations (`@OneToMany`, `@ManyToOne`) - embed or reference manually
- No `@GeneratedValue` - primary keys must be assigned manually or assigned automatically by the database
- No `@Version` for optimistic locking
- No `@Transient` - non-annotated fields are ignored by default
- Java `record` types are supported for read-only operations

NoSQL Entity Examples

Class-based entity with embedded objects:

```

import jakarta.nosql.Entity;
import jakarta.nosql.Id;
import jakarta.nosql.Column;
import jakarta.nosql.Embeddable;

@Entity("customers")
public class Customer {

    @Id
    private String id;

    @Column
    private String name;

    @Column
    private String email;

    @Column
    private Address address; // Embedded object

    @Column
    private List<String> tags;

    @Column
    private Map<String, Object> metadata;

    // Constructors, getters, and setters
}

@Embeddable
public class Address {
    @Column
    private String street;

    @Column
    private String city;

    @Column
    private String country;

    // Constructors, getters, and setters
}

```

Record-based entity (read-only):

```

@Entity("sensor_data")
public record SensorReading(
    @Id String id,
    @Column String sensorId,

```

```

@Column LocalDateTime timestamp,
@Column Double temperature,
@Column Double humidity,
@Column Map<String, Double> additionalMetrics
) {}

```

Repository Usage with NoSQL Entities

NoSQL entities work with the same repository interfaces as JPA entities:

```

@Repository
public interface CustomerRepository extends CrudRepository<Customer, String> {

    @Find
    List<Customer> findCustomers(@By("name") String name);

    @Find
    List<Customer> findCustomers(@By("address.city") String city);

    @Query("WHERE tags IN :tags")
    List<Customer> findCustomersByTags(List<String> tags);

    @Count
    long countCustomers(@By("address.country") String country);
}

@Repository
public interface SensorReadingRepository extends CrudRepository<SensorReading, String>
{

    @Find
    List<SensorReading> findReadings(@By("sensorId") String sensorId);

    @Query("WHERE timestamp BETWEEN :start AND :end")
    List<SensorReading> findByTimestampRange(LocalDateTime start, LocalDateTime end);

    @Count
    long countBySensorId(@By("sensorId") String sensorId);
}

```

Database Configuration

NoSQL database connections are configured using MicroProfile Config properties. Configuration varies by database type and specific database.

MongoDB Configuration Example:

Add properties to your `microprofile-config.properties` file:

```
# Document database configuration
jnosql.document.database=mystore
jnosql.mongodb.host=localhost:27017
jnosql.mongodb.user=myuser
jnosql.mongodb.password=mypassword
```

MongoDB-specific properties:

Property	Description
<code>jnosql.mongodb.host</code>	MongoDB server hostname and port (default: localhost:27017)
<code>jnosql.mongodb.user</code>	Username for MongoDB authentication
<code>jnosql.mongodb.password</code>	Password for MongoDB authentication
<code>jnosql.mongodb.authentication.source</code>	The source where the user is defined
<code>jnosql.mongodb.authentication.mechanism</code>	Authentication mechanism (see MongoDB JavaDoc)

Common Configuration Properties:

Property	Description
<code>jnosql.<type>.provider</code>	Fully qualified class name of the database configuration provider
<code>jnosql.<type>.database</code>	Database name or identifier
<code>jnosql.<database>.host</code>	Database host and port (format: host:port)
<code>jnosql.<database>.user</code>	Username for authentication
<code>jnosql.<database>.password</code>	Password for authentication
<code>jnosql.<database>.timeout</code>	Connection timeout in milliseconds

Where `<type>` is one of: `document`, `keyvalue`, `column`, `graph` and `<database>` is the specific database name.

Adding NoSQL Database Dependencies

To use a specific NoSQL database, add the corresponding Eclipse JNoSQL database dependency. For example:

MongoDB (Document Database):

```
<dependency>
  <groupId>org.eclipse.jnosql.databases</groupId>
  <artifactId>jnosql-mongodb</artifactId>
  <version>{jnosql-version}</version>
</dependency>
```

Redis (Key-Value Database):

```
<dependency>
  <groupId>org.eclipse.jnosql.databases</groupId>
  <artifactId>jnosql-redis</artifactId>
  <version>{jnosql-version}</version>
</dependency>
```

Cassandra (Wide-Column Database):

```
<dependency>
  <groupId>org.eclipse.jnosql.databases</groupId>
  <artifactId>jnosql-cassandra</artifactId>
  <version>{jnosql-version}</version>
</dependency>
```

Neo4j (Graph Database):

```
<dependency>
  <groupId>org.eclipse.jnosql.databases</groupId>
  <artifactId>jnosql-neo4j</artifactId>
  <version>{jnosql-version}</version>
</dependency>
```

Supported NoSQL Databases

For an up-to-date list of supported databases and driver installation instructions, see the [Eclipse JNoSQL Databases](#) repository.

Document Databases: MongoDB, CouchDB, CouchBase, OrientDB, ArangoDB, RavenDB, Elasticsearch, Solr

Key-Value Databases: Redis, Hazelcast, Infinispan, Memcached, Riak, Oracle NoSQL, ArangoDB

Wide-Column Databases: Cassandra, HBase, DynamoDB

Graph Databases: Neo4j, OrientDB, ArangoDB, TinkerPop-compatible databases

Programmatic Configuration

You can configure NoSQL databases programmatically by creating a configuration supplier. The supplier is typically a CDI bean that implements the [Supplier](#) interface. It should be annotated with [@Alternative](#) and [@Priority](#) to override the default configuration.

For example, a supplier for MongoDB:

```
@ApplicationScoped
```

```

@Alternative
@Priority(Interceptor.Priority.APPLICATION)
public class MongoDBManagerSupplier implements Supplier<DocumentManager> {

    @Produces
    public DocumentManager get() {
        Settings settings = Settings.builder()
            .put("jnosql.mongodb.host", "localhost:27017")
            .put("jnosql.document.database", "mystore")
            .build();

        MongoDBDocumentConfiguration configuration = new MongoDBDocumentConfiguration
        (
        );
        DocumentManagerFactory factory = configuration.apply(settings);
        return factory.apply("mystore");
    }
}

```

Transaction Management

Jakarta Data repositories integrate with Jakarta EE transaction management for JPA entities. Repository methods automatically participate in existing transactions or create new ones as needed.



Transaction management is currently only supported for JPA entities. NoSQL repositories do not support JTA transactions yet.

Declarative Transactions

```

import jakarta.ejb.Stateless;
import jakarta.transaction.Transactional;

@ApplicationScoped
public class OrderService {

    @Inject
    private OrderRepository orderRepository;

    @Inject
    private ProductRepository productRepository;

    @Transactional
    public Order createOrder(OrderRequest request) {
        // All repository operations participate in the same transaction
        Product product = productRepository.findById(request.getProductId())
            .orElseThrow(() -> new IllegalArgumentException("Product not found"));

        Order order = new Order();
    }
}

```



```

        order.setProduct(product);
        order.setQuantity(request.getQuantity());
        order.setCustomerId(request.getCustomerId());

        return orderRepository.save(order);
    }
}

```

Custom Transaction Behavior

You can override the default transaction behavior using the `@Transactional` annotation:

```

@Repository
public interface AuditRepository extends CrudRepository<AuditLog, Long> {

    @Transactional(Transactional.TxType.REQUIRES_NEW)
    AuditLog save(AuditLog auditLog);
}

```

By default, repository methods use **REQUIRED** transaction type, which means they join an existing transaction or create a new one if none exists.

Jakarta Validation Support

Jakarta Data repositories integrate with Jakarta Validation to validate method parameters and return values for JPA entities.



Validation is currently only supported for JPA entities. NoSQL repositories do not support Jakarta Validation yet.

Parameter Validation

For simple field validation, use annotations directly on parameters:

```

@Repository
public interface ProductRepository extends CrudRepository<Product, Long> {

    @Find
    List<Product> findProducts(@By("price") @Max(1000) @Min(0) BigDecimal maxPrice);

    @Find
    List<Product> findProducts(@By("category") @NotBlank String category);
}

```

For entities with validation annotations, use `@Valid` on the parameter:

```

@Repository
public interface ProductRepository extends CrudRepository<Product, Long> {

    @Insert
    Product insertProduct(@Valid Product product);

    @Update
    Product updateProduct(@Valid Product product);

    @Save
    Product saveProduct(@Valid Product product);
}

```

Handling Validation Errors

Validation errors are thrown as `ConstraintViolationException`:

```

@ApplicationScoped
public class ProductService {

    private static final System.Logger logger = System.getLogger(ProductService.class
        .getName());

    @Inject
    private ProductRepository productRepository;

    public Product createProduct(Product product) {
        try {
            return productRepository.insertProduct(product);
        } catch (ConstraintViolationException e) {
            Set<ConstraintViolation<?>> violations = e.getConstraintViolations();
            for (ConstraintViolation<?> violation : violations) {
                String property = violation.getPropertyPath().toString();
                String message = violation.getMessage();
                logger.log(System.Logger.Level.ERROR,
                    () -> "Validation error on " + property + ": " + message);
            }
            throw e;
        }
    }
}

```

Limitations and Considerations

General Limitations

Single NoSQL Database Type per Application

Eclipse GlassFish supports only one NoSQL database per application instance.

On the other hand, you can use multiple JPA persistence units in the same application. You can also use both JPA and NoSQL Data repositories simultaneously in the same application.

Repository Interface Restrictions

- Each repository must be dedicated to a single entity type
- For JPA entities, each repository must use a single persistence unit
- You cannot mix JPA and NoSQL entities in the same repository interface

Mixed JPA and NoSQL Entity Usage

You can use both JPA entities and NoSQL entities in the same Eclipse GlassFish application by defining separate repository interfaces for each entity type and configuring the appropriate database connections. However, there are important restrictions:

Separate Repository Interfaces Required You cannot mix JPA entities and NoSQL entities in the same repository interface. Each repository must be dedicated to a single entity type:

```
// JPA repository - works with relational database
@Repository
public interface JpaProductRepository extends CrudRepository<JpaProduct, Long> {
    @Find
    List<JpaProduct> findProducts(@By("category") String category);
}

// NoSQL repository - works with NoSQL database
@Repository
public interface NoSqlProductRepository extends CrudRepository<NoSqlProduct, String> {
    @Find
    List<NoSqlProduct> findProducts(@By("category") String category);
}
```

Separate Database Configurations Configure separate database connections for JPA and NoSQL entities:

- JPA entities use `persistence.xml` and JDBC data sources
- NoSQL entities use MicroProfile Config properties

No Cross-Database Transactions Transactions cannot span across JPA and NoSQL databases. Each database type manages its own transaction context.

JPA Entity Considerations

- Full JTA transaction support
- Jakarta Validation integration

- JPQL and JDQL query support

NoSQL Entity Limitations

Current Limitations:

No Jakarta Validation Support Validation annotations on NoSQL entities and repository parameters are ignored. Implement validation manually:

```
@ApplicationScoped
public class NoSQLProductService {

    private static final System.Logger logger = System.getLogger(NoSQLProductService
.class.getName());

    @Inject
    private NoSQLProductRepository repository;

    public NoSQLProduct createProduct(NoSQLProduct product) {
        // Manual validation
        if (product.getName() == null || product.getName().isBlank()) {
            throw new IllegalArgumentException("Product name is required");
        }
        if (product.getPrice() == null || product.getPrice().compareTo(BigDecimal.
ZERO) <= 0) {
            throw new IllegalArgumentException("Product price must be greater than
zero");
        }

        logger.log(System.Logger.Level.INFO, () -> "Creating product: " + product
.getName());
        return repository.save(product);
    }
}
```

No JTA Transaction Support NoSQL repositories do not participate in JTA transactions. Manage transactions manually if supported by your database:

```
@ApplicationScoped
public class NoSQLOrderService {

    private static final System.Logger logger = System.getLogger(NoSQLOrderService
.class.getName());

    @Inject
    private NoSQLOrderRepository orderRepository;

    @Inject
    private NoSQLCustomerRepository customerRepository;
```

```

public void processOrder(NoSQLOrder order) {
    // No automatic transaction management
    // Implement compensation logic if needed
    try {
        orderRepository.save(order);
        // Update customer separately - no transaction guarantees
        NoSQLCustomer customer = customerRepository.findById(order.getCustomerId(
    ))
        .orElseThrow();
        customer.incrementOrderCount();
        customerRepository.save(customer);

        logger.log(System.Logger.Level.INFO, () -> "Order processed successfully"
    );
    } catch (Exception e) {
        logger.log(System.Logger.Level.ERROR, () -> "Failed to process order: " +
e.getMessage());
        // Implement manual rollback if needed
        throw e;
    }
}
}

```

Single NoSQL Database per Application You can only configure one NoSQL database per application instance.

Common Issues and Troubleshooting

Configuration Issues:

- Verify database drivers are properly installed
- Check MicroProfile Config properties syntax
- Ensure persistence.xml is in the correct location
- Validate JNDI resource names match configuration

Runtime Issues:

- Check database connectivity and credentials
- Verify entity annotations are correct
- Ensure repository interfaces are properly annotated
- Review application logs for detailed error messages

For more detailed troubleshooting, enable debug logging:

```

# Enable Jakarta Data and JNoSQL debug logging
org.eclipse.jnosql.level=FINE

```

```
org.glassfish.main.jnosql.level=FINE  
# Enable EclipseLink (Jakarta Persistence) logging  
org.eclipse.persistence.level=FINE
```

Chapter 13. Developing Web Applications

This chapter describes how web applications are supported in the Eclipse GlassFish.

The following topics are addressed here:

- [Using Servlets](#)
- [Using JavaServer Pages](#)
- [Creating and Managing Sessions](#)
- [Using Comet](#)
- [Advanced Web Application Features](#)

For general information about web applications, see " [The Web Tier](#)" in The Jakarta EE Tutorial.



The Web Profile of the Eclipse GlassFish supports the EJB 3.1 Lite specification, which allows enterprise beans within web applications, among other features. The full Eclipse GlassFish supports the entire EJB 3.1 specification. For details, see [JSR 318](#) (<http://jcp.org/en/jsr/detail?id=318>).

Using Servlets

Eclipse GlassFish supports the Java Servlet Specification version 4.0.



Servlet API version 4.0 is fully backward compatible with versions 3.0, 2.3, 2.4, and 2.5, so all existing servlets should work without modification or recompilation.

To develop servlets, use the Java Servlet API. For information about using the Java Servlet API, see the documentation at <https://jakarta.ee/specifications/servlet/>.

This section describes how to create effective servlets to control application interactions running on a Eclipse GlassFish, including standard-based servlets. In addition, this section describes the Eclipse GlassFish features to use to augment the standards.

The following topics are addressed here:

- [Caching Servlet Results](#)
- [About the Servlet Engine](#)

Caching Servlet Results

The Eclipse GlassFish 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 Eclipse GlassFish caches the request results for a specific amount of time. In this way, if another data call occurs, the Eclipse GlassFish 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 a Eclipse GlassFish web application handles response caching, you edit specific fields in the `glassfish-web.xml` file.



A servlet that uses caching is not portable.

For Javadoc tool pages relevant to caching servlet results, see the `com.sun.appserv.web.cache` package.

For information about JSP caching, see [JSP Caching](#).

The following topics are addressed here:

- [Caching Features](#)
- [Default Cache Configuration](#)
- [Caching Example](#)
- [The CacheKeyGenerator Interface](#)

Caching Features

The Eclipse GlassFish 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](#).
- 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 Eclipse GlassFish Application Deployment Guide and "dispatcher" in Eclipse GlassFish Application Deployment Guide. These are elements in the `glassfish-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 `glassfish-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'
    -->
  </cache-mapping>
</cache>
```

```

-- 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 `glassfish-web.xml` caching settings, see ["cache"](#) in Eclipse GlassFish 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 `glassfish-web.xml` deployment descriptor. For more information, see ["default-helper"](#) in Eclipse GlassFish Application Deployment Guide.

About the Servlet Engine

Servlets exist in and are managed by the servlet engine in the Eclipse GlassFish. 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.

The following topics are addressed here:

- [Instantiating and Removing Servlets](#)
- [Request Handling](#)

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 Eclipse GlassFish 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 Eclipse GlassFish 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 the JavaServer Pages web site at <https://jakarta.ee/specifications/pages/>.

This section describes how to use JavaServer Pages (JSP files) as page templates in a Eclipse GlassFish web application.

The following topics are addressed here:

- [JSP Tag Libraries and Standard Portable Tags](#)
- [JSP Caching](#)
- [Options for Compiling JSP Files](#)

JSP Tag Libraries and Standard Portable Tags

Eclipse GlassFish supports tag libraries and standard portable tags. For more information, see the JavaServer Pages Standard Tag Library (JSTL) page at <https://jakarta.ee/specifications/tags/>.

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/jspcachtags.jar` tag library for JSP caching is not automatically available to web applications. See [JSP Caching](#), 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.

The following topics are addressed here:

- [Enabling JSP Caching](#)
- [Caching Scope](#)
- [The `cache` Tag](#)
- [The `flush` Tag](#)

For more information about response caching as it pertains to servlets, see [Caching Servlet Results](#).

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 [Eclipse GlassFish Reference Manual](#).

To enable JSP caching for a single web application, follow these steps:

1. Extract the `META-INF/jspcachtags.tld` file from the `as-install/modules/web-glue.jar` file.
2. Create a new JAR file (for example, `jspcachtags.jar`) containing just the `META-INF/jspcachtags.tld` file previously extracted.

3. Bundle this new JAR file in the **WEB-INF/lib** directory of your web application.



Web applications that use JSP caching without bundling the tag library are not portable.

Refer to Eclipse GlassFish tags in JSP files as follows:

```
<%@ taglib prefix="prefix" uri="http://glassfish.org/taglibs/cache" %>
```

Subsequently, the cache tags are available as `<'prefix:cache'>` and `<'prefix:flush'>`. For example, if your prefix is `myafx`, the cache tags are available as `<myafx:cache>` and `<myafx: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 `jspcachtags.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 7-1 The `cache` Attributes

Attribute	Default	Description
<code>key</code>	<code>ServletPath`_` Suffix</code>	(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.
<code>timeout</code>	<code>60s</code>	(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: <code>s</code> for seconds, <code>m</code> for minutes, <code>h</code> for hours, <code>d</code> for days. For example, <code>2h</code> specifies two hours.
<code>nocache</code>	<code>false</code>	(optional) If set to <code>true</code> , the body content is executed and served as if there were no <code>cache</code> 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.
<code>refresh</code>	<code>false</code>	(optional) If set to <code>true</code> , the body content is executed and the response is cached again. This lets you programmatically refresh the cache immediately regardless of the <code>timeout</code> setting.
<code>scope</code>	<code>application</code>	(optional) The scope of the cache. Can be <code>request</code> , <code>session</code> , or <code>application</code> . See Caching Scope .

Example of `cache`

The following example represents a cached JSP file:

```
<%@ taglib prefix="mypfx" uri="http://glassfish.org/taglibs/cache" %>
<%@ 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 7-2 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 .

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

Eclipse GlassFish 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 [Eclipse GlassFish Reference Manual](#).
- The **jspc** command line tool in **glassfish/bin** allows you to precompile JSP files at the command line. For details, run **glassfish/bin/jspc -help**

Creating and Managing Sessions

This section describes how to create and manage HTTP sessions that allows users and transaction information to persist between interactions.

The following topics are addressed here:

- [Configuring Sessions](#)
- [Session Managers](#)

Configuring Sessions

The following topics are addressed here:

- [HTTP Sessions, Cookies, and URL Rewriting](#)
- [Coordinating Session Access](#)
- [Saving Sessions During Redeployment](#)
- [Logging Session Attributes](#)
- [Distributed Sessions and Persistence](#)

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 `glassfish-web.xml` file for an individual web application. For more about the properties you can configure, see "[session-properties](#)" in Eclipse GlassFish Application Deployment Guide and "[cookie-properties](#)" in Eclipse GlassFish Application Deployment Guide.

For information about configuring default session properties for the entire web container, see [Using the default-web.xml File](#) and the [Eclipse GlassFish High Availability Administration Guide](#).

Coordinating Session Access

Make sure that multiple threads don't simultaneously modify the same session object in conflicting ways. If the persistence type is `replicated` (see [The replicated Persistence Type](#)), the following message in the log file indicates that this might be happening:

```
Primary Key Constraint violation while saving session session_id
```

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 `--keepstate=true` option of the `asadmin redeploy` command. For example:


```
asadmin redeploy --keepstate=true --name hello.war
```

For details, see the [Eclipse GlassFish Reference Manual](#).

The default for `--keepstate` is false. This option is supported only on the default server instance, named `server`. It is not supported and ignored for any other target.

For web applications, this feature is applicable only if in the `glassfish-web-app.xml` file the `persistence-type` attribute of the `session-manager` element is `file`.

If any active web session fails to be preserved or restored, none of the sessions will be available when the redeployment is complete. However, the redeployment continues and a warning is logged.

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 [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.

Distributed Sessions and Persistence

A distributed HTTP session can run in multiple Eclipse GlassFish instances, provided the following criteria are met:

- Each server instance has the same distributable web application deployed to it. The `web-app` element of the `web.xml` deployment descriptor file must have the `distributable` subelement specified.
- The web application uses high-availability session persistence. If a non-distributable web application is configured to use high-availability session persistence, a warning is written to the server log, and the session persistence type reverts to `memory`. See [The replicated Persistence Type](#).
- All objects bound into a distributed session must be of the types listed in [Table 7-3](#).
- The web application must be deployed using the `deploy` or `deploydir` command with the `--availabilityenabled` option set to `true`. See the [Eclipse GlassFish Reference Manual](#).



Contrary to the Servlet 5.0 specification, Eclipse GlassFish does not throw an `IllegalArgumentException` if an object type not supported for failover is bound into a distributed session.

Keep the distributed session size as small as possible. Session size has a direct impact on overall system throughput.

In the event of an instance or hardware failure, another server instance can take over a distributed session, with the following limitations:

- If a distributable web application references a Jakarta EE component or resource, the reference might be lost. See [Table 7-3](#) for a list of the types of references that `HTTPSession` failover supports.
- References to open files or network connections are lost.

For information about how to work around these limitations, see the [Eclipse GlassFish Deployment Planning Guide](#).

In the following table, No indicates that failover for the object type might not work in all cases and that no failover support is provided. However, failover might work in some cases for that object type. For example, failover might work because the class implementing that type is serializable.

For more information about the `InitialContext`, see [Accessing the Naming Context](#). For more information about transaction recovery, see [Using the Transaction Service](#). For more information about Administered Objects, see "[Administering JMS Physical Destinations](#)" in Eclipse GlassFish Administration Guide.

Table 7-3 Object Types Supported for Jakarta EE Web Application Session State Failover

Java Object Type	Failover Support
Colocated or distributed stateless session, stateful session, or entity bean reference	Yes
JNDI context	Yes, <code>InitialContext</code> and <code>java:comp/env</code>
UserTransaction	Yes, but if the instance that fails is never restarted, any prepared global transactions are lost and might not be correctly rolled back or committed.
JDBC DataSource	No
Java Message Service (JMS) ConnectionFactory, Destination	No
Jakarta Mail Session	No
Connection Factory	No
Administered Object	No
Web service reference	No
Serializable Java types	Yes

Java Object Type	Failover Support
Extended persistence context	No

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.

Eclipse GlassFish offers these session management options, determined by the `session-manager` element's `persistence-type` attribute in the `glassfish-web.xml` file:

- The `memory Persistence Type`, the default
- The `file Persistence Type`, which uses a file to store session data
- The `replicated Persistence Type`, which uses other servers in the cluster for session persistence



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 Eclipse GlassFish 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 `glassfish-web.xml` file as in the following example. The `persistence-type` attribute is optional, but must be set to `memory` if included. This overrides the web container availability settings for the web application.

```
<glassfish-web-app>
...

<session-config>
  <session-manager persistence-type="memory" />
  <manager-properties>
    <property name="sessionFilename" value="sessionstate" />
  </manager-properties>
</session-manager>
...
</session-config>
...
</glassfish-web-app>
```

The only manager property that the `memory` persistence type supports is `sessionFilename`, which is listed under "`manager-properties`" in Eclipse GlassFish 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`. The default value is an empty string.

For more information about the `glassfish-web.xml` file, see the [Eclipse GlassFish 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.



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](#).

To specify the `file` persistence type for a specific web application, edit the `glassfish-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.

```
<glassfish-web-app>
...
<session-config>
  <session-manager persistence-type="file">
    <store-properties>
      <property name="directory" value="sessiondir" />
    </store-properties>
  </session-manager>
  ...
</session-config>
...
</glassfish-web-app>
```

The `file` persistence type supports all the manager properties listed under "`manager-properties`" in Eclipse GlassFish Application Deployment Guide except `sessionFilename`, and supports the `directory` store property listed under "`store-properties`" in Eclipse GlassFish Application Deployment Guide.

For more information about the `glassfish-web.xml` file, see the [Eclipse GlassFish Application Deployment Guide](#).

The `replicated` Persistence Type

The replicated persistence type uses other servers in the cluster for session persistence. Clustered server instances replicate session state. Each backup instance stores the replicated data in memory. This allows sessions to be distributed. For details, see [Distributed Sessions and Persistence](#). In

addition, you can configure the frequency and scope of session persistence. The other servers are also used as the passivation and activation store. Use this option in a production environment that requires session persistence.

To use the replicated persistence type, you must enable availability. Select the Availability Service component under the relevant configuration in the Administration Console. Check the Availability Service box. To enable availability for the web container, select the Web Container Availability tab, then check the Availability Service box. All instances in an Eclipse GlassFish cluster should have the same availability settings to ensure consistent behavior. For details, see the [Eclipse GlassFish High Availability Administration Guide](#).

To change settings such as persistence frequency and persistence scope for the entire web container, use the Persistence Frequency and Persistence Scope drop-down lists on the Web Container Availability tab in the Administration Console, or use the `asadmin set` command. For example:

```
asadmin set
server-config.availability-service.web-container-availability.persistence-
frequency=time-based
```

For more information, see the description of the `asadmin set` command in the [Eclipse GlassFish Reference Manual](#).

To specify the `replicated` persistence type for a specific web application, edit the `glassfish-web.xml` file as in the following example. Note that `persistence-type` must be set to `replicated`. This overrides the web container availability settings for the web application.

```
<glassfish-web-app>
...
<session-config>
  <session-manager persistence-type="replicated">
    <manager-properties>
      <property name="persistenceFrequency" value="web-method" />
    </manager-properties>
    <store-properties>
      <property name="persistenceScope" value="session" />
    </store-properties>
  </session-manager>
  ...
</session-config>
...
</glassfish-web-app>
```

The `replicated` persistence type supports all the manager properties listed under "`manager-properties`" in Eclipse GlassFish Application Deployment Guide except `sessionFilename`, and supports the `persistenceScope` store property listed under "`store-properties`" in Eclipse GlassFish Application Deployment Guide.

For more information about the `glassfish-web.xml` file, see the [Eclipse GlassFish Application Deployment Guide](#).

To specify that web sessions for which high availability is enabled are first buffered and then replicated using a separate asynchronous thread, use the `--asyncreplication=true` option of the `asadmin deploy` command. For example:

```
asadmin deploy --availabilityenabled=true --asyncreplication=true --name hello.war
```

If `--asyncreplication` is set to true (the default), performance is improved but availability is reduced. If the instance where states are buffered but not yet replicated fails, the states are lost. If set to false, performance is reduced but availability is guaranteed. States are not buffered but immediately transmitted to other instances in the cluster.

Using Comet

This section explains the Comet programming technique and how to create and deploy a Comet-enabled application with the Eclipse GlassFish.

The following topics are addressed here:

- [Introduction to Comet](#)
- [Grizzly Comet](#)
- [Bayeux Protocol](#)

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 Eclipse GlassFish 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** — 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 Eclipse GlassFish.
- **Bayeux Protocol** — 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** (<http://dojotoolkit.org>).

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

For details on using Grizzly Comet including a sample application, refer to the [Grizzly Comet](#)

[documentation](#).

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))
```

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 Eclipse GlassFish that support the Bayeux protocol. Just enable Comet and the Bayeux protocol, then write and deploy the client.

The following topics are addressed here:

- [Enabling Comet](#)
- [To Configure the web.xml File](#)
- [To Write, Deploy, and Run the Client](#)

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="true"
```

Substitute the name of the protocol for `http-1`.

To Configure the `web.xml` File

To enable the Bayeux protocol on the Eclipse GlassFish, 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

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 Eclipse GlassFish, see the [Eclipse GlassFish Application Deployment Guide](#).

For more information about the Bayeux protocol, see [Bayeux Protocol](https://docs.cometd.org/current/reference/#_bayeux) (https://docs.cometd.org/current/reference/#_bayeux).

For more information about the Dojo toolkit, see <http://dojotoolkit.org/>.

Advanced Web Application Features

The following topics are addressed here:

- [Internationalization Issues](#)
- [Virtual Server Properties](#)

- [Class Loader Delegation](#)
- [Using the `default-web.xml` File](#)
- [Configuring Logging and Monitoring in the Web Container](#)
- [Configuring Idempotent URL Requests](#)
- [Header Management](#)
- [Configuring Valves and Catalina Listeners](#)
- [Alternate Document Roots](#)
- [Using a `context.xml` File](#)
- [Enabling WebDav](#)
- [Using SSI](#)
- [Using CGI](#)

Internationalization Issues

The following topics are addressed here:

- [The Server's Default Locale](#)
- [Servlet Character Encoding](#)

The Server's Default Locale

To set the default locale of the entire Eclipse GlassFish, which determines the locale of the Administration Console, the logs, and so on, use the Administration Console. Select the domain component. 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 Eclipse GlassFish determines the character encoding for the servlet request and the servlet response. For encodings you can use, see <http://docs.oracle.com/javase/8/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 `glassfish-web.xml` file
- The `default-charset` attribute of the `parameter-encoding` element in the `glassfish-web.xml` file
- The default, which is `ISO-8859-1`

For details about the `parameter-encoding` element, see "[parameter-encoding](#)" in Eclipse GlassFish

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 "[glassfish-web-app](#)" in Eclipse GlassFish 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 `glassfish-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 `glassfish-web.xml`, see the [Eclipse GlassFish 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 Jakarta EE platform.

For general information about class loaders, see [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. Select Logger Settings under the relevant configuration, or select the Stand-Alone Instances component, select the instance from the table, and select the Monitor tab.

Configuring Idempotent URL Requests

An idempotent request is one that does not cause any change or inconsistency in an application when retried. To enhance the availability of your applications deployed on an Eclipse GlassFish cluster, configure the load balancer to retry failed idempotent HTTP requests on all the Eclipse GlassFish instances in a cluster. This option can be used for read-only requests, for example, to retry a search request.

The following topics are addressed here:

- [Specifying an Idempotent URL](#)
- [Characteristics of an Idempotent URL](#)

Specifying an Idempotent URL

To configure idempotent URL response, specify the URLs that can be safely retried in `idempotent-url-pattern` elements in the `glassfish-web.xml` file. For example:

```
<idempotent-url-pattern url-pattern="sun_java/*" no-of-retries="10"/>
```

For details, see "[idempotent-url-pattern](#)" in Eclipse GlassFish Application Deployment Guide.

If none of the server instances can successfully serve the request, an error page is returned.

Characteristics of an Idempotent URL

Since all requests for a given session are sent to the same application server instance, and if that Eclipse GlassFish instance is unreachable, the load balancer returns an error message. Normally, the request is not retried on another Eclipse GlassFish instance. However, if the URL pattern matches that specified in the `glassfish-web.xml` file, the request is implicitly retried on another Eclipse GlassFish instance in the cluster.

In HTTP, some methods (such as GET) are idempotent, while other methods (such as POST) are not. In effect, retrying an idempotent URL should not cause values to change on the server or in the

database. The only difference should be a change in the response received by the user.

Examples of idempotent requests include search engine queries and database queries. The underlying principle is that the retry does not cause an update or modification of data.

A search engine, for example, sends HTTP requests with the same URL pattern to the load balancer. Specifying the URL pattern of the search request to the load balancer ensures that HTTP requests with the specified URL pattern are implicitly retried on another Eclipse GlassFish instance.

For example, if the request URL sent to the Eclipse GlassFish is of the type `/search/something.html`, then the URL pattern can be specified as `/search/*`.

Examples of non-idempotent requests include banking transactions and online shopping. If you retry such requests, money might be transferred twice from your account.

Header Management

In all Editions of the Eclipse GlassFish, 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.addXXXHeader()` and `HttpServletResponse.setXXXHeader()` 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 Eclipse GlassFish releases, or the `org.glassfish.web.valve.GlassFishValve` interface from the current Eclipse GlassFish 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 `glassfish-web.xml` file, valve and listener properties for a web module look like this:

```
<glassfish-web-app ...>
...
  <property name="valve_1" value="org.glassfish.extension.Valve"/>
  <property name="listener_1" value="org.glassfish.extension.MyLifecycleListener"/>
</glassfish-web-app>
```

You can define these same properties for a virtual server. For more information, see [Virtual Server Properties](#).

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 `glassfish-web-app` element in the `glassfish-web.xml` file or a virtual server property. For more information about these elements, see "`glassfish-web-app`" in Eclipse GlassFish Application Deployment Guide.

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 `glassfish-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:

```
/srv/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:

```
/srv/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](#). For more information about the `context.xml` file, see [The Context Container \(http://tomcat.apache.org/tomcat-5.5-doc/config/context.html\)](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 Eclipse GlassFish.

Enabling WebDav

To enable WebDav in the Eclipse GlassFish, you edit the `web.xml` and `glassfish-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 `glassfish-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](#).

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`, Eclipse GlassFish 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 7-4 `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.

init-param	Type	Default	Description
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](#).

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 7-5 `CGIServlet` `init-param` Values

<code>init-param</code>	Type	Default	Description
<code>cgiPathPrefix</code>	<code>String</code>	<code>WEB-INF/cgi</code>	Specifies the subdirectory containing the CGI programs.
<code>debug</code>	<code>int</code>	<code>0</code> (for no debugging)	Specifies the debugging level.
<code>executable</code>	<code>String</code>	<code>perl</code>	Specifies the executable for running the CGI script.
<code>parameterEncoding</code>	<code>String</code>	<code>System.getProperty("file.encoding", "UTF-8")</code>	Specifies the parameter's encoding.
<code>passShellEnvironment</code>	<code>boolean</code>	<code>false</code>	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 [Eclipse GlassFish Application Deployment Guide](#).

Chapter 14. Using Enterprise JavaBeans Technology

This chapter describes how Enterprise JavaBeans (EJB) technology is supported in the Eclipse GlassFish.

The following topics are addressed here:

- [Value Added Features](#)
- [EJB Timer Service](#)
- [Using Session Beans](#)
- [Using Read-Only Beans](#)
- [Using Message-Driven Beans](#)

For general information about enterprise beans, see [Enterprise Beans](#) in The Jakarta EE Tutorial.



The Web Profile of the Eclipse GlassFish supports the EJB 3.1 Lite specification, which allows enterprise beans within web applications, among other features. The full Eclipse GlassFish supports the entire EJB 3.1 specification. For details, see [JSR 318](#) (<http://jcp.org/en/jsr/detail?id=318>).

The Eclipse GlassFish 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 Eclipse GlassFish provides a number of value additions that relate to EJB development. References to more in-depth material are included.

The following topics are addressed here:

- [Read-Only Beans](#)
- [The pass-by-reference Element](#)
- [Pooling and Caching](#)
- [Priority Based Scheduling of Remote Bean Invocations](#)
- [Immediate Flushing](#)

Read-Only Beans

Another feature that the Eclipse GlassFish 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.



Read-only beans are specific to the Eclipse GlassFish 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 Eclipse GlassFish 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 `glassfish-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](#).

The `pass-by-reference` Element

The `pass-by-reference` element in the `glassfish-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 Eclipse GlassFish Application Deployment Guide.

Pooling and Caching

The EJB container of the Eclipse GlassFish 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 `glassfish-ejb-jar.xml` file.

In addition, the Eclipse GlassFish 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 `glassfish-ejb-jar.xml` file.

Pooling and caching parameters for the `glassfish-ejb-jar.xml` file are described in "bean-cache" in Eclipse GlassFish Application Deployment Guide.

Pooling Parameters

One of the most important parameters for Eclipse GlassFish pooling is `steady-pool-size`. When `steady-pool-size` is set to a value 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.

Note that the `steady-pool-size` and `max-pool-size` parameters only govern the number of instances that are pooled over a long period of time. They do not necessarily guarantee that the number of instances that may exist in the JVM at a given time will not exceed the value specified by `max-pool-size`. 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

Eclipse GlassFish 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](#).

The Eclipse GlassFish 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).

Priority Based Scheduling of Remote Bean Invocations

You can create multiple thread pools, each having its own work queues. An optional element in the `glassfish-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 `glassfish-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 Eclipse GlassFish 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 Eclipse GlassFish is preconfigured to use an embedded version of the Apache Derby database.

The EJB Timer Service configuration can store persistent timer information in any database supported by the Eclipse GlassFish for persistence. For a list of the JDBC drivers currently supported by the Eclipse GlassFish, see the [Eclipse GlassFish Release Notes](#). For configurations of supported and other drivers, see "[Configuration Specifics for JDBC Drivers](#)" in Eclipse GlassFish Administration Guide.

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
```

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 in a server instance, 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 after an exception or rollback of a container-managed transaction. 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`.



Do not use the `jdbc/__TimerPool` resource for timers in clustered Eclipse GlassFish environments. You must instead use a custom JDBC resource or the `jdbc/__default` resource. See the instructions below, in [To Deploy an EJB Timer to a Cluster](#). Also refer to "[Enabling the jdbc/__default Resource in a Clustered Environment](#)" in Eclipse GlassFish Administration Guide.

For information about the `asadmin list-timers` and `asadmin migrate-timers` subcommands, see the [Eclipse GlassFish Reference Manual](#). For information about migrating EJB timers, see "[Migrating EJB Timers](#)" in Eclipse GlassFish High Availability Administration Guide.

You can use the `--keepstate` option of the `asadmin redeploy` command to retain EJB timers between redeployments.

The default for `--keepstate` is false. This option is supported only on the default server instance, named `server`. It is not supported and ignored for any other target.

When the `--keepstate` is set to true, each application that uses an EJB timer is assigned an ID in the timer database. The EJB object that is associated with a given application is assigned an ID that is constructed from the application ID and a numerical suffix. To preserve active timer data, Eclipse GlassFish stores the application ID and the EJB ID in the timer database. To restore the data, the class loader of the newly redeployed application retrieves the EJB timers that correspond to these IDs from the timer database.

For more information about the `asadmin redeploy` command, see the [Eclipse GlassFish Reference Manual](#).

To Deploy an EJB Timer to a Cluster

This procedure explains how to deploy an EJB timer to a cluster.

By default, the Eclipse GlassFish 8 timer service points to the preconfigured `jdbc/__TimerPool` resource, which uses an embedded Apache Derby database configuration that will not work in clustered environments.

The problem is that embedded Apache Derby database runs in the Eclipse GlassFish Java VM, so when you use the `jdbc/__TimerPool` resource, each DAS and each clustered server instance will have its own database table. Because of this, clustered server instances will not be able to find the database table on the DAS, and the DAS will not be able to find the tables on the clustered server instances.

The solution is to use either a custom JDBC resource or the `jdbc/default` resource that is preconfigured but not enabled by default in Eclipse GlassFish. The `jdbc/default` resource does not use the embedded Apache Derby database by default.

Before You Begin

If creating a new timer resource, the resource should be created before deploying applications that will use the timer.



Do not use the `jdbc/__TimerPool` resource for timers in clustered Eclipse GlassFish environments. You must instead use a custom JDBC resource or the `jdbc/__default` resource. See "[Enabling the jdbc/__default Resource in a Clustered Environment](#)" in Eclipse GlassFish Administration Guide.

1. Execute the following command:

```
asadmin set configs.config.cluster_name-config.ejb-container.ejb-timer-  
service.timer-  
datasource=jdbc/my-timer-resource
```

2. Restart the DAS and the target cluster(s).

```
asadmin stop-cluster cluster-name  
asadmin stop-domain domain-name  
asadmin start-domain domain-name  
asadmin start-cluster cluster-name
```

Troubleshooting

If you inadvertently used the `jdbc/__TimerPool` resource for your EJB timer in a clustered Eclipse GlassFish environment, the DAS and the clustered server instances will be using separate Apache Derby database tables that are running in individual Java VMs. For timers to work in a clustered environment, the DAS and the clustered server instances must share a common database table.

If you attempt to deploy an application with EJB timers without setting the timer resource correctly, the startup will fail, and you will be left with a marker file, named `ejb-timer-service-app`, on the DAS that will prevent the Timer Service from correctly creating the database table.

The solution is to remove the marker file on the DAS, restart the DAS and the clusters, and then redploy any applications that rely on the offending EJB timer. The marker file is located on the DAS in domain-dir `/generated/ejb/`ejb-timer-service-app`.

Using Session Beans

This section provides guidelines for creating session beans in the Eclipse GlassFish environment.

The following topics are addressed here:

- [About the Session Bean Containers](#)
- [Stateful Session Bean Failover](#)
- [Session Bean Restrictions and Optimizations](#)

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.

The following topics are addressed here:

- [Stateless Container](#)
- [Stateful Container](#)

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 Eclipse GlassFish specific deployment descriptor file, `glassfish-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 `glassfish-ejb-jar.xml`, see "[The glassfish-ejb-jar.xml File](#)" in Eclipse GlassFish Application Deployment Guide.

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 `glassfish-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 `glassfish-ejb-jar.xml`, see "[The glassfish-ejb-jar.xml File](#)" in Eclipse GlassFish 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``.



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](#).

The Session Store Location setting also determines where the session state is persisted if it is not highly available; see [Choosing a Persistence Store](#).

Stateful Session Bean Failover

An SFSB's state can be saved in a persistent store in case a server instance fails. The state of an SFSB is saved to the persistent store at predefined points in its life cycle. This is called checkpointing. If SFSB checkpointing is enabled, checkpointing generally occurs after any transaction involving the SFSB is completed, even if the transaction rolls back.

However, if an SFSB participates in a bean-managed transaction, the transaction might be committed in the middle of the execution of a bean method. Since the bean's state might be

undergoing transition as a result of the method invocation, this is not an appropriate instant to checkpoint the bean's state. In this case, the EJB container checkpoints the bean's state at the end of the corresponding method, provided the bean is not in the scope of another transaction when that method ends. If a bean-managed transaction spans across multiple methods, checkpointing is delayed until there is no active transaction at the end of a subsequent method.

The state of an SFSB is not necessarily transactional and might be significantly modified as a result of non-transactional business methods. If this is the case for an SFSB, you can specify a list of checkpointed methods. If SFSB checkpointing is enabled, checkpointing occurs after any checkpointed methods are completed.

The following table lists the types of references that SFSB failover supports. All objects bound into an SFSB must be one of the supported types. In the table, No indicates that failover for the object type might not work in all cases and that no failover support is provided. However, failover might work in some cases for that object type. For example, failover might work because the class implementing that type is serializable.

Table 8-1 Object Types Supported for Jakarta EE Stateful Session Bean State Failover

Java Object Type	Failover Support
Colocated or distributed stateless session, stateful session, or entity bean reference	Yes
JNDI context	Yes, <code>InitialContext</code> and <code>java:comp/env</code>
UserTransaction	Yes, but if the instance that fails is never restarted, any prepared global transactions are lost and might not be correctly rolled back or committed.
JDBC DataSource	No
Java Message Service (JMS) ConnectionFactory, Destination	No
Jakarta Mail Session	No
Connection Factory	No
Administered Object	No
Web service reference	No
Serializable Java types	Yes
Extended persistence context	No

For more information about the `InitialContext`, see [Accessing the Naming Context](#). For more information about transaction recovery, see [Using the Transaction Service](#). For more information about Administered Objects, see "[Administering JMS Physical Destinations](#)" in Eclipse GlassFish Administration Guide.



Idempotent URLs are supported along the HTTP path, but not the RMI-IIOP path. For more information, see [Configuring Idempotent URL Requests](#).

If a server instance to which an RMI-IIOP client request is sent crashes during the request processing (before the response is prepared and sent back to the client), an error is sent to the client. The client must retry the request explicitly. When the client retries the request, the request is sent to another server instance in the cluster, which retrieves session state information for this client.

HTTP sessions can also be saved in a persistent store in case a server instance fails. In addition, if a distributable web application references an SFSB, and the web application's session fails over, the EJB reference is also failed over. For more information, see [Distributed Sessions and Persistence](#).

If an SFSB that uses session persistence is undeployed while the Eclipse GlassFish instance is stopped, the session data in the persistence store might not be cleared. To prevent this, undeploy the SFSB while the Eclipse GlassFish instance is running.

Configure SFSB failover by:

- [Choosing a Persistence Store](#)
- [Enabling Checkpointing](#)
- [Specifying Methods to Be Checkpointed](#)

Choosing a Persistence Store

The following types of persistent storage are supported for passivation and checkpointing of the SFSB state:

- The local file system - Allows a single server instance to recover the SFSB state after a failure and restart. This store also provides passivation and activation of the state to help control the amount of memory used. This option is not supported in a production environment that requires SFSB state persistence. This is the default storage mechanism if availability is not enabled.
- Other servers - Uses other server instances in the cluster for session persistence. Clustered server instances replicate session state. Each backup instance stores the replicated data in memory. This is the default storage mechanism if availability is enabled.

Choose the persistence store in one of the following ways:

- To use the local file system, first disable availability. Select the Availability Service component under the relevant configuration in the Administration Console. Uncheck the Availability Service box. Then select the EJB Container component and edit the Session Store Location value. The default is `domain-dir`/session-store``.
- To use other servers, select the Availability Service component under the relevant configuration in the Administration Console. Check the Availability Service box. To enable availability for the EJB container, select the EJB Container Availability tab, then check the Availability Service box. All instances in an Eclipse GlassFish cluster should have the same availability settings to ensure consistent behavior.

For more information about SFSB state persistence, see the [Eclipse GlassFish High Availability](#)

Using the `--keepstate` Option

If you are using the file system for persistence, you can use the `--keepstate` option of the `asadmin redeploy` command to retain the SFSB state between redeployments.

The default for `--keepstate` is false. This option is supported only on the default server instance, named `server`. It is not supported and ignored for any other target.

Some changes to an application between redeployments prevent this feature from working properly. For example, do not change the set of instance variables in the SFSB bean class.

If any active SFSB instance fails to be preserved or restored, none of the SFSB instances will be available when the redeployment is complete. However, the redeployment continues and a warning is logged.

To preserve active state data, Eclipse GlassFish serializes the data and saves it in memory. To restore the data, the class loader of the newly redeployed application deserializes the data that was previously saved.

For more information about the `asadmin redeploy` command, see the [Eclipse GlassFish Reference Manual](#).

Using the `--asyncreplication` Option

If you are using replication on other servers for persistence, you can use the `--asyncreplication` option of the `asadmin deploy` command to specify that SFSB states are first buffered and then replicated using a separate asynchronous thread. If `--asyncreplication` is set to true (default), performance is improved but availability is reduced. If the instance where states are buffered but not yet replicated fails, the states are lost. If set to false, performance is reduced but availability is guaranteed. States are not buffered but immediately transmitted to other instances in the cluster.

For more information about the `asadmin deploy` command, see the [Eclipse GlassFish Reference Manual](#).

Enabling Checkpointing

The following sections describe how to enable SFSB checkpointing:

- [Server Instance and EJB Container Levels](#)
- [Application and EJB Module Levels](#)
- [SFSB Level](#)

Server Instance and EJB Container Levels

To enable SFSB checkpointing at the server instance or EJB container level, see [Choosing a Persistence Store](#).

Application and EJB Module Levels

To enable SFSB checkpointing at the application or EJB module level during deployment, use the `asadmin deploy` or `asadmin deploydir` command with the `--availabilityenabled` option set to `true`. For details, see the [Eclipse GlassFish Reference Manual](#).

SFSB Level

To enable SFSB checkpointing at the SFSB level, set `availability-enabled="true"` in the `ejb` element of the SFSB's `glassfish-ejb-jar.xml` file as follows:

```
<glassfish-ejb-jar>
...
<enterprise-beans>
...
  <ejb availability-enabled="true">
    <ejb-name>MySFSB</ejb-name>
  </ejb>
...
</enterprise-beans>
</glassfish-ejb-jar>
```

Specifying Methods to Be Checkpointed

If SFSB checkpointing is enabled, checkpointing generally occurs after any transaction involving the SFSB is completed, even if the transaction rolls back.

To specify additional optional checkpointing of SFSBs at the end of non-transactional business methods that cause important modifications to the bean's state, use the `checkpoint-at-end-of-method` element within the `ejb` element in `glassfish-ejb-jar.xml`.

For example:

```
<glassfish-ejb-jar>
...
<enterprise-beans>
...
  <ejb availability-enabled="true">
    <ejb-name>ShoppingCartEJB</ejb-name>
    <checkpoint-at-end-of-method>
      <method>
        <method-name>addToCart</method-name>
      </method>
    </checkpoint-at-end-of-method>
  </ejb>
...
</enterprise-beans>
</glassfish-ejb-jar>
```

For details, see "[checkpoint-at-end-of-method](#)" in Eclipse GlassFish Application Deployment Guide.

The non-transactional methods in the `checkpoint-at-end-of-method` element can be the following:

- `create` methods defined in the home or business interface of the SFSB, if you want to checkpoint the initial state of the SFSB immediately after creation
- For SFSBs using container managed transactions only, methods in the remote interface of the bean marked with the transaction attribute `TX_NOT_SUPPORTED` or `TX_NEVER`
- For SFSBs using bean managed transactions only, methods in which a bean managed transaction is neither started nor committed

Any other methods mentioned in this list are ignored. At the end of invocation of each of these methods, the EJB container saves the state of the SFSB to persistent store.



If an SFSB does not participate in any transaction, and if none of its methods are explicitly specified in the `checkpoint-at-end-of-method` element, the bean's state is not checkpointed at all even if `availability-enabled="true"` for this bean.

For better performance, specify a small subset of methods. The methods chosen should accomplish a significant amount of work in the context of the Jakarta EE application or should result in some important modification to the bean's state.

Session Bean Restrictions and Optimizations

This section discusses restrictions on developing session beans and provides some optimization guidelines.

- [Optimizing Session Bean Performance](#)
- [Restricting Transactions](#)
- [EJB Singletons](#)

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.

EJB Singletons

EJB Singletons are created for each server instance in a cluster, and not once per cluster.

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.



Read-only beans are specific to the Eclipse GlassFish 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 here:

- [Read-Only Bean Characteristics and Life Cycle](#)
- [Read-Only Bean Good Practices](#)
- [Refreshing Read-Only Beans](#)
- [Deploying Read-Only Beans](#)

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 [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 `glassfish-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](#)
- [Refreshing Periodically](#)
- [Refreshing Programmatically](#)

Invoking a Transactional Method

Invoking any transactional method invokes `ejbLoad`.

Refreshing Periodically

Use the `refresh-period-in-seconds` element in the `glassfish-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.



This is the only way to refresh the bean state if the data can be modified external to the Eclipse GlassFish.

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 <http://glassfish.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 `glassfish-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 `glassfish-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 glassfish-ejb-jar.xml File](#)" in Eclipse GlassFish Application Deployment Guide.

Using Message-Driven Beans

This section describes message-driven beans and explains the requirements for creating them in the Eclipse GlassFish environment.

The following topics are addressed here:

- [Message-Driven Bean Configuration](#)
- [Message-Driven Bean Restrictions and Optimizations](#)

Message-Driven Bean Configuration

The following topics are addressed here:

- [Connection Factory and Destination](#)
- [Message-Driven Bean Pool](#)
- [Domain-Level Settings](#)

For information about setting up load balancing for message-driven beans, see [Load-Balanced Message Inflow](#).

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 Eclipse GlassFish 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 `glassfish-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 `glassfish-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 `glassfish-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 `glassfish-ejb-jar.xml`, see "[The glassfish-ejb-jar.xml File](#)" in Eclipse GlassFish 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 `2`.

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. Select the Stand-Alone Instances component, select the instance from the table, and select the Monitor tab. Or select the Clusters component, select the cluster from the table, select the Instances tab, select the instance from the table, and select the Monitor tab.



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 "[Administering the Monitoring Service](#)" in Eclipse GlassFish 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 `onMessage` Runtime Exception](#)

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 `glassfish-ejb-jar.xml` file). For example, you can tune the Open 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 "[Administering the Monitoring Service](#)" in Eclipse GlassFish 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 Eclipse GlassFish 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 [Eclipse GlassFish 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 `jakarta.jms.Message.getJMSRedelivered` method to check whether a received message is a redelivered message.



The `cmt-max-runtime-exceptions` property is deprecated.

Chapter 15. Using Container-Managed Persistence

This chapter contains information on how EJB 2.1 container-managed persistence (CMP) works in Eclipse GlassFish.

The following topics are addressed here:

- [Eclipse GlassFish Support for CMP](#)
- [CMP Mapping](#)
- [Automatic Schema Generation for CMP](#)
- [Schema Capture](#)
- [Configuring the CMP Resource](#)
- [Performance-Related Features](#)
- [Configuring Queries for 1.1 Finders](#)
- [CMP Restrictions and Optimizations](#)



The Web Profile of the Eclipse GlassFish supports the EJB 3.1 Lite specification, which allows enterprise beans within web applications, among other features. The full Eclipse GlassFish supports the entire EJB 3.1 specification. For details, see [JSR 318](#) (<http://jcp.org/en/jsr/detail?id=318>).

Eclipse GlassFish Support for CMP

Eclipse GlassFish 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](#).
 - 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 Eclipse GlassFish 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](#).

- Eclipse GlassFish performance-related features, including the following:
 - Version column consistency checking
 - Relationship prefetching
 - Read-Only Beans

For details, see [Performance-Related Features](#).

CMP Mapping

Implementation for entity beans that use CMP is mostly a matter of mapping CMP fields and CMR fields (relationships) to the database.

The following topics are addressed here:

- [Mapping Capabilities](#)
- [The Mapping Deployment Descriptor File](#)
- [Mapping Considerations](#)

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 Eclipse GlassFish 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.
- **`glassfish-ejb-jar.xml`** - The Eclipse GlassFish standard file for assembling enterprise beans. For

a detailed description, see "[The glassfish-ejb-jar.xml File](#)" in Eclipse GlassFish 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 Eclipse GlassFish 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](#).

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).



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 Eclipse GlassFish 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](#)). You can manually generate the schema using the [capture-schema](#) utility ([Using the capture-schema Utility](#)).

Mapping Considerations

The following topics are addressed here:

- [Join Tables and Relationships](#)
- [Automatic Primary Key Generation](#)
- [Fixed Length CHAR Primary Keys](#)
- [Managed Fields](#)
- [BLOB Support](#)
- [CLOB Support](#)

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](#).

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 Eclipse GlassFish 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 Eclipse GlassFish does not support database-generated primary key values.

If the database schema is created during deployment, the Eclipse GlassFish 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 Eclipse GlassFish 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 Eclipse GlassFish 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 Eclipse GlassFish always tries to keep the state of the objects in memory synchronized with the database.

A managed field can have any `fetches-with` subelement. If the `fetches-with` subelement is `<default/>`, the `-DallowManagedFieldsInDefaultFetchGroup` flag must be set to `true`. See [Default Fetch Group Flags](#) and `"fetches-with"` in Eclipse GlassFish 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 Eclipse GlassFish 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 Eclipse GlassFish, see the [Eclipse GlassFish Release Notes](#). For configurations of supported and other drivers, see "[Configuration Specifics for JDBC Drivers](#)" in Eclipse GlassFish 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 `glassfish-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 Eclipse GlassFish 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 Eclipse GlassFish, see the [Eclipse GlassFish Release Notes](#). For configurations of supported and other drivers, see "[Configuration Specifics for JDBC Drivers](#)" in Eclipse GlassFish 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 `glassfish-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 Eclipse GlassFish 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.

The following topics are addressed here:

- [Supported Data Types for CMP](#)
- [Generation Options for CMP](#)



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 9-1 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
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



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 Eclipse

GlassFish, see the [Eclipse GlassFish Release Notes](#). For configurations of supported and other drivers, see "[Configuration Specifics for JDBC Drivers](#)" in Eclipse GlassFish Administration Guide.

Table 9-2 Mappings of JDBC Types to Database Vendor Specific Types for CMP

JDBC Type	Apache Derby, CloudScape	Oracle	DB2	Sybase ASE 12.5	MS-SQL Server
BIT	SMALLINT	SMALLINT	SMALLINT	TINYINT	BIT
TINYINT	SMALLINT	SMALLINT	SMALLINT	TINYINT	TINYINT
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



Before using these options, make sure you have a properly configured CMP resource. See [Configuring the CMP Resource](#).

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](#).

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](#).

The following optional data subelements of the `cmp-resource` element in the `glassfish-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 Eclipse GlassFish Application Deployment Guide and [Configuring the CMP Resource](#).

Table 9-3 The `glassfish-ejb-jar.xml` Generation Elements

Element	Default	Description
<code>create-tables-at-deploy</code>	false	If <code>true</code> , causes database tables to be created for beans that are automatically mapped by the EJB container. No unique constraints are created. If <code>false</code> , does not create tables.
<code>drop-tables-at-undeploy</code>	false	If <code>true</code> , causes database tables that were automatically created when the bean(s) were last deployed to be dropped when the bean(s) are undeployed. If <code>false</code> , does not drop tables.
<code>database-vendor-name</code>	none	<p>Specifies the name of the database vendor for which tables are created. Allowed values are <code>javadb</code>, <code>db2</code>, <code>mssql</code>, <code>mysql</code>, <code>oracle</code>, <code>postgresql</code>, <code>pointbase</code>, <code>derby</code> (also for CloudScape), and <code>sybase</code>, case-insensitive.</p> <p>If no value is specified, a connection is made to the resource specified by the <code>jndi-name</code> subelement of the <code>cmp-resource</code> element in the <code>glassfish-ejb-jar.xml</code> 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.</p>

Element	Default	Description
<code>schema-generator-properties</code>	none	<p>Specifies field-specific column attributes in <code>property</code> subelements. Each property name is of the following format:</p> <p>bean-name`.field-name`.attribute</p> <p>For example:</p> <p><code>Employee.firstName.jdbc-type</code></p> <p>Also allows you to set the <code>use-unique-table-names</code> property. If <code>true</code>, this property specifies that generated table names are unique within each Eclipse GlassFish domain. The default is <code>false</code>.</p> <p>For further information and an example, see "schema-generator-properties" in Eclipse GlassFish Application Deployment Guide.</p>

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

Table 9-4 The `asadmin deploy` and `asadmin deploydir` Generation Options for CMP

Option	Default	Description
<code>--createtables</code>	none	<p>If <code>true</code>, causes database tables to be created for beans that need them. No unique constraints are created. If <code>false</code>, does not create tables. If not specified, the value of the <code>create-tables-at-deploy</code> attribute in <code>glassfish-ejb-jar.xml</code> is used.</p>

Option	Default	Description
<code>--dropandcreate-tables</code>	none	<p>If <code>true</code>, and if tables were automatically created when this application was last deployed, tables from the earlier deployment are dropped and fresh ones are created.</p> <p>If <code>true</code>, 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.</p> <p>If <code>false</code>, settings of <code>create-tables-at-deploy</code> or <code>drop-tables-at-undeploy</code> in the <code>glassfish-ejb-jar.xml</code> file are overridden.</p>
<code>--unique-table-names</code>	none	<p>If <code>true</code>, specifies that table names are unique within each Eclipse GlassFish domain. If not specified, the value of the <code>use-unique-table-names</code> property in <code>glassfish-ejb-jar.xml</code> is used.</p>

Option	Default	Description
<code>--dbvendorname</code>	none	<p>Specifies the name of the database vendor for which tables are created. Allowed values are <code>javadb</code>, <code>db2</code>, <code>mssql</code>, <code>oracle</code>, <code>postgresql</code>, <code>pointbase</code>, <code>derby</code> (also for CloudScape), and <code>sybase</code>, case-insensitive.</p> <p>If not specified, the value of the <code>database-vendor-name</code> attribute in <code>glassfish-ejb-jar.xml</code> is used.</p> <p>If no value is specified, a connection is made to the resource specified by the <code>jndi-name</code> subelement of the <code>cmp-resource</code> element in the <code>glassfish-ejb-jar.xml</code> 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.</p>

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 9-5 The `asadmin undeploy` Generation Options for CMP

Option	Default	Description
<code>--droptables</code>	none	<p>If <code>true</code>, causes database tables that were automatically created when the bean(s) were last deployed to be dropped when the bean(s) are undeployed. If <code>false</code>, does not drop tables.</p> <p>If not specified, the value of the <code>drop-tables-at-undeploy</code> attribute in <code>glassfish-ejb-jar.xml</code> is used.</p>

For more information about the `asadmin deploy`, `asadmin deploydir`, and `asadmin undeploy` commands, see the [Eclipse GlassFish Reference Manual](#).

When command line and `glassfish-ejb-jar.xml` options are both specified, the `asadmin` options take precedence.

Schema Capture

The following topics are addressed here:

- [Automatic Database Schema Capture](#)
- [Using the capture-schema Utility](#)

Automatic Database Schema Capture

You can configure a CMP bean in Eclipse GlassFish 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 `glassfish-ejb-jar.xml` file is used to get a connection to the database, and automatic generation of the schema is performed.



Before capturing the database schema automatically, make sure you have a properly configured CMP resource. See [Configuring the CMP Resource](#).

Using the capture-schema Utility

You can use the `capture-schema` command to manually generate the database metadata (`.dbschema`) file. For details, see the [Eclipse GlassFish 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` option 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 `glassfish-ejb-jar.xml` file. Set `PersistenceManagerFactory` properties as properties of the `cmp-resource` element in the `glassfish-ejb-jar.xml` file. See "[cmp-resource](#)" in Eclipse GlassFish 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 Eclipse GlassFish, see the [Eclipse GlassFish Release Notes](#). For configurations of supported and other drivers, see "Configuration Specifics for JDBC Drivers" in Eclipse GlassFish Administration Guide.

For example, if the JDBC resource has the JNDI name `jdbc/MyDatabase`, set the CMP resource in the `glassfish-ejb-jar.xml` file as follows:

```
<cmp-resource>
  <jndi-name>jdbc/MyDatabase</jndi-name>
</cmp-resource>
```

Performance-Related Features

The Eclipse GlassFish 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.

The following topics are addressed here:

- [Version Column Consistency Checking](#)
- [Relationship Prefetching](#)
- [Read-Only Beans](#)
- [Default Fetch Group Flags](#)



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 Eclipse GlassFish 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 Eclipse GlassFish Application Deployment Guide.

To disable prefetching for specific custom finders, use the `prefetch-disabled` element in the `glassfish-ejb-jar.xml` file. See "`prefetch-disabled`" in Eclipse GlassFish 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 Eclipse GlassFish provides is the read-only bean, an entity bean that is never modified by an EJB client. Read-only beans avoid database updates completely.



Read-only beans are specific to the Eclipse GlassFish 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 Eclipse GlassFish 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 `glassfish-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](#).

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 Eclipse GlassFish 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

The following topics are addressed here:

- [About JDOQL Queries](#)
- [Query Filter Expression](#)
- [Query Parameters](#)
- [Query Variables](#)
- [JDOQL Examples](#)

About JDOQL Queries

The Enterprise JavaBeans Specification, v1.1 does not specify the format of the finder method description. The Eclipse GlassFish 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 Eclipse GlassFish specific deployment descriptor (`glassfish-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) (<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 Eclipse GlassFish 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`.



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 `glassfish-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 `glassfish-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 && 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_JD0State 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 `glassfish-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 > player.salary) && (player.name == name)
  </query-filter>
  <query-variables>
    mypackage.PlayerEJB_170160966_JD0State player
  </query-variables>
</finder>
```

CMP Restrictions and Optimizations

This section discusses restrictions and performance optimizations that pertain to using CMP.

The following topics are addressed here:

- [Disabling ORDER BY Validation](#)
- [Setting the Heap Size on DB2](#)
- [Eager Loading of Field State](#)
- [Restrictions on Remote Interfaces](#)
- [PostgreSQL Case Insensitivity](#)
- [No Support for lock-when-loaded on Sybase](#)
- [Sybase Finder Limitation](#)
- [Date and Time Fields](#)

- Set `RECURSIVE_TRIGGERS` to `false` on `MSSQL`
- [MySQL Database Restrictions](#)

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 Oracle 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.html>.

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 `fetches-with` element in `sun-cmp-mappings.xml` for fields that are used infrequently. See "[fetches-with](#)" in Eclipse GlassFish 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 Eclipse GlassFish 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 Eclipse GlassFish 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://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 16. Developing Java Clients

This chapter describes how to develop, assemble, and deploy Java clients.

The following topics are addressed here:

- [Introducing the Application Client Container](#)
- [Developing Clients Using the ACC](#)
- [Developing Clients Without the ACC](#)



The Web Profile of the OracleEclipse GlassFish supports the EJB 3.1 Lite specification, which allows enterprise beans within web applications, among other features. The full Eclipse GlassFish supports the entire EJB 3.1 specification. For details, see [JSR 318](#).

Accordingly, the Application Client Container is supported only in the full Eclipse GlassFish, not in the Web Profile.

JMS resources are supported only in the full Eclipse GlassFish, not in the Web Profile. See [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 Jakarta EE application client components (application clients), which are used to access a variety of Jakarta EE services (such as JMS resources, EJB components, web services, security, and so on.) from a JVM outside the Eclipse GlassFish.

The ACC communicates with the Eclipse GlassFish using RMI-IIOP protocol and manages the details of RMI-IIOP communication using the client ORB that is bundled with it. Compared to other Jakarta EE containers, the ACC is lightweight.

For information about debugging application clients, see [Application Client Debugging](#).



Interoperability between application clients and Eclipse GlassFishs 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 Eclipse GlassFish'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 Eclipse GlassFish's authentication system. It also supports SSL (Secure Socket Layer)/IIOP if configured and when necessary; see [Using RMI/IIOP Over SSL](#).

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 Jakarta EE specification, section 9.2, Application Clients: Security, for more details.

Application clients can use [Programmatic Login Using the ProgrammaticLogin Class](#).

ACC Naming

The client container enables the application clients to use the Java Naming and Directory Interface (JNDI) to look up Jakarta 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.

Application Client 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 Eclipse GlassFish 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](#).

Application Client JAR File

In Eclipse GlassFish 8, the downloaded appclient JAR file is smaller than in previous releases, with dependent classes in separate JAR files. When copying the downloaded 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.

The following topics are addressed here:

- [To Access an EJB Component From an Application Client](#)
- [To Access a JMS Resource From an Application Client](#)

- [Using Java Web Start](#)
- [Using the Embeddable ACC](#)
- [Running an Application Client Using the `appclient` Script](#)
- [Using the `package-appclient` Script](#)
- [The `client.policy` File](#)
- [Using RMI/IIOP Over SSL](#)
- [Connecting to a Remote EJB Module Through a Firewall](#)
- [Specifying a Splash Screen](#)
- [Setting Login Retries](#)
- [Using Libraries with Application Clients](#)

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](#).
If load balancing is enabled as in Step 7 and the EJB components being accessed are in a different cluster, the endpoint list must be included in the lookup, as follows:

```
corbaname:host1:port1,host2:port2,.../NameService#ejb/jndi-name
```

2. Define the `@EJB` annotations or the `ejb-ref` elements in the `application-client.xml` file. Define the corresponding `ejb-ref` elements in the `glassfish-application-client.xml` file.
For more information on the `glassfish-application-client.xml` file, see "[The `glassfish-application-client.xml` file](#)" in Eclipse GlassFish Application Deployment Guide. For a general explanation of how to map JNDI names using reference elements, see [Mapping References](#).
3. Deploy the application client and EJB component together in an application.
For more information on deployment, see the [Eclipse GlassFish Application Deployment Guide](#).
To get the client JAR file, use the `--retrieve` option of the `asadmin deploy` command.
To retrieve the stubs and ties generated during deployment, use the `asadmin get-client-stubs` command.
For details, see the [Eclipse GlassFish Reference Manual](#).
4. Ensure that the client JAR file includes the following files:
 - A Java class to access the bean.
 - `application-client.xml` - (optional) Jakarta EE application client deployment descriptor.
 - `glassfish-application-client.xml` - (optional) Eclipse GlassFish specific client deployment descriptor. For information on the `glassfish-application-client.xml` file, see "[The `glassfish-application-client.xml` file](#)" in Eclipse GlassFish 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. This step is not needed if the client and server machines are the same.

If you are using the `appclient` script, package the Eclipse GlassFish 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](#) and the [Eclipse GlassFish 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 Eclipse GlassFish 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](#).

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 [Eclipse GlassFish Reference Manual](#).

7. To set up load balancing and failover of remote EJB references, define at least two `target-server` elements in the `sun-acc.xml` file or the `appclient` script. This step is not needed for Java Web Start.

If the Eclipse GlassFish instance on which the application client is deployed participates in a cluster, the ACC finds all currently active IIOP endpoints in the cluster automatically. However, a client should have at least two endpoints specified for bootstrapping purposes, in case one of the endpoints has failed.

The `target-server` elements in the `sun-acc.xml` file specify one or more IIOP endpoints used for load balancing. The `address` attribute is an IPv4 address or host name, and the `port` attribute specifies the port number. See "`client-container`" in Eclipse GlassFish Application Deployment Guide.

The `--targetserver` option of the `appclient` script specifies one or more IIOP endpoints used for load balancing. For more information, see [Running an Application Client Using the appclient Script](#).

Next Steps

- For instructions on running the application client, see [Using Java Web Start](#) or [Running an Application Client Using the appclient Script](#).
- For more information about RMI-IIOP load balancing and failover, see "[RMI-IIOP Load Balancing and Failover](#)" in Eclipse GlassFish High Availability Administration Guide.

To Access a JMS Resource From an Application Client

1. Create a JMS client.

For detailed instructions on developing a JMS client, see [Java Message Service Examples](#) in The Jakarta EE Tutorial.

2. Next, configure a JMS resource on the Eclipse GlassFish.

For information on configuring JMS resources, see "[Administering JMS Connection Factories and Destinations](#)" in Eclipse GlassFish Administration Guide.

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 `glassfish-application-client.xml` file.

For more information on the `glassfish-application-client.xml` file, see "[The glassfish-application-client.xml file](#)" in Eclipse GlassFish Application Deployment Guide. For a general explanation of how to map JNDI names using reference elements, see [Mapping References](#).

4. Ensure that the client JAR file includes the following files:

- A Java class to access the resource.
- `application-client.xml` - (optional) Jakarta EE application client deployment descriptor.
- `glassfish-application-client.xml` - (optional) Eclipse GlassFish specific client deployment descriptor. For information on the `glassfish-application-client.xml` file, see "[The glassfish-application-client.xml file](#)" in Eclipse GlassFish 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. This step is not needed if the client and server machines are the same.

If you are using the `appclient` script, package the Eclipse GlassFish 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](#) and the [Eclipse GlassFish Reference Manual](#).

6. Run the application client.

See [Using Java Web Start](#) or [Running an Application Client Using the appclient Script](#).

Using Java Web Start

Java Web Start allows your application client to be easily launched and automatically downloaded and updated.

The following topics are addressed here:

- [Enabling and Disabling Java Web Start](#)
- [Downloading and Launching an Application Client](#)
- [The Application Client URL](#)
- [Signing JAR Files Used in Java Web Start](#)
- [Error Handling](#)
- [Vendor Icon, Splash Screen, and Text](#)
- [Creating a Custom JNLP File](#)

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 `glassfish-application-client.xml` file. See the [Eclipse GlassFish Application Deployment Guide](#).

The Eclipse GlassFish 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 applications.application.app-name.property.java-web-start-enabled="false"
```

To disable Java Web Start for a stand-alone eligible application client, use the following command:

```
asadmin set applications.application.module-name.property.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 [Eclipse GlassFish 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](#).

- 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 when you deploy the application. For details about naming, see "[Naming Standards](#)" in Eclipse GlassFish 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 `glassfish-application-client.xml` file. This overrides the appclient-module-id or application-id. See the [Eclipse GlassFish 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
```



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 if that certificate is not trusted. It then 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.

Your first Java Web Start launch of an application client is likely to involve this prompting because by default Eclipse GlassFish uses a self-signed certificate that is not linked to a trusted authority.

The Eclipse GlassFish 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 Eclipse GlassFish, which starts an application client during a Java Web Start launch: `as-install/lib/gf-client.jar`.

The other type is a generated application client JAR file. As part of deployment, the Eclipse GlassFish 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 one of the JAR files 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 topics are addressed here:

- [Automatically Signing JAR Files](#)
- [Using the `jar-signing-alias` Deployment Property](#)

Automatically Signing JAR Files

The Eclipse GlassFish 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 Eclipse GlassFish looks for `domain-dir`/java-web-start/gf-client.jar``. When a request for an application's generated application client JAR file arrives, the Eclipse GlassFish 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 Eclipse GlassFish creates a signed version of the JAR file automatically and deposits it in the relevant directory. Whether the Eclipse GlassFish 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, by default the Eclipse GlassFish uses its self-signed certificate. When you create a new domain, either by installing the Eclipse GlassFish or by using the `asadmin create-domain` command, the Eclipse GlassFish 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.

Starting with Java SE 7 Update 21, stricter security is enforced for applications launched using Java Web Start. Application users will see various security messages, depending on their Java security settings. If Java security is set to Very High on their systems, users will not be able to launch application clients signed using the Eclipse GlassFish self-signed certificate.

To minimize impacts to application users, all Java Web Start applications should be signed with a trusted certificate instead of the Eclipse GlassFish self-signed certificate. If you use the Eclipse GlassFish Java Web Start feature or deploy applications that provide their own Java Web Start applications, perform the following steps:

1. Obtain a trusted certificate from a certification authority if your organization does not already have one.
2. Stop Eclipse GlassFish.
3. Replace the Eclipse GlassFish self-signed certificate with the trusted certificate by importing the trusted certificate into the Eclipse GlassFish keystore using the `s1as` alias. By default, the keystore is located at `domain-dir`/config/keystore.p12`` (PKCS12 format, recommended). For legacy compatibility, JKS format keystores (`keystore.jks`) are also supported.

For more information about importing a trusted certificate into the domain keystore, see "[Administering JSSE Certificates](#)" in Eclipse GlassFish Server Security Guide.

4. Delete any signed JARs already generated by Eclipse GlassFish:
 - a. At the command prompt, type:
`rm -rf domain-dir/java_web_start`
 - b. For each application that contains an application client launched using Java Web Start, type:
`rm -rf domain-dir/generated/xml/app-name/signed`
 - c. Restart Eclipse GlassFish.
5. Ensure that the Java security setting on user systems is set to Very High.

After you perform these steps, the first time a user launches an application client on their system, Java Web Start detects that the server's signed JARs are newer than those cached on the user's system and downloads them again. This happens on the first launch only, regardless of the client. Even though the application client is now signed using a trusted certificate, users will again be asked whether to trust the downloaded application and can choose to skip that prompt for future launches.

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, Eclipse GlassFish signs the JAR file automatically using the self-signed certificate from the domain, `self`. Java Web Start then asks the user whether to trust the code and displays the Eclipse GlassFish 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. To add a certificate to the domain keystore, see "[Administering JSSE Certificates](#)" in Eclipse GlassFish Security 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 [Eclipse GlassFish Reference Manual](#).

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 `glassfish-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 `glassfish-application-client.xml` file, see the [Eclipse GlassFish Application Deployment Guide](#).

You can also specify a vendor-specific icon, splash screen, text string, or a combination by using a custom JNLP file; see [Creating a Custom JNLP File](#).

Creating a Custom JNLP File

You can partially customize the Java Network Launching Protocol (JNLP) file that Eclipse GlassFish uses for Java Web Start.

The following topics are addressed here:

- [Specifying the JNLP File in the Deployment Descriptor](#)
- [Referring to JAR Files from the JNLP File](#)
- [Referring to Other JNLP Files](#)
- [Combining Custom and Automatically Generated Content](#)

For more information about JNLP, see the [Java Web Start Architecture JNLP Specification and API Documentation](http://java.sun.com/javase/technologies/desktop/javawebstart/download-spec.html) (<http://java.sun.com/javase/technologies/desktop/javawebstart/download-spec.html>).

Specifying the JNLP File in the Deployment Descriptor

To specify a custom JNLP file for Java Web Start, use the `jnlp-doc` element in the `glassfish-application-client.xml` file. If none is specified, a default JNLP file is generated.

The value of the `jnlp-doc` element is a relative path with the following format:

```
[path-to-JAR-in-EAR!]path-to-JNLP-in-JAR
```

The default path-to-JAR-in-EAR is the current application client JAR file. For example, if the JNLP file is in the application client JAR file at `custom/myInfo.jnlp`, the element value would look like this:

```
<java-web-start-access>
  <jnlp-doc>custom/myInfo.jnlp</jnlp-doc>
</java-web-start-access>
```

If the application client is inside an EAR file, you can place the custom JNLP file inside another JAR file in the EAR. For example, if the JNLP file is in a JAR file at `other/myLib.jar`, the element value would look like this, with an exclamation point (!) separating the path to the JAR from the path in the JAR:

```
<java-web-start-access>
  <jnlp-doc>other/myLib.jar!custom/myInfo.jnlp</jnlp-doc>
</java-web-start-access>
```

For more information about the `glassfish-application-client.xml` file, see the Eclipse GlassFish Application Deployment Guide.

Referring to JAR Files from the JNLP File

As with any JNLP document, the custom JNLP file can refer to JAR files the application client requires.

Do not specify every JAR on which the client depends. Eclipse GlassFish automatically handles JAR files that the Jakarta EE specification requires to be available to the application client. This includes JAR files listed in the application client JAR file's manifest `Class-Path` and JAR files in the EAR file's library directory (if any) and their transitive closures. The custom JNLP file should specify only those JAR files the client needs that Eclipse GlassFish would not otherwise include.

Package these JAR files in the EAR file, as with any JAR file required by an application client. Use relative URIs in the `<jar href="...">` and `<nativelib href="...">` elements to point to the JAR files. The codebase that Eclipse GlassFish assigns for the final client JNLP file corresponds to the top level of the EAR file. Therefore, relative `href` references correspond directly to the relative path to the

JAR files within the EAR file.

Neither the Jakarta EE specification nor Eclipse GlassFish supports packaging JAR files inside the application client JAR file itself. Nothing prevents this, but Eclipse GlassFish does no special processing of such JAR files. They do not appear in the runtime class path and they cannot be referenced from the custom JNLP file.

Referring to Other JNLP Files

The JNLP file can also refer to other custom JNLP files using `<extension href="..." />` elements. To be consistent with relative `href` references to JAR files, the relative `href` references to JNLP files are resolved within the EAR file. You can place these JNLP files directly in the EAR file or inside JAR files that the EAR file contains. Use one of these formats for these `href` references:

```
[path-to-JAR-in-EAR!]path-to-JNLP-in-JAR  
  
path-to-JNLP-in-EAR
```

Note that these formats are not equivalent to the format of the `jnlp-doc` element in the `glassfish-application-client.xml` file.

These formats follow the standard entry-within-a-JAR URI syntax and semantics. Support for this syntax comes from the automated Java Web Start support in Eclipse GlassFish. This is not a feature of Java Web Start or the JNLP standard.

Combining Custom and Automatically Generated Content

Eclipse GlassFish recognizes these types of content in the JNLP file:

- Owned — Eclipse GlassFish owns the content and ignores any custom content
- Merged — Automatically generated content and custom content are merged
- Defaulted — Custom content is used if present, otherwise default content is provided

You can compose a complete JNLP file and package it with the application client. Eclipse GlassFish then combines it with its automatically generated JNLP file. You can also provide content that only adds to or replaces what Eclipse GlassFish generates. The custom content must conform to the general structure of the JNLP format so that Eclipse GlassFish can properly place it in the final JNLP file.

For example, to specify a single native library to be included only for Windows systems, the new element to add might be as follows:

```
<nativelib href="windows/myLib.jar"/>
```

However, you must indicate where in the overall document this element belongs. The actual custom JNLP file should look like this:

```
<jnlp>
  <resources os="Windows">
    <nativelib href="windows/myLib.jar"/>
  </resources>
</jnlp>
```

Eclipse GlassFish provides default `<information>` and `<resources>` elements, without specifying attributes such as `os`, `arch`, `platform`, or `locale`. Eclipse GlassFish merges its own content within those elements with custom content under those elements. Further, you can provide your own `<information>` and `<resources>` elements (and fragments within them) that specify at least one of these attributes.

In general, you can perform the following customizations:

- Override the Eclipse GlassFish defaults for the child elements of `<information>` elements that have no attribute settings for `os`, `arch`, `platform`, and `locale`. Among these child elements are `<title>`, `<vendor>`, `<description>`, `<icon>`, and so on.
- Add `<information>` elements with `os`, `arch`, `platform`, or `locale` settings. You can also add child elements.
- Add child elements of `<resources>` elements that have no attribute settings for `os`, `arch`, or `locale`. Among these child elements are `<jar>`, `<property>`, `<nativelib>`, and so on. You can also customize attributes of the `<java>` child element.
- Add `<resources>` elements that specify at least one of `os`, `arch`, or `locale`. You can also add child elements.

This flexibility allows you to add JAR files to the application (including platform-specific native libraries) and set properties to control the behavior of your application clients.

The following tables provide more detail about what parts of the JNLP file you can add to and modify.

Table 10-1 Owned JNLP File Content

JNLP File Fragment	Description
<code><jnlp codebase="xxx" ...></code>	Eclipse GlassFish controls this content for application clients packaged in EAR files. The developer controls this content for application clients packaged in WAR files.
<code><jnlp href="xxx" ...></code>	Eclipse GlassFish controls this content for application clients packaged in EAR files. The developer controls this content for application clients packaged in WAR files.
<code><jnlp></code> <code><security></code>	Eclipse GlassFish must control the permissions requested for each JNLP file. All permissions are needed for the main file, which launches the ACC. The permissions requested for other JNLP documents depend on whether the JAR files referenced in those documents are signed.

JNLP File Fragment	Description
<pre><jnlp> <application-desc> <argument> ...</pre>	Eclipse GlassFish sets the main-class and the arguments to be passed to the client.

Table 10-2 Defaulted JNLP File Content

JNLP File Fragment	Description
<pre><jnlp spec="xxx" ...></pre>	Specifies the JNLP specification version.
<pre><jnlp> <information [no-attributes]></pre>	Specifies the application title, vendor, home page, various description text values, icon images, and whether offline execution is allowed.
<pre><jnlp> <resources [no-attributes]> <java version="xxx" java-vm-args="yyy" ...></pre>	Specifies the Java SE version or selected VM parameter settings.

Table 10-3 Merged JNLP File Content

JNLP File Fragment	Description
<pre><jnlp> <information [attributes]></pre>	You can specify one or more of the os , arch , platform , and locale attributes for the <information> element. You can also specify child elements; Eclipse GlassFish provides no default children.
<pre><jnlp> <resources [attributes]></pre>	You can specify one or more of the os , arch , platform , and locale attributes for the <resources> element. You can also specify child elements; Eclipse GlassFish provides no default children.
<pre><jnlp> <resources [no-attributes]> <jar ...></pre>	Adds JAR files to be included in the application to the JAR files provided by Eclipse GlassFish.
<pre><jnlp> <resources [no-attributes]> <nativelib ...></pre>	Adds native libraries to be included in the application. Each entry in a JAR listed in a <nativelib> element must be a native library for the correct platform. The full syntax of the <nativelib> element lets the developer specify the platform for that native library.

JNLP File Fragment	Description
<pre><jnlp> <resources [no-attributes]> <property ...></pre>	Adds system properties to be included in the application to the system properties defined by Eclipse GlassFish.
<pre><jnlp> <resources [no-attributes]> <extension ...></pre>	Specifies another custom JNLP file.
<pre><jnlp> <component-desc ...></pre>	Includes another custom JNLP file that specifies a component extension.
<pre><jnlp> <installer-desc ...></pre>	Includes another custom JNLP file that specifies an installer extension.

Using the Embeddable ACC

You can embed the ACC into your application client. If you place the `as-install/lib/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. For details, see the [Eclipse GlassFish Reference Manual](#).

Using the `package-appclient` Script

You can package the Eclipse GlassFish 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 [Eclipse GlassFish 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 Jakarta 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://docs.oracle.com/javase/tutorial/security/tour2/index.html>.

For more information about the permissions you can set in the `client.policy` file, see <http://docs.oracle.com/javase/7/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 `glassfish-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 Using the ProgrammaticLogin Class](#) 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 `glassfish-ejb-jar.xml` and `sun-acc.xml` files, see the [Eclipse GlassFish Application Deployment Guide](#).

To use a client certificate, configure the `ior-security-config` element in the `glassfish-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](#).
- 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}"
```

- Optionally, you can set the `env` element using Ant. 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 Eclipse GlassFish instance that is behind a firewall, you must set ORB Virtual Address Agent Implementation (ORBVAAs) options. Use the `asadmin create-jvm-options` command as follows:

```
asadmin create-jvm-options -Dcom.sun.corba.ee.ORBVAASHost=public-IP-address
asadmin create-jvm-options -Dcom.sun.corba.ee.ORBVAASPort=public-port
asadmin create-jvm-options
-Dcom.sun.corba.ee.ORBUserConfigurators.com.sun.corba.ee.impl.plugin.hwlb.VirtualAddressAgentImpl=x
```

Set the `ORBVAASHost` and `ORBVAASPort` 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 Eclipse GlassFish configuration.

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](#).
- In the environment that runs the `appclient` script, set the `VM_OPTS` 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](#).

During application (EAR file) deployment, the Eclipse GlassFish generates façade JAR files, one for the application and one for each application client in the application. During application client module deployment, the Eclipse GlassFish 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 [Eclipse GlassFish 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 [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`.

Developing Clients Without the ACC

This section describes the procedure to create, assemble, and deploy a Java-based client that is not packaged using the Application Client Container (ACC).

The following topics are addressed here:

- [To access an EJB component from a stand-alone client](#)
- [To access an EJB component from a server-side module](#)
- [To access a JMS resource from a stand-alone client](#)

For information about using the ACC, see [Developing Clients Using the ACC](#).

To access an EJB component from a stand-alone client

1. In your client code, instantiate the `InitialContext`:

```
InitialContext ctx = new InitialContext();
```

It is not necessary to explicitly instantiate a naming context that points to the CosNaming service.

2. In the client code, look up the home object by specifying the JNDI name of the home object.

Here is an EJB 2.x example:

```
Object ref = ctx.lookup("jndi-name");
BeanAHome = (BeanAHome)PortableRemoteObject.narrow(ref, BeanAHome.class);
```

Here is an EJB 3.x example:

```
BeanRemoteBusiness bean =(BeanRemoteBusiness) ctx.lookup(
    "com.acme.BeanRemoteBusiness");
```

If load balancing is enabled as in Step 6 and the EJB components being accessed are in a different cluster, the endpoint list must be included in the lookup, as follows:

```
corbaname:host1:port1,host2:port2,.../NameService#ejb/jndi-name
```

For more information about naming and lookups, see [Accessing the Naming Context](#).

3. Deploy the EJB component to be accessed.

For more information on deployment, see "[About Deployment Tools](#)" in Eclipse GlassFish Application Deployment Guide.

4. Copy the `as-install/lib/gf-client.jar` file to the client machine and include it in the classpath on the client side.

The `gf-client.jar` file references Eclipse GlassFish JAR files in its `MANIFEST.MF` file. If there is no Eclipse GlassFish installation on the client machine, you must also copy the `as-install/modules` directory to the client machine and maintain its directory structure relative to the `as-install/lib/gf-client.jar` file. Or you can use the `package-appclient` script; see [Using the package-appclient Script](#).

5. To access EJB components that are residing in a remote system, set the following system properties for the Java Virtual Machine startup options:

```
-Dorg.omg.CORBA.ORBInitialHost=${ORBhost}
-Dorg.omg.CORBA.ORBInitialPort=${ORBport}
```

Here `ORBhost` is the Eclipse GlassFish hostname and `ORBport` is the ORB port number (default is `3700` for the default server instance, named `server`).

You can use the `asadmin get` command to get the IIOP port numbers. For example:

```
asadmin get "configs.config.server-config.iop-service.iop-listener.orb-listener-1.*"
```

6.

To set up load balancing and remote EJB reference failover, define the `endpoints` property as follows:

```
-Dcom.sun.appserv.iop.endpoints=host1:port1,host2:port2,...
```

The `endpoints` property specifies a comma-separated list of one or more IIOP endpoints used for load balancing. An IIOP endpoint is in the form `host:port`, where the host is an IPv4 address or host name, and the port specifies the port number.

If the `endpoints` list is changed dynamically in the code, the new list is used only if a new `InitialContext` is created.

7. Make sure the `etc/hosts` file on the client machine maps the Eclipse GlassFish hostname and external IP address.
8. Run the stand-alone client.

As long as the client environment is set appropriately and the JVM is compatible, you merely need to run the `main` class.

To access an EJB component from a server-side module

A server-side module can be a servlet, another EJB component, or another type of module.

1.

In your module code, instantiate the `InitialContext`:

```
InitialContext ctx = new InitialContext();
```

It is not necessary to explicitly instantiate a naming context that points to the CosNaming service.

To set up load balancing and remote EJB reference failover, define the `endpoints` property as follows:

```
Hashtable env = new Hashtable();  
env.put("com.sun.appserv.iop.endpoints", "host1:port1,host2:port2,...");  
InitialContext ctx = new InitialContext(env);
```

The `endpoints` property specifies a comma-separated list of one or more IIOP endpoints used for load balancing. An IIOP endpoint is in the form `host:port`, where the host is an IPv4 address

or host name, and the port specifies the port number.

You can use the `asadmin get` command to get the IIOP port numbers. For example:

```
asadmin get "configs.config.server-config.iiop-service.iiop-listener.orb-listener-1.*"
```

If the `endpoints` list is changed dynamically in the code, the new list is used only if a new `InitialContext` is created.

2. In the module code, look up the home object by specifying the JNDI name of the home object.

Here is an EJB 2.x example:

```
Object ref = ctx.lookup("jndi-name");  
BeanAHome = (BeanAHome)PortableRemoteObject.narrow(ref, BeanAHome.class);
```

Here is an EJB 3.x example:

```
BeanRemoteBusiness bean =(BeanRemoteBusiness) ctx.lookup(  
    "com.acme.BeanRemoteBusiness");
```

If load balancing is enabled as in Step 1 and the EJB components being accessed are in a different cluster, the endpoint list must be included in the lookup, as follows:

```
corbaname:host1:port1,host2:port2,.../NameService#ejb/jndi-name
```

For more information about naming and lookups, see [Accessing the Naming Context](#).

3. Deploy the EJB component to be accessed.

For more information on deployment, see "[About Deployment Tools](#)" in Eclipse GlassFish Application Deployment Guide.

4. To access EJB components that are residing in a remote system, set the following system properties for the Java Virtual Machine startup options:

```
-Dorg.omg.CORBA.ORBInitialHost=${ORBhost}  
-Dorg.omg.CORBA.ORBInitialPort=${ORBport}
```

Here `ORBhost` is the Application Server hostname and `ORBport` is the ORB port number (default is `3700` for the default server instance, named `server`).

5. Deploy the module.

For more information on deployment, see "[About Deployment Tools](#)" in Eclipse GlassFish Application Deployment Guide.

To access a JMS resource from a stand-alone client

1. Create a JMS client.

For detailed instructions on developing a JMS client, see [Java Message Service Examples](#) in The Jakarta EE Tutorial.

2. Configure a JMS resource on Eclipse GlassFish.

For information on configuring JMS resources, see "[Administering JMS Connection Factories and Destinations](#)" in Eclipse GlassFish Administration Guide.

3. Copy the following JAR files to the client machine and include them in the classpath on the client side:

- `gf-client.jar` - available at `as-install/lib`
- `imqjmsra.jar` - available at `as-install/lib/install/applications/jmsra`

The `gf-client.jar` file references Eclipse GlassFish JAR files in its `MANIFEST.MF` file. If there is no Eclipse GlassFish installation on the client machine, you must also copy the `as-install/modules` directory to the client machine and maintain its directory structure relative to the `as-install/lib/gf-client.jar` file. Or you can use the `package-applient` script; see [Using the package-applient Script](#).

4. To access EJB components that are residing in a remote system, set the following system properties for the Java Virtual Machine startup options:

```
-Dorg.omg.CORBA.ORBInitialHost=${ORBhost}  
-Dorg.omg.CORBA.ORBInitialPort=${ORBport}
```

Here `ORBhost` is the Application Server hostname and `ORBport` is the ORB port number (default is `3700` for the default server instance, named `server`).

You can use the `asadmin get` command to get the IIOP port numbers. For example:

```
asadmin get "configs.config.server-config.iiop-service.iiop-listener.orb-listener-  
1.*"
```

5. Run the stand-alone client.

As long as the client environment is set appropriately and the JVM is compatible, you merely need to run the `main` class.

Chapter 17. Developing Connectors

This chapter describes Eclipse GlassFish support for the Jakarta EE Connector Architecture, also known as [JSR 322](http://jcp.org/en/jsr/detail?id=322) (<http://jcp.org/en/jsr/detail?id=322>).

The Jakarta 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 Jakarta EE Connector architecture, EIS vendors no longer need to customize their product for each application server. Application server vendors who conform to the Jakarta 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 Jakarta EE Connector Architecture Specification.



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 Eclipse GlassFish.

For more information about connectors, see [Resource Adapters and Contracts](#) in The Jakarta EE Tutorial.

For information about deploying a connector to the Eclipse GlassFish, see the [Eclipse GlassFish Application Deployment Guide](#).

The following topics are addressed here:

- [Connector Support in the Eclipse GlassFish](#)
- [Advanced Connector Configuration Options](#)
- [Inbound Communication Support](#)
- [Outbound Communication Support](#)
- [Configuring a Message Driven Bean to Use a Resource Adapter](#)

Connector Support in the Eclipse GlassFish

The Eclipse GlassFish 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 Eclipse GlassFish. The Connector 1.5 specification introduces major additions in defining system level contracts between the Eclipse GlassFish 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 Eclipse GlassFish 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 Eclipse GlassFish 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 Eclipse GlassFish. 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 Eclipse GlassFish, 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 Eclipse GlassFish uses the Open Message Queue software. To provide JDBC infrastructure to clients, the Eclipse GlassFish uses its own JDBC system resource adapters. The Eclipse GlassFish automatically makes these system resource adapters available to any client that requires them.

For more information about JMS in the Eclipse GlassFish, see [Using the Java Message Service](#). For more information about JDBC in the Eclipse GlassFish, see [Using the JDBC API for Database Access](#).

Connector Configuration

The Eclipse GlassFish does not need to use **sun-ra.xml**, which previous Eclipse GlassFish 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 Eclipse GlassFish.

Advanced Connector Configuration Options

The following topics are addressed here:

- [Thread Associations](#)

- [Security Maps](#)
- [Work Security Maps](#)
- [Overriding Configuration Properties](#)
- [Testing a Connector Connection Pool](#)
- [Flushing a Connector Connection Pool](#)
- [Handling Invalid Connections](#)
- [Setting the Shutdown Timeout](#)
- [Specifying the Class Loading Policy](#)
- [Using Last Agent Optimization of Transactions](#)
- [Disabling Pooling for a Connection](#)
- [Using Application-Scoped Connectors](#)

Thread Associations

Connectors can submit work instances to the Eclipse GlassFish for execution. By default, the Eclipse GlassFish 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 [Eclipse GlassFish 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 [Eclipse GlassFish 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 [Eclipse GlassFish 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 Eclipse GlassFish 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 Eclipse GlassFish uses the wild card character specification, if any, to determine 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 [Eclipse GlassFish 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 Eclipse GlassFish, the Eclipse GlassFish 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 Eclipse GlassFish 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 [Eclipse GlassFish 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.

You can also use token replacement for overriding resource adapter configuration properties in

individual server instances when the resource adapter is deployed to a cluster. For example, for a property called `inboundPort`, you can assign the value `${inboundPort}`. You can then assign a different value to this property for each server instance. Changes to system properties take effect upon server restart.

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 [Eclipse GlassFish 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 [Eclipse GlassFish 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 [Eclipse GlassFish Reference Manual](#).

Handling Invalid Connections

If a resource adapter generates a `ConnectionErrorOccured` event, the Eclipse GlassFish 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 [Eclipse GlassFish Reference Manual](#).

The interface `ValidatingManagedConnectionFactory` exposes the method `getInvalidConnections` to allow retrieval of the invalid connections. The Eclipse GlassFish 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 [Eclipse GlassFish Reference Manual](#).

The Eclipse GlassFish 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 [Eclipse GlassFish 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 Eclipse GlassFish, see [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 [Eclipse GlassFish Reference Manual](#).

Using Application-Scoped Connectors

You can define an application-scoped connector or other resource for an enterprise application, web module, EJB module, connector module, or application client module by supplying a `glassfish-resources.xml` deployment descriptor file. For details, see "[Application-Scoped Resources](#)" in Eclipse GlassFish Application Deployment Guide.

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 [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>
      jakarta.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 `glassfish-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 Message Queue message provider, is specified as the resource adapter module identifier for the `SampleMDB` bean.

```

<glassfish-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>
...
</glassfish-ejb-jar>

```

When the message-driven bean is deployed, the Eclipse GlassFish 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 Eclipse GlassFish 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 Eclipse GlassFish 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` element 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 18. Developing Lifecycle Listeners

Lifecycle listener modules provide a means of running short or long duration Java-based tasks within the Eclipse GlassFish 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.



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, see the `com.sun.appserv.server` package.

The following topics are addressed here:

- [Server Life Cycle Events](#)
- [The LifecycleListener Interface](#)
- [The LifecycleEvent Class](#)
- [The Server Lifecycle Event Context](#)
- [Deploying a Lifecycle Module](#)
- [Considerations for Lifecycle Modules](#)

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 [Eclipse GlassFish Application Deployment Guide](#), or see the `asadmin create-lifecycle-module` command in the [Eclipse GlassFish 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. Do not place it in the `lib` directory for a particular instance, or it will be deleted when that instance synchronizes with 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](#).

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`.

Chapter 19. Developing OSGi-enabled Jakarta EE Applications

This chapter describes the features and interfaces that Eclipse GlassFish provides to develop OSGi-enabled enterprise applications. This chapter includes the following sections:

- [Overview of OSGi Application and Eclipse GlassFish](#)
- [Developing OSGi Application Bundles for Eclipse GlassFish](#)
- [Deploying OSGi Bundles in Eclipse GlassFish](#)

Overview of OSGi Application and Eclipse GlassFish

Eclipse GlassFish is fully-compliant with Jakarta EE 10, so it provides the latest Jakarta EE APIs and frameworks. It is built using OSGi technology, and includes as its OSGi module management subsystem the [Apache Felix OSGi framework](http://felix.apache.org) (<http://felix.apache.org>), which is a fully-compliant implementation of the OSGi Service Platform R4 Version 4.3 specification. Eclipse GlassFish supports deployment of OSGi-based applications using this framework. OSGi applications can make use of core as well as enterprise OSGi features. Eclipse GlassFish makes available many of its Jakarta EE platform services, such as the transaction service, HTTP service, JDBC Service and JMS, as OSGi services. It also enables use of Jakarta EE programming model in OSGi applications, so enterprise Java application developers can continue to leverage their existing skills in OSGi-based applications. See [Benefits of Using OSGi in Enterprise Java Applications](#) for more information.

OSGi applications are deployed as one or more OSGi bundles, and the Eclipse GlassFish deployment and administration infrastructure enables you to deploy and manage your OSGi bundles. This chapter classifies OSGi bundles into two categories based on the features they use:

- Plain OSGi Application Bundles - bundles that do not contain any Jakarta EE components. See [Developing Plain OSGi Bundles](#).
- Hybrid Application Bundles - bundles that are an OSGi bundle as well as a Jakarta EE module. At runtime, such modules have both an OSGi bundle context and a Jakarta EE context. Eclipse GlassFish supports the following hybrid application bundles:
 - Web Application Bundles (or WABs), see [Developing Web Application Bundles](#).
 - EJB Application Bundles, see [Developing EJB Application Bundles](#).

Benefits of Using OSGi in Enterprise Java Applications

Enterprise applications typically need transactional, secured access to data stores, messaging systems and other such enterprise information systems, and have to cater to a wide variety of clients such as web browsers and desktop applications, and so on. Jakarta EE makes development of such applications easier with a rich set of APIs and frameworks. It also provides a scalable, reliable and easy to administer runtime to host such applications.

The OSGi platform complements these features with modularity. It enables applications to be separated into smaller, reusable modules with a well defined and robust dependency specification.

A module explicitly specifies its capabilities and requirements. This explicit dependency specification encourages developers to visualize dependencies among their modules and help them make their modules highly cohesive and less coupled. The OSGi module system is dynamic: it allows modules to be added and removed at runtime. OSGi has very good support for versioning: it supports package versioning as well module versioning. In fact, it allows multiple versions of the same package to coexist in the same runtime, thus allowing greater flexibility to deployers. The service layer of the OSGi platform encourages a more service-oriented approach to build a system. The service-oriented approach and dynamic module system used together allow a system to be more agile during development as well as in production. It makes them better suited to run in an Platform-as-a-Service (PaaS) environment.

With Eclipse GlassFish, you do not have to choose one of the two platforms. A hybrid approach like OSGi enabling your Jakarta EE applications allows new capabilities to applications hitherto unavailable to applications built using just one of the two platforms.

Developing OSGi Application Bundles for Eclipse GlassFish

Eclipse GlassFish enables interaction between OSGi components and Jakarta EE components. OSGi services managed by the OSGi framework can invoke Jakarta EE components managed by the Jakarta EE container and vice versa. For example, developers can declaratively export EJBs as OSGi services without having to write any OSGi code. This allows any plain OSGi component, which is running without the Jakarta EE context, to discover the EJB and invoke it. Similarly, Jakarta EE components can locate OSGi services provided by plain OSGi bundles and use them as well. Eclipse GlassFish extends the Jakarta EE Context and Dependency Injection (CDI) framework to make it easier for Jakarta EE components to consume dynamic OSGi services in a type-safe manner.

- [Developing Plain OSGi Bundles](#)
- [Developing Web Application Bundles](#)
- [Developing EJB Application Bundles](#)

Developing Plain OSGi Bundles

Jakarta EE components (like an EJB or Servlet) can look up Jakarta EE platform services using JNDI names in the associated Jakarta EE naming context. Such code can rely on the Jakarta EE container to inject the required services as well. Unfortunately, neither of them works when the code runs outside a Jakarta EE context. An example of such code is the `BundleActivator` of an OSGi bundle. For such code to access Jakarta EE platform services, Eclipse GlassFish makes key services and resources of the underlying Jakarta EE platform available as OSGi services. Thus, an OSGi bundle deployed in Eclipse GlassFish can access these services using OSGi Service look-up APIs or by using a white board pattern. The following Jakarta EE services are available as OSGi services:

- [HTTP Service](#)
- [Transaction Service](#)
- [JDBC Data Source Service](#)
- [JMS Resource Service](#)

HTTP Service

The Eclipse GlassFish web container is made available as a service for OSGi users who do not use OSGi Web Application Bundles (WABs). This service is made available using the standard OSGi/HTTP service specification, which is a light API that predates the concept of a web application as we know it today. This simple API allows users to register servlets and static resources dynamically and draw a boundary around them in the form of a `HttpContext`. This simple API can be used to build feature-rich web application, such as the Felix Web Console for example.

The Eclipse GlassFish web container has one or more virtual servers. A virtual server has one or more web application deployed in it. Each web application has a distinct context path. Each virtual server has a set of HTTP listeners. Each HTTP listener listens on a particular port. When multiple virtual servers are present, one of them is treated as the default virtual server. Every virtual server comes configured with a default web application. The default web application is used to serve static content from the `docroot` of Eclipse GlassFish. This default web application uses `/` as the context path. A web application contains static and dynamic resources. Each virtual server is mapped to an `org.osgi.services.http.HttpService` instance. When there are multiple virtual servers present, there will be multiple occurrences of `HttpService` registered in the service registry. In order to distinguish one service from another, each service is registered with a service property named `VirtualServer`, whose value is the name of the virtual server. The service corresponding to default virtual server has the highest ranking, so when looking up a service of type `HttpService` without any additional criteria returns the `HttpService` corresponding to the default virtual server. In a typical Eclipse GlassFish installation, the default virtual server is configured to listen on port 8080 for the HTTP protocol and port 8181 for the HTTPS protocol.

The context path `/` is reserved for the default web application. Every resource and servlet registered using the `registerResource()` and `registerServlet()` methods of `HttpService` are made available under a special context path named `/osgi` in the virtual server. The `/osgi` context path can be changed to some other value by setting an appropriate value in the OSGi configuration property or in a system property called `org.glassfish.osgihttp.ContextPath`.

For example, `HelloWorldServlet` will be available at `http://localhost:8080/osgi/helloworld` when the following code is executed:

```
HttpService httpService = getHttpService(); // Obtain HttpService
httpService.registerServlet(httpService.registerServlet("/helloworld",
new HelloWorldServlet(), null, ctx);
```

Transaction Service

The Java Transaction API (JTA) defines three interfaces to interact with the transaction management system: `UserTransaction`, `TransactionManager`, and `TransactionSynchronizationRegistry`. They all belong to the `javax.transaction` package. `TransactionManager` and `TransactionSynchronizationRegistry` are intended for system level code, such as a persistence provider. Whereas, `UserTransaction` is the entity that you should use to control transactions. All the objects of the underlying JTA layer are made available in the OSGi service registry using the following service interfaces:

- `javax.transaction.UserTransaction`
- `javax.transaction.TransactionManager`
- `javax.transaction.TransactionSynchronisationRegistry`

There is no additional service property associated with them. Although `UserTransaction` appears to be a singleton, in reality any call to it gets rerouted to the actual transaction associated with the calling thread. Code that runs in the context of a Jakarta EE component typically gets a handle on `UserTransaction` by doing a JNDI lookup in the component naming context or by using injection, as shown here:

```
(UserTransaction)(new InitialContext().lookup("java:comp/UserTransaction"));
```

or

```
@Resource UserTransaction utx;
```

When certain code (such as an OSGi Bundle Activator), which does not have a Jakarta EE component context, wants to get hold of `UserTransaction`, or any of the other JTA artifacts, then they can look it up in the service registry. Here is an example of such code:

```
BundleContext context;
ServiceReference txRef =
    context.getServiceReference(UserTransaction.class.getName());
UserTransaction utx = (UserTransaction);
context.getService(txRef);
```

JDBC Data Source Service

Any JDBC data source created in Eclipse GlassFish is automatically made available as an OSGi Service; therefore, OSGi bundles can track availability of JDBC data sources using the `ServiceTracking` facility of the OSGi platform. The life of the OSGi service matches that of the underlying data source created in Eclipse GlassFish. For instructions on administering JDBC resources in Eclipse GlassFish, see the [Eclipse GlassFish Administration Guide](#).

Eclipse GlassFish registers each JDBC data source as an OSGi service with `objectClass = "javax.sql.DataSource"` and a service property called `jndi-name`, which is set to the JNDI name of the data source. Here is a code sample that looks up a data source service:

```
@Inject
@OSGiService(true, "(jndi-name=jdbc/MyDS)")
private DataSource ds;
```


JMS Resource Service

Like JDBC data sources, JMS administered objects, such as destinations and connection factories, are also automatically made available as OSGi services. Their service mappings are as follows.

JMS Object	Service Interface	Service Properties	Comments
JMS Queue destination	<code>jakarta.jms.Queue</code>	<code>jndi-name</code>	<code>jndi-name</code> is set to the JNDI name of the queue
JMS Topic destination	<code>jakarta.jms.Topic</code>	<code>jndi-name</code>	<code>jndi-name</code> is set to the JNDI name of the topic
JMS connection factory	<code>jakarta.jms.QueueConnectionFactory</code> or <code>jakarta.jms.TopicConnectionFactory</code> or <code>jakarta.jms.ConnectionFactory</code>	<code>jndi-name</code>	<code>jndi-name</code> is set to the JNDI name of the topic. The actual service interface depends on which type of connection factory was created.

Developing Web Application Bundles

When a web application is packaged and deployed as an OSGi bundle, it is called a Web Application Bundle (WAB). WAB support is based on the OSGi Web Application specification, which is part of the OSGi Service Platform, Enterprise Specification, Release 4, Version 4.3. A WAB is packaged as an OSGi bundle, so all the OSGi packaging rules apply to WAB packaging. When a WAB is not packaged like a WAR, the OSGi Web Container of Eclipse GlassFish maps the WAB to the hierarchical structure of web application using the following rules:

- The root of the WAB corresponds to the `docroot` of the web application.
- Every JAR in the Bundle-ClassPath of the WAB is treated like a JAR in `WEB-INF/lib/`.
- Every directory except "." in Bundle-ClassPath of the WAB is treated like `WEB-INF/classes/`.
- Bundle-ClassPath entry of type "." is treated as if the entire WAB is a JAR in `WEB-INF/lib/`.
- Bundle-ClassPath includes the Bundle-ClassPath entries of any attached fragment bundles.

The simplest way to avoid knowing these mapping rules is to avoid the problem in the first place. Moreover, there are many packaging tools and development time tools that understand WAR structure. Therefore, we strongly recommend that you package the WAB exactly like a WAR, with only additional OSGi metadata.

Required WAB Metadata

In addition to the standard OSGi metadata, the main attributes of `META-INF/MANIFEST.MF` of the WAB must have an additional attribute called `Web-ContextPath`. The `Web-ContextPath` attribute specifies the value of the context path of the web application. Since the root of a WAB is mapped to the `docroot` of the web application, it should not be used in the `Bundle-ClassPath`. Moreover, `WEB-INF/classes/`

should be specified ahead of `WEB-INF/lib/` in the `Bundle-ClassPath` in order to be compliant with the search order used for traditional WAR files.

Assuming the WAB is structured as follows:

```
foo.war/  
  index.html  
  foo.jsp  
  WEB-INF/classes/  
      foo/BarServlet.class  
  WEB-INF/lib/lib1.jar  
  WEB-INF/lib/lib2.jar
```

Then the OSGi metadata for the WAB as specified in `META-INF/MANIFEST.MF` of the WAB would appear as follows:

```
MANIFEST.MF:Manifest-Version: 1.0  
Bundle-ManifestVersion: 2  
Bundle-SymbolicName: com.acme.foo  
Bundle-Version: 1.0  
Bundle-Name: Foo Web Application Bundle Version 1.0  
Import-Package: javax.servlet; javax.servlet.http, version=[3.0, 4.0, 5.0)  
Bundle-ClassPath: WEB-INF/classes, WEB-INF/lib/lib1.jar, WEB-INF/lib/lib2.jar  
Web-ContextPath: /foo
```

How WABs Consume OSGi Services

Since a WAB has a valid `Bundle-Context`, it can consume OSGi services. Although you are free to use any OSGi API to locate OSGi services, Eclipse GlassFish makes it easy for WAB users to use OSGi services by virtue of extending the Context and Dependency Injection (CDI) framework. Here's an example of the injection of an OSGi Service into a Servlet:

```
@WebServlet  
public class MyServlet extends HttpServlet {  
    @Inject @OSGiService(dynamic=true)  
    FooService fooService;  
}
```

To learn more about this feature, refer to [OSGi CDI Extension for WABs](#).

OSGi CDI Extension for WABs

Eclipse GlassFish includes a CDI extension that enables web applications, such as servlets, that are part of WABs to express a type-safe dependency on an OSGi service using CDI APIs. An OSGi service can be provided by any OSGi bundle without any knowledge of Jakarta EE/CDI, and they are allowed to be injected transparently in a type-safe manner into a web application.

A custom CDI Qualifier, `@org.glassfish.osgi CDI.OSGiService`, is used by the component to represent dependency on an OSGi service. The qualifier has additional metadata to customize the service discovery and injection behavior. The following `@OSGiService` attributes are currently available:

- `serviceCriteria` — An LDAP filter query used for service selection in the OSGi service registry.
- `waitTimeout` — Waits the specified duration for a service that matches the criteria specified to appear in the OSGi service registry.
- `dynamic` — Dynamically obtain a service reference (true/false).

Since OSGi services are dynamic, they may not match the life cycle of the application component that has injected a reference to the service. Through this attribute, you could indicate that a service reference can be obtained dynamically or not. For stateless or idempotent services, a dynamic reference to a service implementation would be useful. The container then injects a proxy to the service and dynamically switches to an available implementation when the current service reference is invalid.

Example 13-1 Example of a WAB Using CDI

In this example, Bundle B0 defines a service contract called `com.acme.Foo` and exports the `com.acme` package for use by other bundles. Bundle B1 in turn provides a service implementation, `FooImpl`, of the `com.acme.Foo` interface. It then registers the service `FooImpl` service with the OSGi service registry with `com.acme.Foo` as the service interface.

Bundle B2 is a hybrid application bundle that imports the `com.acme` package. It has a component called `BarServlet` that expresses a dependency to `com.acme.Foo` by adding a field/setter method and qualifies that injection point with `@OSGiService`. For instance, `BarServlet` could look like:

```
@Servlet
public void BarServlet extends HttpServlet{
    @Inject @OSGiService(dynamic=true)
    private com.acme.Foo f;
}
```

Developing EJB Application Bundles

Another type of hybrid application bundle is the EJB Application Bundle. When an EJB Jar is packaged with additional OSGi metadata and deployed as an OSGi bundle it is called an EJB Application Bundle. Eclipse GlassFish supports only packaging the OSGi bundle as a simple JAR file with required OSGi metadata, just as you would package an `ejb-jar` file.

Required EJB Metadata

An EJB Application Bundle must have a manifest metadata called `Export-EJB` in order to be considered as an EJB Bundle. For syntax of `Export-EJB` header, please refer to the Publishing EJB as OSGi Service section. Here's an example of an EJB Application Bundle with its metadata:

```
myEjb.jar/
```

```
com/acme/Foo
com/acme/impl/FooEJB
META-INF/MANIFEST.MF
```

MANIFEST.MF:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-SymbolicName: com.acme.foo EJB bundle
Bundle-Version: 1.0.0.BETA
Bundle-Name: com.acme.foo EJB bundle version 1.0.0.BETA
Export-EJB: ALL
Export-Package: com.acme; version=1.0
Import-Package: javax.ejb; version=[3.0, 4.0), com.acme; version=[1.0, 1.1)
```

How EJB Bundles Consume OSGi Services

Since an EJB has a valid Bundle-Context, it can consume OSGi services. Although you are free to use any OSGi API to locate OSGi services, Eclipse GlassFish makes it easy to use OSGi services by virtue of extending the Context and Dependency Injection (CDI) framework. Here's an example of injection of an OSGi Service into a servlet:

```
@Stateless
public class MyEJB {
    @Inject @OSGiService(dynamic=true)
    Foo foo;
    ...
}
```

To learn more about this feature, refer to [Using the OSGi CDI Extension With EJB Bundles](#).

Using the OSGi CDI Extension With EJB Bundles

Eclipse GlassFish includes a CDI extension that enables EJB application bundles to express a type-safe dependency on an OSGi Service using CDI APIs. An OSGi service can be provided by any OSGi bundle without any knowledge of Jakarta EE/CDI, and they are allowed to be injected transparently in a type-safe manner into an EJB bundle.

A custom CDI Qualifier, `@org.glassfish.osgi CDI OSGiService`, is used by the component to represent dependency on an OSGi service. The qualifier has additional metadata to customize the service discovery and injection behavior. The following `@OSGiService` attributes are currently available:

- `dynamic` — Dynamically obtain a service reference (true/false).
- `waitTimeout` — Waits for specified duration for a service to appear in the OSGi service registry.
- `serviceCriteria` — An LDAP filter query used for service selection.

Deploying OSGi Bundles in Eclipse GlassFish

For instruction on deploying OSGi bundle, see " [OSGi Bundle Deployment Guidelines](#)" in Eclipse GlassFish Application Deployment Guide.

Part III

Chapter 20. Using Services and APIs

Chapter 21. Using the JDBC API for Database Access

This chapter describes how to use the Java Database Connectivity (JDBC) API for database access with the Eclipse GlassFish. This chapter also provides high level JDBC implementation instructions for servlets and EJB components using the Eclipse GlassFish.

The JDBC specifications are available at <https://www.oracle.com/java/technologies/javase/javase-tech-database.html>.

A useful JDBC tutorial is located at <https://docs.oracle.com/javase/tutorial/jdbc/index.html>.



The Eclipse GlassFish does not support connection pooling or transactions for an application's database access if it does not use standard Jakarta EE **DataSource** objects.

The following topics are addressed here:

- [Statements](#)
- [Connections](#)
- [Connection Wrapping](#)
- [Allowing Non-Component Callers](#)
- [Using Application-Scoped JDBC Resources](#)
- [Restrictions and Optimizations](#)

Statements

The following topics are addressed here:

- [Using an Initialization Statement](#)
- [Setting a Statement Timeout](#)
- [Statement Leak Detection and Leaked Statement Reclamation](#)
- [Statement Caching](#)
- [Statement Tracing](#)

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 [Eclipse GlassFish 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 [Eclipse GlassFish 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 [Eclipse GlassFish Reference Manual](#).

Statement Leak Detection and Leaked Statement Reclamation

If statements are not closed by an application after use, it is possible for the application to run out of cursors. Enabling statement leak detection causes statements to be considered as leaked if they are not closed within a specified period. Additionally, leaked statements can be reclaimed automatically.

To enable statement leak detection, set Statement Leak Timeout In Seconds for the JDBC connection pool to a positive, nonzero value in one of the following ways:

- Specify the `--statementleaktimeout` option in the `create-jdbc-connection-pool` subcommand. For more information, see `create-jdbc-connection-pool(1)`.
- Specify the `statement-leak-timeout-in-seconds` option in the `set` subcommand. For example:

```
asadmin set resources.jdbc-connection-pool.pool-name.statement-leak-timeout-in-seconds=300
```

When selecting a value for Statement Leak Timeout In Seconds, make sure that:

- It is less than the Connection Leak Timeout; otherwise, the connection could be closed before

the statement leak is recognized.

- It is greater than the Statement Timeout; otherwise, a long running query could be mistaken as a statement leak.

After enabling statement leak detection, enable leaked statement reclamation by setting Reclaim Leaked Statements for the JDBC connection pool to a `true` value in one of the following ways:

- Specify the `--statementleakreclaim=true` option in the `create-jdbc-connection-pool` subcommand. For more information, see `create-jdbc-connection-pool(1)`.
- Specify the `statement-leak-reclaim` option in the `set` subcommand. For example:

```
asadmin set resources.jdbc-connection-pool.pool-name.statement-leak-reclaim=true
```

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 Eclipse GlassFish 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 [Eclipse GlassFish 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 [Eclipse GlassFish 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 [Eclipse GlassFish 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 [Eclipse GlassFish Reference Manual](#).

The Eclipse GlassFish 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 Eclipse GlassFish, place the implementation classes in `as-install/lib`.

The Eclipse GlassFish 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 `jakarta.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|glassfish
6.0|jakarta.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.

When SQL statement tracing is enabled and JDBC connection pool monitoring is enabled, Eclipse GlassFish maintains a tracing cache of recent queries and their frequency of use. The following JDBC connection pool properties can be configured to control this cache and the monitoring statistics available from it:

time-to-keep-queries-in-minutes

Specifies how long in minutes to keep a query in the tracing cache, tracking its frequency of use. The default value is 5 minutes.

number-of-top-queries-to-report

Specifies how many of the most used queries, in frequency order, are listed the monitoring report. The default value is 10 queries.

Set these parameters in one of the following ways:

- Add them as properties in the Edit JDBC Connection Pool Properties page in the Administration Console. For more information, click the Help button in the Administration Console.
- Specify them using the `--property` option in the `create-jdbc-connection-pool` subcommand. For more information, see `create-jdbc-connection-pool(1)`.
- Set them using the `set` subcommand. For example:

```
asadmin set resources.jdbc-connection-pool.pool-name.property.time-to-keep-queries-in-minutes=10
```

Connections

The following topics are addressed here:

- [Transparent Pool Reconfiguration](#)
- [Disabling Pooling](#)
- [Associating Connections with Threads](#)
- [Custom Connection Validation](#)
- [Sharing Connections](#)
- [Marking Bad Connections](#)
- [Handling Invalid Connections](#)

Transparent Pool Reconfiguration

When the properties or attributes of a JDBC connection pool are changed, the connection pool is destroyed and re-created. Normally, applications using the connection pool must be redeployed as a consequence. This restriction can be avoided by enabling transparent JDBC connection pool reconfiguration. When this feature is enabled, applications do not need to be redeployed. Instead, requests for a new connections are blocked until the reconfiguration operation completes. Connection requests from any in-flight transactions are served using the old pool configuration so as to complete the transaction. Then, connections are created using the pool's new configuration, and any blocked connection requests are served with connections from the re-created pool..

To enable transparent JDBC connection pool reconfiguration, set the `dynamic-reconfiguration-wait-timeout-in-seconds` property of the JDBC connection pool to a positive, nonzero value in one of the following ways:

- Add it as a property in the Edit JDBC Connection Pool Properties page in the Administration Console. For more information, click the Help button in the Administration Console.
- Specify it using the `--property` option in the `create-jdbc-connection-pool` subcommand. For more information, see `create-jdbc-connection-pool(1)`.
- Set it using the `set` subcommand. For example:

```
asadmin set resources.jdbc-connection-pool.pool-name.property.dynamic-
```

```
reconfiguration-wait-timeout-in-seconds=15
```

This property specifies the time in seconds to wait for in-use connections to close and in-flight transactions to complete. Any connections in use or transaction in flight past this time must be retried.

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 [Eclipse GlassFish 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 [Eclipse GlassFish 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-count`
- `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 [Eclipse GlassFish 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 [Eclipse GlassFish 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 Eclipse GlassFish 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 [Eclipse GlassFish 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 [Eclipse GlassFish Reference Manual](#).

By default, optimized validation mechanisms are provided for DB2, Apache Derby, MSSQL, MySQL, Oracle, PostgreSQL and Sybase 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 Jakarta 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 "[Administering Database Connectivity](#)" in Eclipse GlassFish Administration Guide.

Marking Bad Connections

The `DataSource` implementation in the Eclipse GlassFish 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 Eclipse GlassFish 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 [Eclipse GlassFish Reference Manual](#).

The interface `ValidatingManagedConnectionFactory` exposes the method `getInvalidConnections` to allow retrieval of the invalid connections. The Eclipse GlassFish 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 topics are addressed here:

- [Wrapping Connections](#)
- [Obtaining a Physical Connection From a Wrapped Connection](#)
- [Using the `Connection.unwrap\(\)` Method](#)

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 [Eclipse GlassFish Reference Manual](#).

Obtaining a Physical Connection From a Wrapped Connection

The `DataSource` implementation in the Eclipse GlassFish provides a `getConnection` method that retrieves the JDBC driver's `SQLConnection` from the Eclipse GlassFish'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

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.

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 Jakarta 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 [Eclipse GlassFish 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 [Eclipse GlassFish 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](#).

Using Application-Scoped JDBC Resources

You can define an application-scoped database or other resource for an enterprise application, web module, EJB module, connector module, or application client module by supplying a `glassfish-resources.xml` deployment descriptor file. For details, see "[Application-Scoped Resources](#)" in Eclipse GlassFish Application Deployment Guide.

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 Oracle JDBC drivers for Sybase databases create a stored procedure for each parameterized `PreparedStatement`. On the Eclipse GlassFish, 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 22. Using the Transaction Service

The Jakarta EE platform provides several abstractions that simplify development of dependable transaction processing for applications. This chapter discusses Jakarta EE transactions and transaction support in the Eclipse GlassFish.

The following topics are addressed here:

- [Handling Transactions with Databases](#)
- [Handling Transactions with Enterprise Beans](#)
- [Handling Transactions with the Java Message Service](#)
- [The Transaction Manager, the Transaction Synchronization Registry, and `UserTransaction`](#)

For more information about the Java Transaction API (JTA), see "[Administering Transactions](#)" in Eclipse GlassFish Administration Guide and the following site: <https://jakarta.ee/specifications/transactions/>.

You might also want to read "[Transactions](#)" in The Jakarta EE Tutorial.

Handling Transactions with Databases

The following topics are addressed here:

- [Using JDBC Transaction Isolation Levels](#)
- [Using Non-Transactional Connections](#)

Using JDBC Transaction Isolation Levels

Not all database vendors support all transaction isolation levels available in the JDBC API. The Eclipse GlassFish permits specifying any isolation level your database supports. The following table defines transaction isolation levels.

Table 15-1 Transaction Isolation Levels

Transaction Isolation Level	<code>getTransactionIsolation</code> Return Value	Description
<code>read-uncommitted</code>	1	Dirty reads, non-repeatable reads, and phantom reads can occur.
<code>read-committed</code>	2	Dirty reads are prevented; non-repeatable reads and phantom reads can occur.
<code>repeatable-read</code>	4	Dirty reads and non-repeatable reads are prevented; phantom reads can occur.
<code>serializable</code>	8	Dirty reads, non-repeatable reads and phantom reads are prevented.

By default, the transaction isolation level is undefined (empty), and the JDBC driver's default isolation level is used. 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 [Eclipse GlassFish 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 [Eclipse GlassFish 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 "[To Create a JDBC Connection Pool](#)" in Eclipse GlassFish Administration Guide.

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.

Setting or resetting the transaction isolation level for every `getConnection` call can degrade performance. So by default the isolation level is not guaranteed.

Applications that change the transaction isolation level on a pooled connection programmatically risk polluting the JDBC connection pool, which can lead to errors. If an application changes the isolation level, enabling the `is-isolation-level-guaranteed` setting in the pool can minimize such errors.

You can guarantee the transaction isolation level in the following ways:

- Check the Isolation Level Guaranteed box 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 `--isolationguaranteed` option in the `asadmin create-jdbc-connection-pool` command. For more information, see the [Eclipse GlassFish Reference Manual](#).
- Specify the `is-isolation-level-guaranteed` option in the `asadmin set` command. For example:

```
asadmin set domain1.resources.jdbc-connection-pool.DerbyPool.is-isolation-level-guaranteed=true
```

For more information, see the [Eclipse GlassFish Reference Manual](#).

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 [Eclipse GlassFish 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 [Eclipse GlassFish Reference Manual](#).

- Use the `DataSource` implementation in the Eclipse GlassFish, 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.

Handling Transactions with Enterprise Beans

This section describes the transaction support built into the Enterprise JavaBeans programming model for the Eclipse GlassFish.

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.

The following topics are addressed here:

- [Flat Transactions](#)
- [Global and Local Transactions](#)
- [Commit Options](#)
- [Bean-Level Container-Managed Transaction Timeouts](#)

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

Both local and global transactions are demarcated using the `javax.transaction.UserTransaction` interface, which the client must use. Local transactions bypass the XA commit protocol and are faster. For more information, see [The Transaction Manager, the Transaction Synchronization Registry, and UserTransaction](#).

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.



Commit option A is not supported for this Eclipse GlassFish 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 `glassfish-ejb-jar.xml` 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.

Bean-Level Container-Managed Transaction Timeouts

The 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. The default value, `0`, specifies that the server waits indefinitely for a transaction to complete.

To override this timeout for an individual bean, use the optional `cmt-timeout-in-seconds` element in `glassfish-ejb-jar.xml`. The default value, `0`, specifies that the 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.

Handling Transactions with the Java Message Service

The following topics are addressed here:

- [Transactions and Non-Persistent Messages](#)
- [Using the ConfigurableTransactionSupport Interface](#)

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 Jakarta 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 Eclipse GlassFish 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.

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.

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](#).

If possible, you should use the `javax.transaction.TransactionSynchronizationRegistry` interface instead of `javax.transaction.TransactionManager`, for portability. You can look up the implementation of this interface by using the JNDI name `java:comp/''TransactionSynchronizationRegistry`. For details, see the `TransactionSynchronizationRegistryInterface` API documentation and [Java Specification Request \(JSR\) 907](#)

If accessing the `javax.transaction.TransactionManager` implementation is absolutely necessary, you can look up the Eclipse GlassFish implementation of this interface using the JNDI name `java:appserver/TransactionManager`. This lookup should not be used by the application code.

Chapter 23. Using the Java Naming and Directory Interface

A naming service maintains a set of bindings, which relate names to objects. The Jakarta 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 <https://docs.oracle.com/javase/tutorial/jndi/overview/index.html>. You can also see the JNDI tutorial at <https://docs.oracle.com/javase/jndi/tutorial/>.

The following topics are addressed here:

- [Accessing the Naming Context](#)
- [Configuring Resources](#)
- [Using a Custom `jndi.properties` File](#)
- [Mapping References](#)



The Web Profile of the Eclipse GlassFish supports the EJB 3.1 Lite specification, which allows enterprise beans within web applications, among other features. The full Eclipse GlassFish 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 Eclipse GlassFish provides a naming environment, or context, which is compliant with standard Jakarta 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 Jakarta 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 following topics are addressed here:

- [Portable Global JNDI Names](#)
- [Eclipse GlassFish V2 Vendor-Specific Global JNDI Names](#)
- [Disabling Eclipse GlassFish V2 JNDI Names](#)
- [Accessing EJB Components Using the `CosNaming` Naming Context](#)
- [Accessing EJB Components in a Remote Eclipse GlassFish](#)
- [Naming Environment for Lifecycle Modules](#)



Each resource within a server instance must have a unique name. However, two resources in different server instances or different domains can have the same name.

Portable Global JNDI Names

If an EJB component is a kind of session bean and it is deployed to any implementation supporting the EJB 3.1 specification (for example, Eclipse GlassFish 8), it automatically has one or more portable JNDI names defined based on the syntax in the specification. Note that this is true of existing EJB 3.0 and 2.x applications that are deployed to an implementation supporting EJB 3.1. No code changes are required to the bean class itself in order to have the portable global JNDI name automatically assigned when deployed to an EJB 3.1 container.

For more information, see the Jakarta EE Platform Specification, section EE.5.2.2, "Application Component Environment Namespaces" (<http://jcp.org/en/jsr/detail?id=366>), and the EJB 3.1 Specification, section 4.4, "Global JNDI Access" (<http://jcp.org/en/jsr/detail?id=318>).

If the `disable-nonportable-jndi-names` property is set to false (the default), a Eclipse GlassFish V2-specific JNDI name is assigned in addition to a portable global JNDI name. For more information, see [Eclipse GlassFish V2 Vendor-Specific Global JNDI Names](#) and [Disabling Eclipse GlassFish V2 JNDI Names](#).

Eclipse GlassFish V2 Vendor-Specific Global JNDI Names

Eclipse GlassFish v2 vendor-specific global JNDI names are assigned according to the following precedence rules:

1. A global JNDI name assigned in the `glassfish-ejb-jar.xml`, `glassfish-web.xml`, or `glassfish-application-client.xml` deployment descriptor file has the highest precedence. See [Mapping References](#).
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: `@jakarta.annotation.Resource`, `@javax.ejb.EJB`, `@javax.ejb.Stateless`, `@javax.ejb.Singleton`, `@javax.ejb.Stateful`, and `@javax.ejb.MessageDriven`.
4. In most cases, a default global JNDI name is assigned (and recorded in the server log) if no name is assigned in deployment descriptors or annotations.
 - For a session or entity bean, a Eclipse GlassFish V2-specific JNDI name is assigned as follows:
 - 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 Eclipse GlassFish V2-specific default. For an entity bean, a global JNDI name must be assigned.
- 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`.

Disabling Eclipse GlassFish V2 JNDI Names

The EJB 3.1 specification supported by Eclipse GlassFish 8 defines portable EJB JNDI names for session beans. Because of this, there is less need to continue to use older vendor-specific JNDI names.

By default, Eclipse GlassFish V2-specific JNDI names are applied automatically by Eclipse GlassFish 8 for backward compatibility. However, this can lead to some ease-of-use issues. For example, deploying two different applications containing a remote EJB component that exposes the same remote interface causes a conflict between the default JNDI names.

The default handling of V2-specific JNDI names in Eclipse GlassFish 8 can be managed by using the `asadmin` command:

```
asadmin> set server.ejb-container.property.disable-nonportable-jndi-names="true"
```

The `disable-nonportable-jndi-names` property is a boolean flag that can take the following values:

`false`

Enables the automatic use of Eclipse GlassFish V2-specific JNDI names in addition to portable global JNDI names. This is the default setting.

`true`

Disables the automatic use of V2-specific JNDI names. In all cases, only portable global JNDI names are used.

Note that this setting applies to all session beans deployed to the server.

Accessing EJB Components Using the `CosNaming` Naming Context

The preferred way of accessing the naming service, even in code that runs outside of a Jakarta 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 `org.glassfish.jndi.cosnaming.CNCtxFactory` in the client JVM software when accessing EJB components. You can set this property using the `asadmin create-jvm-options` command, as follows:

```
asadmin> create-jvm-options  
-Djava.naming.factory.initial=org.glassfish.jndi.cosnaming.CNCtxFactory
```

For details about `asadmin create-jvm-options`, see the [Eclipse GlassFish Reference Manual](#).

Or you can set this property in the code, as follows:

```
Properties properties = null;
try {
    properties = new Properties();
    properties.put("java.naming.factory.initial",
        "org.glassfish.jndi.cosnaming.CNCtxFactory");
    ...
}
...
```

The `java.naming.factory.initial` property applies to only one instance. The property is not cluster-aware.

Accessing EJB Components in a Remote Eclipse GlassFish

The recommended approach for looking up an EJB component in a remote Eclipse GlassFish 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 Eclipse GlassFish 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 Jakarta 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-ref` element to an interoperable name in a Eclipse GlassFish 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 `glassfish-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 Jakarta 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 rebound 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 Eclipse GlassFish 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 [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 Eclipse GlassFish exposes special resources in the naming environment.

- [External JNDI Resources](#)
- [Custom Resources](#)
- [Built-in Factories for Custom Resources](#)
- [Using Application-Scoped Resources](#)

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 [Eclipse GlassFish 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 [Eclipse GlassFish Reference Manual](#).

Built-in Factories for Custom Resources

The Eclipse GlassFish provides built-in factories for the following types of custom resources:

- [JavaBeanFactory](#)
- [PropertiesFactory](#)
- [PrimitivesAndStringFactory](#)
- [URLFactory](#)

Template `glassfish-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 `glassfish-resources.xml` file, see the [Eclipse GlassFish 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 Eclipse GlassFish.

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 `org.glassfish.resources.custom.factory.PropertiesFactory.fileName` 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 Eclipse GlassFish.

If an XML file is specified, it must match the document type definition (DTD) specified in the API definition of `java.util.Properties` (<http://docs.oracle.com/javase/8/docs/api/java/util/Properties.html>).

- Create the desired properties directly as properties of the custom resource.

If both the `fileName` 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`
 - `long`
 - `double`
 - `float`
 - `char`
 - `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.URLObjectFactory`.
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 Application-Scoped Resources

You can define an application-scoped JNDI or other resource for an enterprise application, web module, EJB module, connector module, or application client module by supplying a `glassfish-resources.xml` deployment descriptor file. For details, see "[Application-Scoped Resources](#)" in Eclipse GlassFish Application Deployment Guide.

Using a Custom `jndi.properties` File

To use a custom `jndi.properties` file, 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 [Class Loaders](#).

For each property found in more than one `jndi.properties` file, the Jakarta 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 Eclipse GlassFish deployment descriptors map resource references in application client, EJB, and web application components to JNDI names configured in Eclipse GlassFish:

- `resource-env-ref` - Maps the `@Resource` or `@Resources` annotation (or the `resource-env-ref` element in the corresponding Jakarta EE XML file) to the absolute JNDI name configured in Eclipse GlassFish.
- `resource-ref` - Maps the `@Resource` or `@Resources` annotation (or the `resource-ref` element in the corresponding Jakarta EE XML file) to the absolute JNDI name configured in Eclipse GlassFish.
- `ejb-ref` - Maps the `@EJB` annotation (or the `ejb-ref` element in the corresponding Jakarta EE XML file) to the absolute JNDI name configured in Eclipse GlassFish.

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 `glassfish-web.xml`, `glassfish-application-client.xml`, `glassfish-ejb-jar.xml`, and `glassfish-application.xml` deployment descriptor files. For more information about how these elements behave in each of the deployment descriptor files, see "[Elements of the Eclipse GlassFish Deployment Descriptors](#)" in Eclipse GlassFish 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, Jakarta Mail 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:jdbc/helloDbDs`. If this is the JNDI name of the JDBC resource configured in the Eclipse GlassFish, the annotation alone is enough to reference the resource.

However, you can use a Eclipse GlassFish specific deployment descriptor to override the annotation. For example, the `resource-ref` element in the `glassfish-web.xml` file maps the `res-ref-name` (the name specified in the annotation) to the JNDI name of another JDBC resource configured in Eclipse GlassFish.

```
<resource-ref>
  <res-ref-name>jdbc/helloDbDs</res-ref-name>
  <jndi-name>jdbc/helloDbDataSource</jndi-name>
</resource-ref>
```

Chapter 24. Using the Java Message Service

This chapter describes how to use the Java Message Service (JMS) API. The Eclipse GlassFish has a fully integrated JMS provider: the Open Message Queue software.



JMS resources are supported only in the full Eclipse GlassFish, not in the Web Profile.

For information about the JMS, see [Messaging](#) in The Jakarta EE Tutorial.

For detailed information about JMS concepts and JMS support in the Eclipse GlassFish, see "[Administering the Java Message Service \(JMS\)](#)" in Eclipse GlassFish Administration Guide.

The following topics are addressed here:

- [Using Application-Scoped JMS Resources](#)
- [Load-Balanced Message Inflow](#)
- [Authentication With `ConnectionFactory`](#)
- [Delivering SOAP Messages Using the JMS API](#)

Using Application-Scoped JMS Resources

You can define an application-scoped JMS or other resource for an enterprise application, web module, EJB module, connector module, or application client module by supplying a `glassfish-resources.xml` deployment descriptor file. For details, see "[Application-Scoped Resources](#)" in Eclipse GlassFish Application Deployment Guide.

Load-Balanced Message Inflow

You can configure `ActivationSpec` properties of the `jmsra` resource adapter in the `glassfish-ejb-jar.xml` file for a message-driven bean using `activation-config-property` elements. Whenever a message-driven bean (`EndpointFactory`) is deployed, the connector runtime engine finds these properties and configures them accordingly in the resource adapter. See "[activation-config-property](#)" in Eclipse GlassFish Application Deployment Guide.

The Eclipse GlassFish transparently enables messages to be delivered in random fashion to message-driven beans having same `ClientID`. The `ClientID` is required for durable subscribers.

For nondurable subscribers in which the `ClientID` is not configured, all instances of a specific message-driven bean that subscribe to same topic are considered equal. When a message-driven bean is deployed to multiple instances of the Eclipse GlassFish, only one of the message-driven beans receives the message. If multiple distinct message-driven beans subscribe to same topic, one instance of each message-driven bean receives a copy of the message.

To support multiple consumers using the same queue, set the `maxNumActiveConsumers` property of the physical destination to a large value. If this property is set, the Oracle Message Queue software allows multiple message-driven beans to consume messages from same queue. The message is

delivered randomly to the message-driven beans. If `maxNumActiveConsumers` is set to `-1`, there is no limit to the number of consumers.

To ensure that local delivery is preferred, set `addresslist-behavior` to `priority`. This setting specifies that the first broker in the `AddressList` is selected first. This first broker is the local colocated Message Queue instance. If this broker is unavailable, connection attempts are made to brokers in the order in which they are listed in the `AddressList`. This setting is the default for Eclipse GlassFish instances that belong to a cluster.

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 Eclipse GlassFish searches the container for authentication information before using the supplied user and password. Version 7 of the Eclipse GlassFish threw an exception in this situation.

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 topics are addressed here:

- [To Send SOAP Messages Using the JMS API](#)
- [To Receive SOAP Messages Using the JMS API](#)

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 GlassFish Web Services." );
/* Create an attachment 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 25. Using the Jakarta Mail API

This chapter describes how to use the Jakarta Mail API, which provides a set of abstract classes defining objects that comprise a mail system.

The following topics are addressed here:

- [Introducing Jakarta Mail](#)
- [Creating a Jakarta Mail Session](#)
- [Jakarta Mail Session Properties](#)
- [Looking Up a Jakarta Mail Session](#)
- [Sending and Reading Messages Using Jakarta Mail](#)
- [Using Application-Scoped Jakarta Mail Resources](#)



Jakarta Mail resources are supported only in the full Eclipse GlassFish, not in the Web Profile.

Introducing Jakarta Mail

The Jakarta Mail 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 Jakarta Mail API includes support for the IMAP4, POP3, and SMTP protocols.

The Jakarta Mail 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.
- Jakarta Mail uses the JavaBeans Activation Framework (JAF) to encapsulate message data and to handle commands intended to interact with that data.

For more information, see "[Administering the Jakarta Mail Service](#)" in Eclipse GlassFish Administration Guide and the Jakarta Mail specification at <https://jakarta.ee/specifications/mail>.

Creating a Jakarta Mail Session

You can create a Jakarta Mail session in the following ways:

- In the Administration Console, open the Resources component and select Jakarta Mail Sessions. For details, click the Help button in the Administration Console.
- Use the `asadmin create-mail-resource` command. For details, see the [Eclipse GlassFish Reference](#)

Jakarta Mail Session Properties

You can set properties for a Jakarta Mail `Session` object. Every property name must start with a `mail-` prefix. The Eclipse GlassFish changes the dash (-) character to a period (.) in the name of the property and saves the property to the `MailConfiguration` and Jakarta Mail `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 Jakarta Mail `Session` object, first define the property as follows:

- Name - `mail-from`
- Value - `john.doe@sun.com`

Looking Up a Jakarta Mail Session

The standard Java Naming and Directory Interface (JNDI) subcontext for Jakarta Mail sessions is `java:comp/env/mail`.

Registering Jakarta Mail 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 Jakarta Mail 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 Jakarta Mail 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 [Using the Java Naming and Directory Interface](#).

Sending and Reading Messages Using Jakarta Mail

The following topics are addressed here:

- [To Send a Message Using Jakarta Mail](#)
- [To Read a Message Using Jakarta Mail](#)

To Send a Message Using Jakarta Mail

1. Import the packages that you need.


```
import java.util.*;
import jakarta.activation.*;
import jakarta.mail.*;
import jakarta.mail.internet.*;
import javax.naming.*;
```

2. Look up the Jakarta Mail 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 Jakarta Mail Session](#).

3. Override the Jakarta Mail 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 Jakarta Mail

1. Import the packages that you need.

```
import java.util.*;
import jakarta.activation.*;
```

```
import jakarta.mail.*;
import jakarta.mail.internet.*;
import javax.naming.*;
```

2. Look up the Jakarta Mail session.

```
InitialContext ic = new InitialContext();
String snName = "java:comp/env/mail/MyMailSession";
Session session = (jakarta.mail.Session)ic.lookup(snName);
```

For more information, see [Looking Up a Jakarta Mail Session](#).

3. Override the Jakarta Mail 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();
```

Using Application-Scoped Jakarta Mail Resources

You can define an application-scoped Jakarta Mail or other resource for an enterprise application, web module, EJB module, connector module, or application client module by supplying a **glassfish-resources.xml** deployment descriptor file. For details, see "[Application-Scoped Resources](#)" in Eclipse GlassFish Application Deployment Guide.