GlassFish v3 Application Server Developer's Guide

Technology Preview 2

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 820–4496–06
May 2008

080530@20209

Contents

Tables

Figures

Preface

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

This preface contains information about and conventions for the entire GlassFish™ Application Server documentation set.

Application Server Documentation Set

The Application Server documentation set describes deployment planning and system installation. The Uniform Resource Locator (URL) for Application Server documentation is `http://docs.sun.com/coll/1343.7`. For an introduction to Application Server, refer to the books in the order in which they are listed in the following table.

TABLE P–1 Books in the Application Server Documentation Set

Book Title	Description
Release Notes	Provides late-breaking information about the software and the documentation. Includes a comprehensive, table-based summary of the supported hardware, operating system, Java Development Kit (JDK™), and database drivers.
Quick Start Guide	Explains how to get started with the Application Server product.
Installation Guide	Explains how to install the software and its components.
Application Deployment Guide	Explains how to assemble and deploy applications to the Application Server and provides information about deployment descriptors.

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

Book Title	Description
Developer's Guide	Explains how to create and implement Java Platform, Enterprise Edition (Java EE platform) applications that are intended to run on the Application Server. These applications follow the open Java standards model for Java EE components and APIs. This guide provides information about developer tools, security, debugging, and creating lifecycle modules.
Java EE 5 Tutorial	Explains how to use Java EE 5 platform technologies and APIs to develop Java EE applications.
Java WSIT Tutorial	Explains how to develop web applications by using the Web Service Interoperability Technologies (WSIT). The tutorial focuses on developing web service endpoints and clients that can interoperate with Windows Communication Foundation (WCF) endpoints and clients.
Administration Guide	Explains how to configure and manage Application Server subsystems and components from the command line by using the `asadmin`(1M) utility. Instructions for performing these tasks from the Admin Console are provided in the Admin Console online help.
RESTful Web Services Developer's Guide	Explains how to develop Representational State Transfer (RESTful) web services for Application Server.
Getting Started With JRuby on Rails for the GlassFish Application Server	Explains how to develop Ruby on Rails applications for deployment to Application Server.
Getting Started With Project jMaki for the GlassFish Application Server	Explains how to use the jMaki framework to develop Ajax-enabled web applications that are centered on JavaScript™ technology for deployment to Application Server.
Reference Manual	Provides reference information in man page format for Application Server administration commands, utility commands, and related concepts.

Related Documentation

A Javadoc™ tool reference for packages that are provided with the Application Server is located at `http://glassfish.dev.java.net/nonav/javaee5/api/index.html`. Additionally, the following resources might be useful:

- The Java EE 5 Specifications (`http://java.sun.com/javaee/5/javatech.html`)
- The Java EE Blueprints (`http://java.sun.com/reference/blueprints/index.html`)

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

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

The GlassFish Samples project is a collection of sample applications that demonstrate a broad range of Java EE technologies. The GlassFish Samples are bundled with the Java EE Software Development Kit (SDK), and are also available from the GlassFish Samples project page at `https://glassfish-samples.dev.java.net/`.

Default Paths and File Names

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

TABLE P–2 Default Paths and File Names

Placeholder	Description	Default Value
as-install	Represents the base installation directory for Application Server.	Installations on the Solaris™ operating system and Linux operating system: *user's-home-directory*`/glassfish-v3tp2/glassfish` Windows, all installations: *SystemDrive*`:\Program Files\glassfish-v3tp2\glassfish`
domain-root-dir	Represents the directory containing all domains.	*as-install*`/domains/`
domain-dir	Represents the directory for a domain. In configuration files, you might see *domain-dir* represented as follows: `${com.sun.aas.instanceRoot}`	*domain-root-dir*`/`*domain-name*
instance-dir	Represents the directory for a server instance.	*domain-dir*`/`*instance-name*

Typographic Conventions

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

TABLE P–3 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`

TABLE P–3 Typographic Conventions *(Continued)*

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

Symbol Conventions

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

TABLE P–4 Symbol Conventions

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

Documentation, Support, and Training

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

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

Searching Sun Product Documentation

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

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

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

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

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

Third-Party Web Site References

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

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

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. To share your comments, go to `http://docs.sun.com` and click Send Comments. In the online form, provide the full document title and part number. The part number is a 7-digit or 9-digit number that can be found on the book's title page or in the document's URL. For example, the part number of this book is 820-4496.

PART I

Development Tasks and Tools

◆ ◆ ◆ CHAPTER 1

Setting Up a Development Environment

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

Installing and Preparing the Server for Development

The following components are included in the full installation.

- JDK
- Application Server core
  - Java 2 Platform, Standard Edition (J2SE™) 5
  - Java SE 5 compliant application server
  - Admin Console
  - `asadmin` utility
  - Other development and deployment tools
  - Java DB database, based on the Derby database from Apache (`http://db.apache.org/derby/manuals`)

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

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

- Locate utility classes and libraries so they can be accessed by the proper class loaders. For more information, see "Using the Application Server Parent Class Loader" on page 29.
- Set up debugging. For more information, see Chapter 3, "Debugging Applications."
- Configure the Java Virtual Machine (JVM™) software. For more information, see Chapter 3, "Administering the Java Virtual Machine (JVM)," in *GlassFish v3 Application Server Administration Guide*.

The GlassFish Project

Application Server v3 Technology Preview 2 is developed through the GlassFish℠ project open-source community at `https://glassfish.dev.java.net/`. The GlassFish project provides a structured process for developing the Application Server platform that makes the new features of Java EE 5 available faster, while maintaining the most important feature of Java EE: compatibility. It enables Java developers to access the Application Server source code and to contribute to the development of the Application Server. The GlassFish project is designed to encourage communication between Sun engineers and the community.

The Java ES edition of the Application Server is based on the GlassFish source code, but provides additional value-added features such as access to a high-availability database (HADB) for session persistence and failover.

Development Tools

The following general tools are provided with the Application Server:

- "The `asadmin` Command" on page 23
- "The Admin Console" on page 23

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

- "The NetBeans IDE" on page 23
- "The Migration Tool" on page 23

The following third-party tools might also be useful:

- "Debugging Tools" on page 24
- "Profiling Tools" on page 24
- "The Eclipse IDE" on page 24

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 *GlassFish v3 Application Server Reference Manual*.

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

The Admin Console

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

To access the Admin Console, type `http://`*host*`:4848` (developer profile) or `https://`*host*`:4848` (cluster and enterprise profiles) in your browser. The *host* is the name of the machine on which the Application Server is running. By default, the *host* is `localhost`. For example:

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

The NetBeans IDE

The NetBeans IDE allows you to create, assemble, and debug code from a single, easy-to-use interface. The GlassFish edition of the Application Server is bundled with the NetBeans 5.5 IDE. To download the NetBeans IDE, see `http://www.netbeans.org`. This site also provides documentation on how to use the NetBeans IDE with the bundled Application Server.

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

The Migration Tool

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

Debugging Tools

You can use several debugging tools with the Application Server. For more information, see Chapter 3, "Debugging Applications."

Profiling Tools

You can use several profilers with the Application Server. For more information, see "Profiling Tools" on page 34.

The Eclipse IDE

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

Sample Applications

Sample applications that you can examine and deploy to the Application Server are available. If you installed the Application Server as part of installing the Java EE 5 SDK bundle from Java EE 5 Downloads (`http://java.sun.com/javaee/5/downloads/`), the samples may already be installed. You can download these samples separately from the Code Samples (`http://java.sun.com/javaee/reference/code/index.jsp`) page if you installed the Application Server without them initially.

Most Application Server samples have the following directory structure:

- The `docs` directory contains instructions for how to use the sample.
- The `src/java` directory under each component contains source code for the sample.
- The `src/conf` directory under each component contains the deployment descriptors.

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

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

In addition to the Java EE 5 sample applications, samples are also available on the GlassFish web site at `https://glassfish-samples.dev.java.net/`.

◆ ◆ ◆ CHAPTER 2

Class Loaders

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

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

Note – For GlassFish v3 Technology Preview 2, EJB modules are not supported unless the optional EJB container module is downloaded from the Update Center. Web services are not supported unless the optional Metro module is downloaded from the Update Center. For information about the Update Center, see the *GlassFish v3 Application Server Quick Start Guide*.

The Class Loader Hierarchy

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

TABLE 2–1 GlassFish Application Server Class Loaders

Class Loader	Description
Application Server Parent	The Application Server Parent class loader loads classes in the *domain-dir*`/lib/classes` directory, followed by JAR files in the *domain-dir*`/lib` directory. It is parent to the Application Parent class loader. No special classpath settings are required. See "Using the Application Server Parent Class Loader" on page 29.
Application Parent	The Application Parent class loader loads the classes in and needed by a specific enabled individually deployed module. One instance of this class loader is present in each class loader universe; see "Class Loader Universes" on page 28. The Application Parent class loader is created with a list of URLs that point to the locations of the classes it needs to load. It is parent to the Web class loader. The Application Parent class loader loads classes in the following order: 1. Classes specified by the `--libraries` option during deployment; see "Application-Specific Class Loading" on page 28 2. Classes specified by the module's `location` attribute in the `domain.xml` file, determined during deployment 3. Classes in the module's stubs directory The `location` attribute points to *domain-dir*`/applications/`*module-name*. The stubs directory is *domain-dir*`/generated/ejb/`*module-name*.
Web	The Web class loader loads the servlets and other classes in a specific enabled web module or a Java EE application that contains a web module. This class loader is present in each class loader universe that contains a web module; see "Class Loader Universes" on page 28. One instance is created for each web module. The Web class loader is created with a list of URLs that point to the locations of the classes it needs to load. The classes it loads are in `WEB-INF/classes` or `WEB-INF/lib/*.jar`. It is parent to the JSP Engine class loader.
JSP Engine	The JSP Engine class loader loads compiled JSP classes of enabled JSP files. This class loader is present in each class loader universe that contains a JSP page; see "Class Loader Universes" on page 28. The JSP Engine class loader is created with a list of URLs that point to the locations of the classes it needs to load.

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 classloading to its parent before attempting to load a class itself. A class loader parent can be either the System class loader or another custom class loader. 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 the Web class loader look in the local class loader before delegating to its parent. You can make the Web class loader follow the delegation inversion model in the Servlet specification by setting `delegate="false"` in the `class-loader` element of the `sun-web.xml` file. It is safe to do this only for a web module that does not interact with any other modules. For details, see "class-loader" in *GlassFish v3 Application Server Application Deployment Guide*.

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

Note – For Technology Preview 2, the `delegate` value is ignored and assumed to be set to `true`.

Using the Java Optional Package Mechanism

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

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

For more information, see Optional Packages - An Overview (`http://java.sun.com/j2se/1.5.0/docs/guide/extensions/extensions.html`) and Understanding Extension Class Loading (`http://java.sun.com/docs/books/tutorial/ext/basics/load.html`).

Using the Endorsed Standards Override Mechanism

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

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

For more information and the list of packages that can be overridden, see Endorsed Standards Override Mechanism (`http://java.sun.com/j2se/1.5.0/docs/guide/standards/`).

Class Loader Universes

Access to components within modules installed on the server occurs within the context of isolated class loader universes, each of which has its own Application Parent, Web, and JSP Engine class loaders.

- **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 class loader's classpath; for example, the web class loader's classpath includes these directories:

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

Application-Specific Class Loading

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

- Use the Admin Console. Open the Applications component, then go to the page for the type of module. Select the Deploy button. Type the comma-separated paths in the Libraries field. For details, click the Help button in the Admin Console.
- Use the `asadmin deploy` command with the `--libraries` option and specify comma-separated paths. For details, see the *GlassFish v3 Application Server Reference Manual*.

Application libraries are included in the Application 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).

If multiple 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, modules using application-specific libraries are not portable. Other ways to make libraries available are described in "Circumventing Class Loader Isolation" on page 29.

For general information about deployment, see the *GlassFish v3 Application Server Application Deployment Guide*.

Note – If you see an access control error message when you try to use a library, you may need to grant permission to the library in the `server.policy` file. For more information, see "Changing Permissions for an Application" on page 46.

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 from interfering with each other.

To circumvent this limitation for libraries, utility classes, or individually deployed modules, you can include the relevant path to the required classes. See "Using the Application Server Parent Class Loader" on page 29.

Using the Application Server Parent Class Loader

To use the Application Server Parent class loader, copy the JAR files into the *domain-dir*`/lib` directory or copy the `.class` files into the *domain-dir*`/lib/classes` directory, then restart the server.

Using the Application Server Parent class loader makes a module accessible to all modules deployed on servers that share the same configuration.

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

◆ ◆ ◆ CHAPTER 3

Debugging Applications

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

Enabling Debugging

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

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

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

```
jdb -attach 9009
```

For Windows:

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

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

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

Application Server debugging is based on the JPDA. For more information, see "JPDA Options" on page 32.

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

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

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

1. **Use the Admin Console. In the developer profile, select the Application Server component and the JVM Settings tab. In the cluster profile, 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 to a debugger, specify** `address=`*port-number* **in the Debug Options field.**
4. **To add JPDA options, add any desired JPDA debugging options in Debug Options. See "JPDA Options" on page 32.**

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

JPDA Options

The default JPDA options in Application Server are as follows:

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

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

If you substitute `suspend=y`, the JVM 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 starts.

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

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

Generating a Stack Trace for Debugging

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

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 --user adminuser --verbose [domain-name]
```

On Windows platforms, you must perform an extra preparation step if you want to use Ctrl-Break to print a thread dump. In the *as-install*`/asenv.bat` file, change `AS_NATIVE_LAUNCHER="false"` to `AS_NATIVE_LAUNCHER="true"`.

When the server is in verbose mode, messages are logged to the console or terminal window in addition to the log file. In addition, pressing Ctrl-C stops the server and pressing Ctrl-\ (on UNIX platforms) or Ctrl-Break (on Windows platforms) prints a thread dump. On UNIX platforms, you can also print a thread dump using the `jstack` command (see `http://java.sun.com/j2se/1.5.0/docs/tooldocs/share/jstack.html`) or the command `kill -QUIT` *process_id*.

Application Server Logging

You can use the Application Server's log files to help debug your applications. Use the Admin Console. In the developer profile, select the Application Server component. In the cluster profile, select the Stand-Alone Instances component, and select the instance from the table. Then click the View Log Files button in the General Information page.

To change logging settings in the developer profile, select the Logging tab. In the cluster profile, select Logger Settings under the relevant configuration.

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

Profiling Tools

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

Information about comprehensive monitoring and management support in the Java™ 2 Platform, Standard Edition (J2SE™ platform) is available at `http://java.sun.com/j2se/1.5.0/docs/guide/management/index.html`.

The NetBeans Profiler

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

The HPROF Profiler

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

HPROF can monitor CPU usage, heap allocation statistics, and contention profiles. In addition, it can also report complete heap dumps and states of all the monitors and threads in the Java virtual machine. For more details on the HPROF profiler, see the JDK documentation at `http://java.sun.com/j2se/1.5.0/docs/guide/jvmpi/jvmpi.html#hprof`.

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 Admin Console. In the developer profile, select the Application Server component and the JVM Settings tab. In the cluster profile, 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
  ```

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

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

3 **Restart the Application Server.**

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

The JProbe Profiler

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

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

▼ To Enable Remote Profiling With JProbe

1 **Install JProbe 3.0.1.1.**

For details, see the JProbe documentation.

2 **Configure Application Server using the Admin Console:**

a. **In the developer profile, select the Application Server component and the JVM Settings tab. In the cluster profile, select the JVM Settings component under the relevant configuration. Then select the Profiler tab.**

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

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

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

  `-Xrunjprobeagent`

  `-Xnoclassgc`

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

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

3 **Set the following environment variable:**

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

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

4 **Start the server 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:

a. Select Server Side for the type of application.

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

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

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

PART II

Developing Applications and Application Components

◆ ◆ ◆ CHAPTER 4

Securing Applications

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

For information about administrative security for the Application Server, see Chapter 5, "Administering System Security," in *GlassFish v3 Application Server Administration Guide*.

For general information about Java EE security, see "Chapter 29: Introduction to Security in Java EE" in the Java EE 5 Tutorial (`http://java.sun.com/javaee/5/docs/tutorial/doc/index.html`).

This chapter contains the following sections:

Security Goals

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

- Support for several underlying authentication realms.
- Support for declarative security through Application Server specific XML-based role mapping.

- Support for Java Authorization Contract for Containers (JACC) pluggable authorization as included in the Java EE specification and defined by Java Specification Request (JSR) 115 (`http://www.jcp.org/en/jsr/detail?id=115`).
- Support for Java™ Authentication Service Provider Interface for Containers as included in the Java EE specification and defined by JSR 196 (`http://www.jcp.org/en/jsr/detail?id=196`).
- Support for Web Services Interoperability Technologies (WSIT) as described in The WSIT Tutorial (`https://wsit-docs.dev.java.net/releases/m5/`).

Container Security

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

- "Declarative Security" on page 42
- "Programmatic Security" on page 43

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

Declarative Security

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

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

There are two levels of declarative security, as follows:

- "Application Level Security" on page 42
- "Component Level Security" on page 43

Application Level Security

For an application, roles used by any application container must be defined in `@DeclareRoles` annotations in the code or `role-name` elements in the application deployment descriptor (`application.xml`). The role names are scoped to the EJB XML deployment descriptors

(`ejb-jar.xml` and `sun-ejb-jar.xml` files) and to the servlet XML deployment descriptors (`web.xml` and `sun-web.xml` files). For an individually deployed web or EJB module, you define roles using `@DeclareRoles` annotations or `role-name` elements in the Java EE deployment descriptor files `web.xml` or `ejb-jar.xml`.

To map roles to principals and groups, define matching `security-role-mapping` elements in the `sun-application.xml`, `sun-ejb-jar.xml`, or `sun-web.xml` file for each `role-name` used by the application. For more information, see "Roles, Principals, and Principal to Role Mapping" on page 43.

Component Level Security

Component level security encompasses web components and EJB components.

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

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

Programmatic Security

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

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

Roles, Principals, and Principal to Role Mapping

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

For more information regarding Java EE deployment descriptors, see the Java EE Specification. For more information regarding Application Server deployment descriptors, see Appendix A, "Deployment Descriptor Files," in *GlassFish v3 Application Server Application Deployment Guide*.

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

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

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

You can specify a default principal and a default principal to role mapping, each of which applies to the entire Application Server 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 Admin Console, select the Security component under the relevant configuration. For details, click the Help button in the Admin Console.

Realm Configuration

This section covers the following topics:

Supported Realms

The following realms are supported in the Application Server:

- `file` – Stores user information in a file. This is the default realm when you first install the Application Server.
- `jdbc` – Stores user information in a database.

  In the JDBC realm, the server gets user credentials from a database. The Application Server uses the database information and the enabled JDBC realm option in the configuration file. For digest authentication, a JDBC realm should be created with `jdbcDigestRealm` as the JAAS context.

For information about configuring realms, see "How to Configure a Realm" on page 45.

How to Configure a Realm

You can configure a realm in one of these ways:

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

How to Set a Realm for a Web Application

The `web.xml` deployment descriptor has an optional `realm-name` data subelement that overrides the domain's default realm.

For more information about the deployment descriptor files and elements, see Appendix A, "Deployment Descriptor Files," in *GlassFish v3 Application Server Application Deployment Guide*.

JACC Support

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

The Application Server provides a simple file-based JACC-compliant authorization engine as a default JACC provider. To configure an alternate provider using the Admin Console, open the Security component under the relevant configuration, and select the JACC Providers component. For details, click the Help button in the Admin Console.

The server.policy File

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

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

This section covers the following topics:

Default Permissions

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

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

A few permissions above the minimal set are also granted in the default server.policy file. These are necessary due to various internal dependencies of the server implementation. Java EE application developers must not rely on these additional permissions. In some cases, deleting these permissions might be appropriate. For example, one additional permission is granted specifically for using connectors. If connectors are not used in a particular domain, you should remove this permission, because it is not otherwise necessary.

Changing Permissions for an Application

The default policy for each domain limits the permissions of Java EE deployed applications to the minimal set of permissions required for these applications to operate correctly. Do not add

extra permissions to the default set (the grant block with no codebase, which applies to all code). Instead, add a new grant block with a codebase specific to the applications requiring the extra permissions, and only add the minimally necessary permissions in that block.

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

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

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

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

For a module:

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

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

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

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

Granting the following permission eliminates the message:

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

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

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

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

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

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

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

You can use the J2SE standard `policytool` or any text editor to edit the `server.policy` file. For more information, see
`http://java.sun.com/docs/books/tutorial/security1.2/tour2/index.html`.

For detailed information about policy file syntax, see
`http://java.sun.com/j2se/1.5.0/docs/guide/security/PolicyFiles.html#FileSyntax`.

For information about using system properties in the `server.policy` file, see
`http://java.sun.com/j2se/1.5.0/docs/guide/security/PolicyFiles.html#PropertyExp`..

For detailed information about the permissions you can set in the `server.policy` file, see
`http://java.sun.com/j2se/1.5.0/docs/guide/security/permissions.html`.

The Javadoc for the `Permission` class is at
`http://java.sun.com/j2se/1.5.0/docs/api/java/security/Permission.html`.

Enabling and Disabling the Security Manager

The security manager is disabled by default.

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

- Performance is critical
- Deployment to the production server is carefully controlled
- Only trusted applications are deployed
- Applications don't need policy enforcement

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

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

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

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

◆ ◆ ◆ CHAPTER 5

Developing Web Services

This chapter describes Application Server support for web services. Java™ API for XML-Based Web Services (JAX-WS) version 2.0 is supported. Java API for XML-Based Remote Procedure Calls (JAX-RPC) version 1.1 is supported for backward compatibility. This chapter contains the following sections:

Note – For GlassFish v3 Technology Preview 2, web services are not supported unless the optional Metro module is downloaded from the Update Center. Without the Metro module, a servlet cannot be a web service endpoint, and the `sun-web.xml` elements related to web services are ignored. For information about the Update Center, see the *GlassFish v3 Application Server Quick Start Guide*.

"Part Two: Web Services" in the Java EE 5 Tutorial (`http://java.sun.com/javaee/5/docs/tutorial/doc/index.html`) shows how to deploy simple web services to the Application Server. "Chapter 20: Java API for XML Registries" explains how to set up a registry and create clients that access the registry.

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

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

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

Creating Portable Web Service Artifacts

For a tutorial that shows how to use the `wsimport` and `wsgen` commands, see "Part Two: Web Services" in the Java EE 5 Tutorial (`http://java.sun.com/javaee/5/docs/tutorial/doc/index.html`). For reference information on these commands, see the *GlassFish v3 Application Server Reference Manual*.

Deploying a Web Service

You deploy a web service endpoint to the Application Server just as you would any servlet. After you deploy the web service, the next step is to publish it. For more information about publishing a web service, see "Web Services Registry" on page 53.

You can use the autodeployment feature to deploy a simple JSR 181 (`http://jcp.org/en/jsr/detail?id=181`) annotated file. You can compile and deploy in one step, as in the following example:

```
javac -cp javaee.jar -d domain-dir/autodeploy MyWSDemo.java
```

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

The Sun-specific deployment descriptor file `sun-web.xml` provides optional web service enhancements in the `webservice-endpoint` and `webservice-description` elements, including a `debugging-enabled` subelement that enables the creation of a test page. The test page feature is enabled by default and described in "The Web Service URI, WSDL File, and Test Page" on page 54.

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

Web Services Registry

You deploy a registry to the Application Server just as you would any module, except that if you are using the Admin Console, you must select a Registry Type value. After deployment, you can configure a registry in one of the following ways:

- In the Admin Console, open the Web Services component, and select the Registry tab. For details, click the Help button in the Admin Console.
- To configure a registry using the command line, use the following commands.
  - Set the registry type to `com.sun.appserv.registry.ebxml` or `com.sun.appserv.registry.uddi`. Use a backslash before each period as an escape character. For example:

    ```
    asadmin create-resource-adapter-config --user adminuser
    --property com\.sun\.appserv\.registry\.ebxml=true MyReg
    ```

  - Set any properties needed by the registry. For an ebXML registry, set the `LifeCycleManagerURL` and `QueryManagerURL` properties. In the following example, the system property `REG_URL` is set to `http\\:\\/\\/siroe.com\\:6789\\/soar\\/registry\\/soap`.

    ```
    asadmin create-connector-connection-pool --user adminuser --raname MyReg
    --connectiondefinition javax.xml.registry.ConnectionFactory --property
    LifeCycleManagerURL=${REG_URL}:QueryManagerURL=${REG_URL} MyRegCP
    ```

  - Set a JNDI name for the registry resource. For example:

    ```
    asadmin create-connector-resource --user adminuser --poolname MyRegCP jndi-MyReg
    ```

  For details on these commands, see the *GlassFish v3 Application Server Reference Manual*.

After you deploy a web service, you can publish it to a registry in one of the following ways:

- In the Admin Console, open the Web Services component, select the web service in the listing on the General tab, and select the Publish tab. For details, click the Help button in the Admin Console.
- Use the `asadmin publish-to-registry` command. For example:

```
asadmin publish-to-registry --user adminuser --registryjndinames jndi-MyReg --webservicename my-ws#simple
```

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

The Sun Java Enterprise System (Java ES) includes a Sun-specific ebXML registry. For more information about the Java ES registry and registries in general, see "Chapter 20: Java API for XML Registries" in the Java EE 5 Tutorial (`http://java.sun.com/javaee/5/docs/tutorial/doc/index.html`).

A module that accesses UDDI registries is provided with the Application Server in the *as-install*`/lib/install/applications/jaxr-ra` directory.

You can also use the JWSDP registry available at `http://java.sun.com/webservices/jwsdp/index.jsp` or the SOA registry available at `http://www.sun.com/products/soa/registry/index.html`.

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

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

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

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

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

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

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

You can also test a service using the Admin 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 Admin Console.

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

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

Using the Woodstox Parser

The default XML parser in the Application Server is the GlassFish XML Parser (SJSXP). Using the Woodstox parser, which is bundled with the Application Server, may improve performance. Woodstox and SJSXP both provide implementations of the StAX API. To enable the Woodstox parser, set the following system properties for the default `server-config` configuration in the `domain.xml` file, then restart the server:

```
<config name=server-config>
 ...
   <system-property name="javax.xml.stream.XMLEventFactory"
      value="com.ctc.wstx.stax.WstxEventFactory"/>
   <system-property name="javax.xml.stream.XMLInputFactory"
      value="com.ctc.wstx.stax.WstxInputFactory"/>
   <system-property name="javax.xml.stream.XMLOutputFactory"
      value="com.ctc.wstx.stax.WstxOutputFactory"/>
</config>
```

In addition, set these properties for any other configurations referenced by server instances or clusters on which you want to use the Woodstox parser.

Note – If you are using a stand-alone client, you must set these same properties for the client on the `java` command line as follows:

```
-Djavax.xml.stream.XMLInputFactory=com.ctc.wstx.stax.WstxInputFactory
-Djavax.xml.stream.XMLOutputFactory=com.ctc.wstx.stax.WstxOutputFactory
-Djavax.xml.stream.XMLEventFactory=com.ctc.wstx.stax.WstxEventFactory
```

Setting these properties is not necessary if you are using an application client, which is recommended and supported.

For more information about the Woodstox parser, see `http://woodstox.codehaus.org/`. For more information about the StAX API, see *Chapter 17: Streaming API for XML* in the Java EE 5 Tutorial (`http://java.sun.com/javaee/5/docs/tutorial/doc/index.html`).

◆ ◆ ◆ CHAPTER 6

Using the Java Persistence API

GlassFish Application Server support for the Java Persistence API includes all required features described in the Java Persistence Specification. Although officially part of the Enterprise JavaBeans Specification v3.0, also known as JSR 220 (`http://jcp.org/en/jsr/detail?id=220`), the Java Persistence API can also be used with non-EJB components outside the EJB container.

The Java Persistence API provides an object/relational mapping facility to Java developers for managing relational data in Java applications. For basic information about the Java Persistence API, see "Part Four: Persistence" in the Java EE 5 Tutorial (`http://java.sun.com/javaee/5/docs/tutorial/doc/index.html`).

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

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

Specifying the Database

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

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

If you are using the default persistence provider, the provider attempts to automatically detect the database based on the connection metadata. You can specify the optional `eclipselink.target-database` property to guarantee that the database 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 Application Server and are found to be Java EE compatible.

```
//Supported platforms
eclipselink.JavaDB
eclipselink.Derby
eclipselink.Oracle
eclipselink.SQLServer
eclipselink.DB2
eclipselink.Sybase
eclipselink.MySQL4
eclipselink.PostgreSQL

//Others available
eclipselink.Informix
eclipselink.TimesTen
eclipselink.Attunity
eclipselink.HSQL
```

```
eclipselink.SQLAnyWhere
eclipselink.DBase
eclipselink.DB2Mainframe
eclipselink.Cloudscape
eclipselink.PointBase
```

For more information about the `eclipselink.target-database` property, see *Using EclipseLink JPA Extensions for Session, Target Database and Target Application Server* in (`http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29`).

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

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

For more information about `eclipselink` properties, see "Additional Database Properties" on page 60.

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

To change the persistence provider, see "Changing the Persistence Provider" on page 63.

Additional Database Properties

If you are using the default persistence provider, you can specify in the `persistence.xml` file the database properties listed at Using EclipseLink JPA Extensions (`http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29`).

For schema generation properties, see "Generation Options" on page 62. For query hints, see "Query Hints" on page 63.

Configuring the Cache

If you are using the default persistence provider, you can configure whether caching occurs, the type of caching, the size of the cache, and whether client sessions share the cache. Caching properties for the default persistence provider are described in detail at *Using EclipseLink JPA Extensions for Entity Caching* in Using EclipseLink JPA Extensions (`http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29`).

Setting the Logging Level

One of the default persistence provider's properties that you can set in the `persistence.xml` file is `eclipselink.logging.level`. For example, setting the logging level to `FINE` or higher logs all SQL statements. For details about this property, see *Using EclipseLink JPA Extensions for Logging* inUsing EclipseLink JPA Extensions (`http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29`).

Using Lazy Loading

The default persistence provider treats only `OneToOne`, `ManyToOne`, `OneToMany`, and `ManyToMany` mappings specially when they are annotated as `LAZY`. `OneToMany` and `ManyToMany` mappings are loaded lazily by default in compliance with the Java Persistence Specification. Other mappings are always loaded eagerly. For `OneToOne` and `ManyToOne` mappings, value holder indirection is used. For `OneToMany` and `ManyToMany` mappings, transparent indirection is used.

For basic information about lazy loading, see *What You May Need to Know About EclipseLink JPA Lazy Loading* in Using EclipseLink JPA Extensions (`http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29`). For details about indirection, see *Configuring Indirection (Lazy Loading)* in Configuring a Mapping (`http://wiki.eclipse.org/Configuring_a_Mapping_%28ELUG%29`).

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

- "Annotations" on page 62
- "Generation Options" on page 62

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

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

Annotations

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

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

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

Generation Options

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* in Using EclipseLink JPA Extensions (`http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29`).

Query Hints

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

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

For example:

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

For more information about the query hints available with the default provider, see *How to Use EclipseLink JPA Query Hints* in Using EclipseLink JPA Extensions (http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29).

Changing the Persistence Provider

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

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

First, install the provider. Copy the provider JAR files to the *domain-dir*`/lib` directory, and restart the Application Server. For more information about the *domain-dir*`/lib` directory, see "Using the Application Server Parent Class Loader" on page 29. The new persistence provider is now available to all modules and applications deployed on servers that share the same configuration. However, the *default* provider remains the same.

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

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

```
            </properties>
        </persistence-unit>
    </persistence>
```

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

Database Restrictions and Optimizations

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

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 Application Server uses upper case column names by default, so any mapping references to the column or table names must be in upper case. If column or table names are explicitly specified, the case of all mapping references to the column or table names must be in accordance with the case used in the specified names.

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

Unique Constraints

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

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

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

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

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

Foreign Key Mapping

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

SQL Result Set Mapping

Use the following elements:

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

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

Named Native Queries and JDBC Queries

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

PostgreSQL Case Sensitivity

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

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

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

Sybase Finder Limitation

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

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

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

MySQL Database Restrictions

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

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

  ```
  )  Engine=InnoDB;
  ```

  `InnoDB` provides MySQL with a transaction-safe (ACID compliant) storage engine having commit, rollback, and crash recovery capabilities.

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

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

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

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

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

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

  This example fails with the following error message.

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

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

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

```
mgrId   int         NULL,
PRIMARY KEY (empId),
FOREIGN KEY (mgrId) REFERENCES EMPLOYEE (empId)
ON DELETE SET NULL
) ENGINE=InnoDB;

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

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

◆ ◆ ◆ CHAPTER 7

Developing Web Applications

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

For general information about web applications, see "Part One: The Web Tier" in the Java EE 5 Tutorial (`http://java.sun.com/javaee/5/docs/tutorial/doc/index.html`).

Packaging an EJB JAR File in a Web Application

The Application Server supports the EJB 3.1 specification, which allows EJB JAR files to be packaged in WAR files. EJB classes must reside under `WEB-INF/classes`. For example, the structure of a `hello.war` file might look like this:

```
index.jsp
META-INF/
   MANIFEST.MF
WEB-INF/
   web.xml
   classes/
      com/
         sun/
            v3/
               demo/
                  HelloEJB.class
                  HelloServlet.class
```

For more information about EJB components, see Chapter 8, "Using Enterprise JavaBeans Technology."

Note – For GlassFish v3 Technology Preview 2, EJB modules are not supported unless the optional EJB container module is downloaded from the Update Center. For information about the Update Center, see the *GlassFish v3 Application Server Quick Start Guide*.

For GlassFish v3 Technology Preview 2, only stateless session beans with local interfaces and entity beans that use the Java Persistence API are supported. Stateful, message-driven, and EJB 2.0 and 2.1 entity beans are not supported. Remote interfaces and remote business interfaces for any of the bean types are not supported.

Using Servlets

Application Server supports the Java Servlet Specification version 2.5.

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

To develop servlets, use Sun Microsystems' Java Servlet API. For information about using the Java Servlet API, see the documentation provided by Sun Microsystems at `http://java.sun.com/products/servlet/index.html`.

The Application Server provides the `wscompile` and `wsdeploy` tools to help you implement a web service endpoint as a servlet. For more information about these tools, see the *GlassFish v3 Application Server Reference Manual*.

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

This section contains the following topics:

Invoking a Servlet With a URL

You can call a servlet deployed to the Application Server by using a URL in a browser or embedded as a link in an HTML or JSP file. The format of a servlet invocation URL is as follows:

`http://server:port/context-root/servlet-mapping?name=value`

The following table describes each URL section.

TABLE 7–1 URL Fields for Servlets Within an Application

URL element	Description
server:*port*	The IP address (or host name) and optional port number. To access the default web module for a virtual server, specify only this URL section. You do not need to specify the *context-root* or *servlet-name* unless you also wish to specify name-value parameters.
context-root	For an application, the context root is defined in the `context-root` element of the `application.xml`, `sun-application.xml`, or `sun-web.xml` file. For an individually deployed web module, the context root is specified during deployment. For both applications and individually deployed web modules, the default context root is the name of the WAR file minus the `.war` suffix.
servlet-mapping	The `servlet-mapping` as configured in the `web.xml` file.
?*name*=*value*. . .	Optional request parameters.

In this example, `localhost` is the host name, `MortPages` is the context root, and `calcMortgage` is the servlet mapping:

```
http://localhost:8080/MortPages/calcMortgage?rate=8.0&per=360&bal=180000
```

When invoking a servlet from within a JSP file, you can use a relative path. For example:

```
<jsp:forward page="TestServlet"/>
<jsp:include page="TestServlet"/>
```

Servlet Output

`ServletContext.log` messages are sent to the server log.

By default, the `System.out` and `System.err` output of servlets are sent to the server log, and during startup, server log messages are echoed to the `System.err` output. Also by default, there is no Windows-only console for the `System.err` output.

You can change these defaults using the Admin Console. In the developer profile, select the Application Server component and the Logging tab. In the cluster profile, select the Logger Settings component under the relevant configuration. Then check or uncheck Write to System Log. If this box is checked, `System.out` output is sent to the server log. If it is unchecked, `System.out` output is sent to the system default location only.

For more information, click the Help button in the Admin Console from the Logging page.

Caching Servlet Results

The Application Server can cache the results of invoking a servlet, a JSP, or any URL pattern to make subsequent invocations of the same servlet, JSP, or URL pattern faster. The Application Server caches the request results for a specific amount of time. In this way, if another data call occurs, the Application Server can return the cached data instead of performing the operation again. For example, if your servlet returns a stock quote that updates every 5 minutes, you set the cache to expire after 300 seconds.

Whether to cache results and how to cache them depends on the data involved. For example, it makes no sense to cache the results of a quiz submission, because the input to the servlet is different each time. However, it makes sense to cache a high level report showing demographic data taken from quiz results that is updated once an hour.

To define how an Application Server web application handles response caching, you edit specific fields in the `sun-web.xml` file.

Note – A servlet that uses caching is not portable.

For Javadoc tool pages relevant to caching servlet results, go to `http://glassfish.dev.java.net/nonav/javaee5/api/index.html` and click on the `com.sun.appserv.web.cache` package.

The rest of this section covers the following topics:

Caching Features

The Application Server has the following web application response caching capabilities:

- Caching is configurable based on the servlet name or the URI.
- When caching is based on the URI, this includes user specified parameters in the query string. For example, a response from `/garden/catalog?category=roses` is different from a response from `/garden/catalog?category=lilies`. These responses are stored under different keys in the cache.
- Cache size, entry timeout, and other caching behaviors are configurable.
- Entry timeout is measured from the time an entry is created or refreshed. To override this timeout for an individual cache mapping, specify the `cache-mapping` subelement `timeout`.

- To determine caching criteria programmatically, write a class that implements the `com.sun.appserv.web.cache.CacheHelper` interface. For example, if only a servlet knows when a back end data source was last modified, you can write a helper class to retrieve the last modified timestamp from the data source and decide whether to cache the response based on that timestamp.
- To determine cache key generation programmatically, write a class that implements the `com.sun.appserv.web.cache.CacheKeyGenerator` interface. See "The `CacheKeyGenerator` Interface" on page 75.
- 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.
- 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 *GlassFish v3 Application Server Application Deployment Guide* and "dispatcher" in *GlassFish v3 Application Server Application Deployment Guide*. These are elements in the `sun-web.xml` file.

Default Cache Configuration

If you enable caching but do not provide any special configuration for a servlet or JSP, the default cache configuration is as follows:

- The default cache timeout is 30 seconds.
- Only the HTTP GET method is eligible for caching.
- HTTP requests with cookies or sessions automatically disable caching.
- No special consideration is given to `Pragma:`, `Cache-control:`, or `Vary:` headers.
- The default key consists of the Servlet Path (minus `pathInfo` and the query string).
- A "least recently used" list is maintained to evict cache entries if the maximum cache size is exceeded.
- Key generation concatenates the servlet path with key field values, if any are specified.
- Results produced by resources that are the target of a `RequestDispatcher.include()` or `RequestDispatcher.forward()` call are never cached.

Caching Example

Here is an example cache element in the `sun-web.xml` file:

```
<cache max-capacity="8192" timeout="60">
<cache-helper name="myHelper" class-name="MyCacheHelper"/>
<cache-mapping>
    <servlet-name>myservlet</servlet-name>
    <timeout name="timefield">120</timeout>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
</cache-mapping>
<cache-mapping>
    <url-pattern> /catalog/* </url-pattern>
    <!-- cache the best selling category; cache the responses to
       -- this resource only when the given parameters exist. Cache
       -- only when the catalog parameter has 'lilies' or 'roses'
       -- but no other catalog varieties:
      -- /orchard/catalog?best&category='lilies'
      -- /orchard/catalog?best&category='roses'
      -- but not the result of
       -- /orchard/catalog?best&category='wild'
    -->
    <constraint-field name='best' scope='request.parameter'/>
    <constraint-field name='category' scope='request.parameter'>
        <value> roses </value>
        <value> lilies </value>
    </constraint-field>
     <!-- Specify that a particular field is of given range but the
       -- field doesn't need to be present in all the requests -->
    <constraint-field name='SKUnum' scope='request.parameter'>
        <value match-expr='in-range'> 1000 - 2000 </value>
    </constraint-field>
    <!-- cache when the category matches with any value other than
       -- a specific value -->
    <constraint-field name="category" scope="request.parameter>
        <value match-expr="equals" cache-on-match-failure="true">
       bogus
        </value>
    </constraint-field>
</cache-mapping>
<cache-mapping>
    <servlet-name> InfoServlet </servlet-name>
    <cache-helper-ref>myHelper</cache-helper-ref>
</cache-mapping>
</cache>
```

For more information about the `sun-web.xml` caching settings, see "cache" in *GlassFish v3 Application Server Application Deployment Guide*.

The `CacheKeyGenerator` Interface

The built-in default `CacheHelper` implementation allows web applications to customize the key generation. An application component (in a servlet or JSP) can set up a custom `CacheKeyGenerator` implementation as an attribute in the `ServletContext`.

The name of the context attribute is configurable as the `value` of the `cacheKeyGeneratorAttrName` property in the `default-helper` element of the `sun-web.xml` deployment descriptor. For more information, see "default-helper" in *GlassFish v3 Application Server Application Deployment Guide*.

About the Servlet Engine

Servlets exist in and are managed by the servlet engine in the Application Server. The servlet engine is an internal object that handles all servlet meta functions. These functions include instantiation, initialization, destruction, access from other components, and configuration management. This section covers the following topics:

Instantiating and Removing Servlets

After the servlet engine instantiates the servlet, the servlet engine calls the servlet's `init()` method to perform any necessary initialization. You can override this method to perform an initialization function for the servlet's life, such as initializing a counter.

When a servlet is removed from service, the servlet engine calls the `destroy()` method in the servlet so that the servlet can perform any final tasks and deallocate resources. You can override this method to write log messages or clean up any lingering connections that won't be caught in garbage collection.

Request Handling

When a request is made, the Application Server hands the incoming data to the servlet engine. The servlet engine processes the request's input data, such as form data, cookies, session information, and URL name-value pairs, into an `HttpServletRequest`request object type.

The servlet engine also creates an `HttpServletResponse` response object type. The engine then passes both as parameters to the servlet's `service()` method.

In an HTTP servlet, the default `service()` method routes requests to another method based on the HTTP transfer method: `POST`, `GET`, `DELETE`, `HEAD`, `OPTIONS`, `PUT`, or `TRACE`. For example, HTTP `POST` requests are sent to the `doPost()` method, HTTP `GET` requests are sent to the `doGet()` method, and so on. This enables the servlet to process request data differently, depending on which transfer method is used. Since the routing takes place in the service

method, you generally do not override `service()` in an HTTP servlet. Instead, override `doGet()`, `doPost()`, and so on, depending on the request type you expect.

To perform the tasks to answer a request, override the `service()` method for generic servlets, and the `doGet()` or `doPost()` methods for HTTP servlets. Very often, this means accessing EJB components to perform business transactions, then collating the information in the request object or in a JDBC `ResultSet` object.

Using JavaServer Pages

The Application Server supports the following JSP features:

- JavaServer Pages (JSP) Specification version 2.1
- Precompilation of JSP files, which is especially useful for production servers
- JSP tag libraries and standard portable tags

For information about creating JSP files, see Sun Microsystem's JavaServer Pages web site at `http://java.sun.com/products/jsp/index.html`.

For information about Java Beans, see Sun Microsystem's JavaBeans web page at `http://java.sun.com/beans/index.html`.

Application Server supports tag libraries and standard portable tags. For more information, see the JavaServer Pages Standard Tag Library (JSTL) page at `http://java.sun.com/products/jsp/jstl/index.jsp`.

Web applications don't need to bundle copies of the `jsf-impl.jar` or `appserv-jstl.jar` JSP tag libraries (in *as-install*`/lib`) to use JavaServer™ Faces technology or JSTL, respectively. These tag libraries are automatically available to all web applications.

Creating and Managing Sessions

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

This chapter contains the following sections:

Configuring Sessions

This section covers the following topics:

HTTP Sessions, Cookies, and URL Rewriting

To configure whether and how HTTP sessions use cookies and URL rewriting, edit the `session-properties` and `cookie-properties` elements in the `sun-web.xml` file for an individual web application. For more about the properties you can configure, see "session-properties" in *GlassFish v3 Application Server Application Deployment Guide* and "cookie-properties" in *GlassFish v3 Application Server Application Deployment Guide*.

Coordinating Session Access

Make sure that multiple threads don't simultaneously modify the same session object in conflicting ways.

This is especially likely to occur in web applications that use HTML frames where multiple servlets are executing simultaneously on behalf of the same client. A good solution is to ensure that one of the servlets modifies the session and the others have read-only access.

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.

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

- "The `memory` Persistence Type" on page 77, the default

Note – If the session manager configuration contains an error, the error is written to the server log and the default (`memory`) configuration is used.

For more information, see "session-manager" in *GlassFish v3 Application Server 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 the entire web container, use the `configure-ha-persistence` command. For details, see the *GlassFish v3 Application Server Reference Manual*.

To specify the `memory` persistence type for a specific web application, edit the `sun-web.xml` file as in the following example. The `persistence-type` property is optional, but must be set to `memory` if included. This overrides the web container availability settings for the web application.

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

The only manager property that the `memory` persistence type supports is sessionFilename, which is listed under "manager-properties" in *GlassFish v3 Application Server Application Deployment Guide*.

For more information about the `sun-web.xml` file, see *GlassFish v3 Application Server Application Deployment Guide*.

Using the Grizzly Comet API

This section explains the Comet programming technique and how to create and deploy a Comet-enabled application with the GlassFish v3 Application Server.

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 them to poll the server for new messages.

To accomplish this scenario, Comet applications establish long-lived HTTP connections. This kind of connection reduces the latency experienced by applications that need to periodically poll the server for updates because it relieves them from having to frequently open and close connections. By keeping a connection open, Comet applications can send data to clients at any time over a single connection.

The Grizzly Implementation of Comet

One limitation of the Comet technique is that you must use it with a web server that supports non-blocking connections in order to avoid poor performance. Non-blocking connections are those that do not need to allocate one thread for each request. If the web server were to use blocking connections then it might end up holding many thousands of threads, thereby hindering its scalability.

The GlassFish server includes the Grizzly HTTP Engine, which enables asynchronous request processing (ARP) by avoiding blocking connections. Grizzly's ARP implementation accomplishes this by using the Java NIO API.

With Java NIO, Grizzly enables greater performance and scalability by avoiding the limitations experienced by traditional web servers that must run a thread for each request. Instead, Grizzly's ARP mechanism makes efficient use of a thread pool system and also keeps the state of requests so that it can keep requests alive without holding a single thread for each of them.

Grizzly supports two different implementations of Comet:

- Grizzly Comet
- An implementation of the Bayeux protocol.

Grizzly Comet is based on the ARP and includes a set of APIs that you use from a web component to enable Comet functionality in your web application.

The Bayeaux implementation is a Grizzly implementation of Cometd, which consists of the JSON-based Bayeux message protocol, a set of Dojo libraries, and an event handler. The Grizzly implementation of Cometd consists of a servlet that you reference from your web application.

This tutorial covers the Grizzly Comet implementation only. For more information on using the Bayeux implementation, see Introducing the Cometd framework and its Bayeux protocol support in Grizzly

The Grizzly Comet API

Grizzly's support for Comet includes a small set of APIs that make it easy to add Comet functionality to your web applications. The Grizzly Comet APIs that developers will use most often are the following:

- `CometContext`: A Comet context, which is a shareable space to which applications subscribe in order 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))
```

Client Technologies to Use With Comet

In addition to creating a web component that uses the Comet APIs, you need to enable your client to accept asynchronous updates from the web component. To accomplish this, you can use JavaScript, IFrames, or a framework, such as Dojo.

An IFrame is an HTML element that allows you to include other content in an HTML page. As a result, the client can embed updated content in the IFrame without having to reload the page.

The example explained 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.

Kinds 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, re-using 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 Kind 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.

The Hidden Example

This rest of this tutorial uses the Hidden example to explain how to develop Comet-enabled web applications. You can download the example from grizzly.dev.java.net at,Hidden example download. From there, you can download a pre-built WAR file as well as a JAR file containing the servlet code.

The Hidden example is so called because it uses hidden IFrames. What the example does is it allows multiple clients to increment a counter on the server. When a client increments the counter, the server broadcasts the new count to the clients using the Comet technique.

The Hidden example uses the long-polling technique, but you can easily modify it to use HTTP-streaming by removing two lines. See "Notifying the Comet Handler of an Event" on page 85 and "Creating the HTML Page That Updates and Displays the Content" on page 87 for more information on converting the example to use the HTTP-streaming technique.

The client side of the example uses hidden IFrames with embedded JavaScript tags to connect to the server and to asynchronously post content to and accept updates from the server.

The server side of the example consists of a single servlet that listens for updates from clients, updates the counter, and writes JavaScript code to the client that allows it to update the counter on its page.

See "Deploying and Running a Comet-Enabled Application" on page 89 for instructions on how to deploy and run the example.

When you run the example, the following happens:

1. The `index.html` page opens.
2. The browser loads three frames: the first one accesses the servlet using an HTTP GET; the second one loads the `count.html` page, which displays the current count; and the third one loads the `button.html` page, which is used to send the POST request.
3. After clicking the button on the `button.html` page, the page submits a POST request to the servlet.
4. The `doPost` method calls the `onEvent` method of the Comet handler and redirects the incremented count along with some JavaScript to the `count.html` page on the client.
5. The `updateCount` JavaScript function on the `count.html` page updates the counter on the page.
6. Because this example uses long-polling, the JavaScript code on `count.html` calls `doGet` again to resume the connection after the servlet pushes the update.

Creating a Comet-Enabled Application

This section uses the Hidden example application to demonstrate how to develop a Comet application. The main components of any simple Comet-enabled application are the following:

- A web component, such as a servlet to support the Comet requests and a Comet handler to send updates to the client
- One or more HTML pages that include some client-side technology to open an asynchronous connection to the server and to receive updates from the web component.
- A deployment descriptor to configure the web component.

Developing the Web Component

This section shows you how to create a Comet-enabled web component by giving you instructions for creating the servlet in the Hidden example.

Developing the web component involves performing the following steps:

1. Create a web component to support Comet requests.
2. Register the component with the Comet engine.
3. Define a Comet handler that sends updates to the client.
4. Add the Comet handler to the Comet context.
5. Notify the Comet handler of an event using the Comet context.

▼ Creating a Web Component to Support Comet

1 Create an empty servlet class, like the following:

```
import javax.servlet.*;

public class HiddenCometServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
private String contextPath = null;
    @Override
    public void init(ServletConfig config) throws ServletException {}

    @Override
    protected void doGet(HttpServletRequest req,
    HttpServletResponse res)
    throws ServletException, IOException {}

    @Override
    protected void doPost(HttpServletRequest req,
    HttpServletResponse res)
    throws ServletException, IOException {);
}
```

2 Import the following Comet packages into the servlet class:

```
import com.sun.grizzly.comet.CometContext;
import com.sun.grizzly.comet.CometEngine;
import com.sun.grizzly.comet.CometEvent;
import com.sun.grizzly.comet.CometHandler;
```

3 **Import these additional classes that you need for incrementing a counter and writing output to the clients:**

```
import java.io.IOException;
import java.io.PrintWriter;
import java.util.concurrent.atomic.AtomicInteger;
```

4 **Add a private variable for the counter:**

```
private final AtomicInteger counter = new AtomicInteger();
```

▼ Registering the Servlet with the Comet Engine

1 **In the servlet's `init` method, add the following code to get the component's context path:**

```
ServletContext context = config.getServletContext();
contextPath = context.getContextPath() + "/hidden_comet";
```

2 **Get an instance of the Comet engine by adding this line after the lines from step 1:**

```
 CometEngine engine = CometEngine.getEngine();
```

3 **Register the component with the Comet engine by adding the following lines after those from step 2:**

```
CometContext cometContext = engine.register(contextPath);
cometContext.setExpirationDelay(30 * 1000);
```

▼ Defining a Comet Handler to Send Updates to the Client

1 **Create a private class that implements `CometHandler` and add it to the servlet class:**

```
private class CounterHandler
    implements CometHandler<HttpServletResponse> {
    private HttpServletResponse response;
}
```

2 **Add the following methods to the class:**

```
public void onInitialize(CometEvent event)
    throws IOException {}

    public void onInterrupt(CometEvent event)
        throws IOException {
        removeThisFromContext();
    }

    public void onTerminate(CometEvent event)
        throws IOException {
        removeThisFromContext();
```

```
    }

    public void attach(HttpServletResponse attachment) {
            this.response = attachment;
    }

    private void removeThisFromContext() throws IOException {
        response.getWriter().close();
        CometContext context =
            CometEngine.getEngine().getCometContext(contextPath);
        context.removeCometHandler(this);
    }
```

You need to provide implementations of these methods when implementing `CometHandler`. The `onInterrupt` and `onTerminate` methods execute when certain changes occur in the status of the underlying TCP communication. The `onInterrupt` method executes when communication is resumed. The `onTerminate` method executes when communication is closed. Both methods call `removeThisFromContext`, which removes the `CometHandler` object from the `CometContext` object.

▼ Adding the Comet Handler to the Comet Context

1 **Get an instance of the Comet handler and attach the response to it by adding the following lines to the `doGet` method:**

```
CounterHandler handler = new CounterHandler();
handler.attach(res);
```

2 **Get the Comet context by adding the following lines to `doGet`:**

```
CometEngine engine = CometEngine.getEngine();
CometContext context = engine.getCometContext(contextPath);
```

3 **Add the Comet handler to the Comet context by adding this line to `doGet`:**

```
context.addCometHandler(handler);
```

▼ Notifying the Comet Handler of an Event

1 **Add an `onEvent` method to the `CometHandler` class to define what happens when an event occurs:**

```
public void onEvent(CometEvent event)
    throws IOException {
    if (CometEvent.NOTIFY == event.getType()) {
        int count = counter.get();
        PrintWriter writer = response.getWriter();
        writer.write("<script type='text/javascript'>" +
            "parent.counter.updateCount('" + count + "')" +
```

```
            "</script>\n");
        writer.flush();
        event.getCometContext().resumeCometHandler(this);
    }
}
```

This method first checks if the event type is `NOTIFY`, which means that the web component is notifying the `CometHandler` object that a client has incremented the count. If the event type is `NOTIFY`, the `onEvent` method gets the updated count, and writes out JavaScript to the client. The JavaScript includes a call to the `updateCount` function, which will update the count on the clients' pages.

The last line resumes the Comet request and removes it from the list of active `CometHandler` objects. By this line, you can tell that this application uses the long-polling technique. If you were to delete this line, the application would use the HTTP-Streaming technique.

- **For HTTP-Streaming:**

  Add the same code as for long-polling, except do not include the following line:

  ```
  event.getCometContext().resumeCometHandler(this);
  ```

You don't include this line because you do not want to resume the request. Instead, you want the connection to remain open.

2 **Increment the counter and forward the response by adding the following lines to the `doPost` method:**

```
counter.incrementAndGet();
CometEngine engine = CometEngine.getEngine();
CometContext<?> context =
    engine.getCometContext(contextPath);
context.notify(null);
req.getRequestDispatcher("count.html").forward(req, res);
```

When a user clicks the button, the `doPost` method is called. The `doPost` method increments the counter. It then obtains the current `CometContext` object and calls its `notify` method. By calling `context.notify`, the `doPost` method triggers the `onEvent` method you created in the previous step. After `onEvent` executes, `doPost` forwards the response to the clients.

Creating the Client Pages

Developing the HTML pages for the client involves performing these steps:

1. Create a welcome HTML page, called `index.html`, that contains: one hidden frame for connecting to the servlet through an HTTP GET; one IFrame that embeds the `count.html` page, which contains the updated content; and one IFrame that embeds the `button.html` page, which is used for posting updates using HTTP POST.
2. Create the `count.html` page that contains an HTML element that displays the current count and the JavaScript for updating the HTML element with the new count.

3. Create the `button.html` page that contains a button for the users to submit updates.

Creating a Welcome HTML Page That Contains IFrames for Receiving and Sending Updates

1 Create an HTML page called `index.html`.

2 Add the following content to the page:

```
<html>
    <head>
        <title>Comet Example: Counter with Hidden Frame</title>
    </head>
    <body>
  </body>
</html>
```

3 Add IFrames for connecting to the server and receiving and sending updates to `index.html` in between the `body` tags:

```
<frameset>
    <iframe name="hidden" src="hidden_comet"
        frameborder="0" height="0" width="100%"></iframe>
    <iframe name="counter" src="count.html"
        frameborder="0" height="100%" width="100%"></iframe>
<iframe name="button" src="button.html" frameborder="0" height="30%" widget="100%"></iframe>
</frameset>
```

The first frame, which is hidden, points to the servlet by referencing its context path. The second frame displays the content from `count.html`, which displays the current count. The second frame displays the content from `button.html`, which contains the submit button for incrementing the counter.

Creating the HTML Page That Updates and Displays the Content

1 Create an HTML page called `count.html` and add the following content to it:

```
<html>
    <head>
    </head>
        <body>
            <center>
                <h3>Comet Example: Counter with Hidden Frame</h3>
                <p>
                <b id="count"> </b>
                <p>
            </center>
```

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

This page displays the current count.

2 **Add JavaScript code that updates the count in the page . Add the following lines in between the** `head` **tags of** `count.html`**:**

```
<script type='text/javascript'>
    function updateCount(c) {
        document.getElementById('count').innerHTML = c;
        parent.hidden.location.href = "hidden_comet";
    };
</script>
```

The JavaScript takes the updated count it receives from the servlet and updates the count element in the page. The last line in the `updateCount` function invokes the servlet's `doGet` method again to re-establish the connection.

- **For HTTP-Streaming:**

  Add the same code as for long-polling, except for the following line:

  ```
  parent.hidden.location.href = “hidden_comet”
  ```

  This line invokes the `doGet` method of `CometServlet` again, which would re-establish the connection. In the case of HTTP-Streaming, you want the connection to remain open. Therefore, you don't include this line of code.

▼ Creating the HTML Page That Allows Submitting Updates

- **Create an HTML page called** `button.html` **and add the following content to it:**

```
<html>
    <head>
    </head>
        <body>
            <center>
                <form method="post" action="hidden_comet">
                    <input type="submit" value="Click">
                </form>
            </center>
        </body>
</html>
```

This page displays a form with a button that allows a user to update the count on the server. The servlet will then broadcast the updated count to all clients.

Creating the Deployment Descriptor

This section describes how to create a deployment descriptor to specify how your Comet-enabled web application should be deployed.

▼ Creating the Deployment Descriptor

- **Create a file called `web.xml` and put the following contents in it:**

```
<?xml version="1.0" encoding="UTF-8"?>
    <web-app version="2.5"
        xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation=
            "http://java.sun.com/xml/ns/javaee
            http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd ">

        <servlet>
            <servlet-name>HiddenCometServlet</servlet-name>
            <servlet-class>
                com.sun.grizzly.samples.comet.HiddenCometServlet
            </servlet-class>
            <load-on-startup>0</load-on-startup>
        </servlet>
        <servlet-mapping>
            <servlet-name>HiddenCometServlet</servlet-name>
            <url-pattern>/hidden_comet</url-pattern>
        </servlet-mapping>
    </web-app>
```

This deployment descriptor contains a servlet declaration and mapping for `HiddenCometServlet`. The `load-on-startup` attribute must be set to 0 so that the Comet-enabled servlet will not load until the client makes a request to it.

Deploying and Running a Comet-Enabled Application

Before running a Comet-enabled application in the Application Server, you need to enable Comet in the server. Then you can deploy the application just as you would any other web application.

When running the application, you need to connect to it from at least two different browsers to experience the effect of the servlet updating all clients in response to one client posting an update to the server.

Enabling Comet in the Application Server

Before running a Comet-enabled application, you need to enable Comet in your application server by adding a special property to the `http-listener` element of the `domain.xml` file.

The following steps tell you how to add this property.

1. **Open** *as-install*`/domains/domain1/config/domain.xml` **in a text editor.**
2. **Add the following property in between the** `http-listener` **start and end tags:**

   ```
   <property name="cometSupport" value="true"/>
   ```

3. **Save** `domain.xml`**.**

Deploying the Example

These instructions tell you how to deploy the Hidden example.

1. **Download grizzly-comet-hidden-1.7.3.1.war.**
2. **Download and install the GlassFish v3 Application Server.**
3. **Run the following command to deploy the example:**

   as-install`/bin/asadmin deploy grizzly-comet-hidden-1.7.3.1.war`

Running the Example

These instructions tell you how to run the Hidden example.

1. **Open two web browsers, preferably two different brands of web browser.**
2. **Enter the following URL in both browsers:**

   ```
   http://localhost:8080/grizzly-comet-hidden/index.html
   ```

3. **When the first page loads in both browsers, click the button in one of the browsers and watch the count change in the other browser window.**

Advanced Web Application Features

This section includes summaries of the following topics:

Internationalization Issues

This section covers internationalization as it applies to the following:

The Server's Default Locale

To set the default locale of the entire Application Server, which determines the locale of the Admin Console, the logs, and so on, use the Admin Console. In the developer profile, select the Application Server component, the Advanced tab, and the Domain Attributes tab. In the cluster profile, select the domain component. Then type a value in the Locale field. For details, click the Help button in the Admin Console.

Servlet Character Encoding

This section explains how the Application Server determines the character encoding for the servlet request and the servlet response. For encodings you can use, see `http://java.sun.com/j2se/1.5.0/docs/guide/intl/encoding.doc.html`.

Servlet Request

When processing a servlet request, the server uses the following order of precedence, first to last, to determine the request character encoding:

- The `getCharacterEncoding()` method
- A hidden field in the form, specified by the `form-hint-field` attribute of the `parameter-encoding` element in the `sun-web.xml` file
- The `default-charset` attribute of the `parameter-encoding` element in the `sun-web.xml` file
- The default, which is `ISO-8859-1`

For details about the `parameter-encoding` element, see "parameter-encoding" in *GlassFish v3 Application Server Application Deployment Guide*.

Servlet Response

When processing a servlet response, the server uses the following order of precedence, first to last, to determine the response character encoding:

- The `setCharacterEncoding()` or `setContentType()` method
- The `setLocale()` method
- The default, which is `ISO-8859-1`

Virtual Servers

A virtual server, also called a virtual host, is a virtual web server that serves content targeted for a specific URL. Multiple virtual servers can serve content using the same or different host names, port numbers, or IP addresses. The HTTP service directs incoming web requests to different virtual servers based on the URL.

When you first install the Application Server, a default virtual server is created. You can also assign a default virtual server to each new HTTP listener you create.

Web applications and Java EE applications containing web components (web modules) can be assigned to virtual servers during deployment. A web module can be assigned to more than one virtual server, and a virtual server can have more than one web module assigned to it.

▼ To Assign a Default Virtual Server

1. **In the Admin Console, open the HTTP Service component under the relevant configuration.**
2. **Open the HTTP Listeners component under the HTTP Service component.**
3. **Select or create a new HTTP listener.**
4. **Select from the Default Virtual Server drop-down list.**
   For more information, see "Default Web Modules" on page 93.

See Also For details, click the Help button in the Admin Console from the HTTP Listeners page.

▼ To Assign Virtual Servers

1. **Deploy the application or web module and assign the desired virtual servers to it.**
   For more information, see *GlassFish v3 Application Server Application Deployment Guide*.
2. **In the Admin Console, open the HTTP Service component under the relevant configuration.**
3. **Open the Virtual Servers component under the HTTP Service component.**
4. **Select the virtual server to which you want to assign a default web module.**
5. **Select the application or web module from the Default Web Module drop-down list.**
   For more information, see "Default Web Modules" on page 93.

See Also For details, click the Help button in the Admin Console from the Virtual Servers page.

Default Web Modules

A default web module can be assigned to the default virtual server and to each new virtual server. For details, see "Virtual Servers" on page 92. To access the default web module for a virtual server, point the browser to the URL for the virtual server, but do not supply a context root. For example:

```
http://myvserver:3184/
```

A virtual server with no default web module assigned serves HTML or JavaServer Pages (JSP) content from its document root, which is usually *domain-dir*`/docroot`. To access this HTML or JSP content, point your browser to the URL for the virtual server, do not supply a context root, but specify the target file.

For example:

```
http://myvserver:3184/hellothere.jsp
```

Class Loader Delegation

The Servlet specification recommends that the Web class loader look in the local class loader before delegating to its parent. To make the Web class loader follow the delegation model in the Servlet specification, set `delegate="false"` in the `class-loader` element of the `sun-web.xml` file. It's safe to do this only for a web module that does not interact with any other modules.

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

Note – For Technology Preview 2, the `delegate` value is ignored and assumed to be set to `true`.

For general information about class loaders, see Chapter 2, "Class Loaders."

Using the `default-web.xml` File

You can use the `default-web.xml` file to define features such as filters and security constraints that apply to all web applications.

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 Admin Console to edit the `default-web.xml` file. For details, click the Help button in the Admin 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 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 Application Server cluster, configure the load balancer to retry failed idempotent HTTP requests on all the Application Server instances in a cluster. This option can be used for read-only requests, for example, to retry a search request.

This section describes the following topics:

Specifying an Idempotent URL

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

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

For details, see "idempotent-url-pattern" in *GlassFish v3 Application Server 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 Application Server instance is unreachable, the load balancer returns an error message. Normally, the request is not retried on another Application Server instance. However, if the URL pattern matches that specified in the `sun-web.xml` file, the request is implicitly retried on another Application Server 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 Application Server instance.

For example, if the request URL sent to the Application Server 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 Application Server, the `Enumeration` from `request.getHeaders()` contains multiple elements (one element per request header) instead of a single, aggregated value.

The header names used in `HttpServletResponse.add`*XXX*`Header()` and `HttpServletResponse.set`*XXX*`Header()` are returned as they were created.

Configuring Valves and Catalina Listeners

You can configure custom valves and Catalina listeners for web modules or virtual servers by defining properties. In the `domain.xml` file, valve and listener properties look like this:

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

You can define these properties in one of the following ways, then restart the server:

- You can define virtual server properties using the Admin 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, click the Help button in the Admin Console.

Alternate Document Roots

An alternate document root (docroot) allows a web application to serve requests for certain resources from outside its own docroot, based on whether those requests match one (or more) of the URI patterns of the web application's alternate docroots.

To specify an alternate docroot for a web application or a virtual server, use the `alternatedocroot_`*n* property, where *n* is a positive integer that allows specification of more than one. This property can be a subelement of a `sun-web-app` element in the `sun-web.xml` file or a `virtual-server` element in the `domain.xml` file. For more information about these elements, see "sun-web-app" in *GlassFish v3 Application Server 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 docroots. The URI pattern of the first alternate docroot uses an exact match, whereas the URI patterns of the second and third alternate docroots use extension and longest path prefix matches, respectively.

```
<property name="alternatedocroot_1" value="from=/my.jpg dir=/srv/images/jpg"/>
<property name="alternatedocroot_2" value="from=*.jpg dir=/srv/images/jpg"/>
<property name="alternatedocroot_3" value="from=/jpg/* dir=/src/images"/>
```

The `value` of each alternate docroot has two components: The first component, `from`, specifies the alternate docroot's URI pattern, and the second component, `dir`, specifies the alternate docroot's physical location (directory).

Suppose the above examples belong to a web application deployed at `http://company22.com/myapp`. The first alternate docroot maps any requests with this URL:

```
http://company22.com/myapp/my.jpg
```

To this resource:

```
/svr/images/jpg/my.jpg
```

The second alternate docroot maps any requests with a `*.jpg` suffix, such as:

```
http://company22.com/myapp/*.jpg
```

To this physical location:

```
/svr/images/jpg
```

The third alternate docroot maps any requests whose URI starts with `/myapp/jpg/`, such as:

```
http://company22.com/myapp/jpg/*
```

To the same directory as the second alternate docroot.

For example, the second alternate docroot maps this request:

```
http://company22.com/myapp/abc/def/my.jpg
```

To:

```
/srv/images/jpg/abc/def/my.jpg
```

The third alternate docroot maps:

```
http://company22.com/myapp/jpg/abc/resource
```

To:

```
/srv/images/jpg/abc/resource
```

If a request does not match any of the target web application's alternate docroots, or if the target web application does not specify any alternate docroots, the request is served from the web application's standard docroot, as usual.

Redirecting URLs

You can specify that a request for an old URL is treated as a request for a new URL. This is called *redirecting* a URL.

To specify a redirected URL for a virtual server, use the `redirect_n` property, where *n* is a positive integer that allows specification of more than one. This property is a subelement of a `virtual-server` element in the `domain.xml` file.. Each of these `redirect_n` properties is inherited by all web applications deployed on the virtual server.

The value of each `redirect_n` property has two components, which may be specified in any order:

The first component, `from`, specifies the prefix of the requested URI to match.

The second component, `url-prefix`, specifies the new URL prefix to return to the client. The from prefix is simply replaced by this URL prefix.

For example:

```
<property name="redirect_1" value="from=/dummy url-prefix=http://etude"/>
```

Using a context.xml File

Use the `contextXmlDefault` property to specify the location, relative to *domain-dir*, of the `context.xml` file for a virtual server. For more information about virtual servers, see "Virtual Servers" on page 92. For more information about the `context.xml` file, see The Context Container (`http://tomcat.apache.org/tomcat-5.5-doc/config/context.html`).

Enabling WebDav

To enable WebDav in the Application Server, you edit the `web.xml` and `sun-web.xml` files as follows.

First, enable the WebDav servlet in your `web.xml` file:

```
<servlet>
   <servlet-name>webdav</servlet-name>
   <servlet-class>org.apache.catalina.servlets.WebdavServlet</servlet-class>
```

```
   <init-param>
      <param-name>debug</param-name>
      <param-value>0</param-value>
   </init-param>
   <init-param>
      <param-name>listings</param-name>
      <param-value>true</param-value>
   </init-param>
   <init-param>
      <param-name>readonly</param-name>
      <param-value>false</param-value>
   </init-param>
</servlet>
```

Then define the servlet mapping associated with your WebDav servlet in your `web.xml` file:

```
<servlet-mapping>
   <servlet-name>webdav</servlet-name>
   <url-pattern>/webdav/*</url-pattern>
</servlet-mapping>
```

To protect the WebDav servlet so other users can't modify it, add a security constraint in your `web.xml` file:

```
<security-constraint>
   <web-resource-collection>
      <web-resource-name>Login Resources</web-resource-name>
      <url-pattern>/webdav/*</url-pattern>
   </web-resource-collection>
   <auth-constraint>
      <role-name>Admin</role-name>
   </auth-constraint>
   <user-data-constraint>
      <transport-guarantee>NONE</transport-guarantee>
   </user-data-constraint>
   <login-config>
      <auth-method>BASIC</auth-method>
      <realm-name>default</realm-name>
   </login-config>
   <security-role>
      <role-name>Admin</role-name>
   </security-role>
</security-constraint>
```

Then define a security role mapping in your `sun-web.xml` file:

```
<security-role-mapping>
   <role-name>Admin</role-name>
```

```
   <group-name>Admin</group-name>
</security-role-mapping>
```

If you are using the `file` realm, create a user and password. For example:

```
asadmin create-file-user --user admin --host localhost --port 4848 --terse=true
--groups Admin --authrealmname default admin
```

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 mod_jk

To set up `mod_jk`, follow these steps:

1. Obtain the following software:
   - Apache 2.0.*x*
   - Apache Tomcat Connectors (`http://www.apache.org/dist/tomcat/tomcat-connectors/jk/binaries/`)
   - Apache Tomcat 5.5.16, needed for just one JAR file (`http://archive.apache.org/dist/tomcat/tomcat-5/v5.5.16/bin/apache-tomcat-5.5.16.tar.gz`)
   - Apache Commons Logging 1.0.4 (`http://archive.apache.org/dist/jakarta/commons/logging/binaries/commons-logging-1.0.4.tar.gz`)
   - Apache Commons Modeler 1.1 (`http://archive.apache.org/dist/jakarta/commons/modeler/binaries/modeler-1.1.tar.gz`)
2. Install `mod_jk` as described at `http://tomcat.apache.org/connectors-doc/webserver_howto/apache.html`.
3. Copy the following Tomcat and Jakarta Commons files to *as-install*`/lib`:
   - `tomcat-ajp.jar`
   - `commons-logging.jar`
   - `commons-modeler.jar`
4. Create and configure the following files:
   - `/etc/httpd/conf/httpd.conf`

- `/etc/httpd/conf/worker.properties` or *domain-dir*`/config/glassfish-jk.properties` (to use non-default values of attributes described at `http://tomcat.apache.org/tomcat-5.5-doc/config/ajp.html`)

Examples of these files are shown after these steps. If you use both `worker.properties` and `glassfish-jk.properties` files, the file referenced by `httpd.conf`, or referenced by `httpd.conf` first, takes precedence.

5. Start `httpd`.
6. Enable `mod_jk` using the following command:

```
asadmin create-jvm-options -Dcom.sun.enterprise.web.connector.enableJK=8009
```

7. If you are using the `glassfish-jk.properties` file and not referencing it in `httpd.conf`, point to it using the following command:

```
asadmin create-jvm-options
-Dcom.sun.enterprise.web.connector.enableJK.propertyFile=domain-dir/config/glassfish-jk.properties
```

8. Restart the Application Server.

Here is an example `httpd.conf` file:

```
LoadModule jk_module /usr/lib/httpd/modules/mod_jk.so
JkWorkersFile /etc/httpd/conf/worker.properties
# Where to put jk logs
JkLogFile /var/log/httpd/mod_jk.log
# Set the jk log level [debug/error/info]
JkLogLevel debug
# Select the log format
JkLogStampFormat "[%a %b %d %H:%M:%S %Y] "
# JkOptions indicate to send SSL KEY SIZE,
JkOptions +ForwardKeySize +ForwardURICompat -ForwardDirectories
# JkRequestLogFormat set the request format
JkRequestLogFormat "%w %V %T"
# Send all jsp requests to GlassFish
JkMount /*.jsp worker1
# Send all glassfish-test requests to GlassFish
JkMount /glassfish-test/* worker1
```

Here is an example `worker.properties` or `glassfish-jk.properties` file:

```
# Define 1 real worker using ajp13
worker.list=worker1
# Set properties for worker1 (ajp13)
worker.worker1.type=ajp13
worker.worker1.host=localhost.localdomain
worker.worker1.port=8009
worker.worker1.lbfactor=50
```

```
worker.worker1.cachesize=10
worker.worker1.cache_timeout=600
worker.worker1.socket_keepalive=1
worker.worker1.socket_timeout=300
```

◆ ◆ ◆ CHAPTER 8

Using Enterprise JavaBeans Technology

This chapter describes how Enterprise JavaBeans™ (EJB™) technology is supported in the GlassFish Application Server. This chapter addresses the following topics:

For general information about enterprise beans, see "Part Three: Enterprise Beans" in the Java EE 5 Tutorial (`http://java.sun.com/javaee/5/docs/tutorial/doc/index.html`).

Note – For GlassFish v3 Technology Preview 2, EJB modules are not supported unless the optional EJB container module is downloaded from the Update Center. For information about the Update Center, see the *GlassFish v3 Application Server Quick Start Guide*.

For GlassFish v3 Technology Preview 2, only stateless session beans with local interfaces and entity beans that use the Java Persistence API are supported. Stateful, message-driven, and EJB 2.0 and 2.1 entity beans are not supported. Remote interfaces and remote business interfaces for any of the bean types are not supported.

Summary of EJB 3.1 Changes

The Application Server supports and is compliant with the Sun Microsystems Enterprise JavaBeans (EJB) architecture as defined by the Enterprise JavaBeans Specification, v3.1, also known as JSR 318 (`http://jcp.org/en/jsr/detail?id=318`).

The main changes in the Enterprise JavaBeans Specification, v3.1 that impact enterprise beans in the Application Server environment are as follows:

- An EJB component need not implement any interface as long as it contains one of the component defining annotations or the XML equivalent. Essentially, the local business interface is optional. For example, the following is a simple no-interface bean:

```
@Stateless
public class HelloBean {
   public String sayHello(String msg) {
      return "Hello " + msg;
   }
}
```

  Even though the bean doesn't implement any interface, the client can still inject (or look up) a reference to the session bean. The client still has to perform a JNDI lookup or inject a reference of the bean. More specifically, it *cannot* use the new operator to construct the bean.

```
@EJB HelloBean h;
...
h.sayHello("bob");
```

- EJB classes can be packaged inside WAR files. These classes must reside under `WEB-INF/classes`. For example, the structure of a `hello.war` file might look like this:

```
index.jsp
META-INF/
   MANIFEST.MF
WEB-INF/
   web.xml
   classes/
      com/
         sun/
            v3/
               demo/
                  HelloEJB.class
                  HelloServlet.class
```

  For more information about web applications, see Chapter 7, "Developing Web Applications."

Value Added Features

The Application Server provides a number of value additions that relate to EJB development. These capabilities are discussed in the following sections. References to more in-depth material are included.

Bean-Level Container-Managed Transaction Timeouts

The default transaction timeout for the domain is specified using the Transaction Timeout setting of the Transaction Service. A transaction started by the container must commit (or rollback) within this time, regardless of whether the transaction is suspended (and resumed), or the transaction is marked for rollback.

To override this timeout for an individual bean, use the optional `cmt-timeout-in-seconds` element in `sun-ejb-jar.xml`. The default value, `0`, specifies that the default Transaction Service timeout is used. The value of `cmt-timeout-in-seconds` is used for all methods in the bean that start a new container-managed transaction. This value is *not* used if the bean joins a client transaction.

EJB Timer Service

The EJB Timer Service uses a database to store persistent information about EJB timers.

The EJB Timer Service in Application Server is preconfigured to use an embedded version of the Java DB database. In the Admin Console, open the Resources component and select JDBC Resources. For details, click the Help button in the Admin Console. Change the connection pool name for the JDBC resource named `jdbc/__TimerPool` to point to the same connection pool as the one you are using for the rest of your data. Then start the database.

To enable the timer service, deploy the following application:

as-install`/modules/ejb/ejb-timer-service-app-10.0-tp-2-SNAPSHOT.war`

You can verify that it was deployed successfully by accessing the following URL:

```
http://localhost:8080/ejb-timer-service-app/timer
```

The EJB Timer Service configuration can store persistent timer information in any database supported by the Application Server for persistence. For a list of the JDBC drivers currently supported by the Application Server, see the *GlassFish v3 Application Server Release Notes*. For configurations of supported and other drivers, see "Configuration Specifics for JDBC Drivers" in *GlassFish v3 Application Server Administration Guide*.

To change the database used by the EJB Timer Service, set the EJB Timer Service's Timer DataSource setting to a valid JDBC resource. 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 `7000`.
- Maximum Redeliveries - Specifies the maximum number of times the EJB timer service attempts to redeliver a timer expiration due for exception or rollback. The default is `1`.
- Redelivery Interval - Specifies how long in milliseconds the EJB timer service waits after a failed `ejbTimeout` delivery before attempting a redelivery. The default is `5000`.
- Timer DataSource - Specifies the database used by the EJB Timer Service. The default is `jdbc/__TimerPool`.

Using Session Beans

This section provides guidelines for creating session beans in the Application Server environment. This section addresses the following topics:

- "About the Session Bean Containers" on page 106
- "Session Bean Restrictions and Optimizations" on page 107

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.

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 Application Server specific deployment descriptor file, `sun-ejb-jar.xml`, contains the properties that define the pool:

- `steady-pool-size`
- `resize-quantity`
- `max-pool-size`
- `pool-idle-timeout-in-seconds`

For more information about `sun-ejb-jar.xml`, see "The sun-ejb-jar.xml File" in *GlassFish v3 Application Server Application Deployment Guide*.

The Application Server provides the `wscompile` and `wsdeploy` tools to help you implement a web service endpoint as a stateless session bean. For more information about these tools, see the *GlassFish v3 Application Server Reference Manual*.

Session Bean Restrictions and Optimizations

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

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.

Handling Transactions With Enterprise Beans

This section describes the transaction support built into the Enterprise JavaBeans programming model for the Application Server.

This section provides overview information on the following topics:

Flat Transactions

The Enterprise JavaBeans Specification, v3.0 requires support for flat (as opposed to nested) transactions. In a flat transaction, each transaction is decoupled from and independent of other transactions in the system. Another transaction cannot start in the same thread until the current transaction ends.

Flat transactions are the most prevalent model and are supported by most commercial database systems. Although nested transactions offer a finer granularity of control over transactions, they are supported by far fewer commercial database systems.

Local Transactions

Understanding the distinction between global and local transactions is crucial in understanding the Application Server support for transactions. See "Transaction Scope" on page 119. For GlassFish v3 Technology Preview 2, only local transactions are supported.

Transactions are demarcated using the `javax.transaction.UserTransaction` interface, which the client must use. For more information, see "The Transaction Manager, the Transaction Synchronization Registry, and `UserTransaction`" on page 120.

Administration and Monitoring

The Transaction Timeout setting can be overridden by a bean. See "Bean-Level Container-Managed Transaction Timeouts" on page 105.

In addition, the administrator can monitor transactions using statistics from the transaction manager that provide information on such activities as the number of transactions completed, rolled back, or recovered since server startup, and transactions presently being processed.

For information on administering and monitoring transactions, select the Transaction Service component under the relevant configuration in the Admin Console and click the Help button.

PART III

Using Services and APIs

◆ ◆ ◆ CHAPTER 9

Using the JDBC API for Database Access

This chapter describes how to use the Java™ Database Connectivity (JDBC™) API for database access with the GlassFish Application Server. This chapter also provides high level JDBC implementation instructions for servlets using the Application Server. If the JDK version 1.6 is used, the Application Server supports the JDBC 4.0 API.

The JDBC specifications are available at
`http://java.sun.com/products/jdbc/download.html`.

A useful JDBC tutorial is located at
`http://java.sun.com/docs/books/tutorial/jdbc/index.html`.

Note – The Application Server does not support connection pooling or transactions for an application's database access if it does not use standard Java EE `DataSource` objects.

This chapter discusses the following topics:

General Steps for Creating a JDBC Resource

To prepare a JDBC resource for use in Java EE applications deployed to the Application Server, perform the following tasks:

For information about how to configure some specific JDBC drivers, see "Configuration Specifics for JDBC Drivers" in *GlassFish v3 Application Server Administration Guide*.

Integrating the JDBC Driver

To use JDBC features, you must choose a JDBC driver to work with the Application Server, then you must set up the driver. This section covers these topics:

- "Supported Database Drivers" on page 112
- "Making the JDBC Driver JAR Files Accessible" on page 112

Supported Database Drivers

Supported JDBC drivers are those that have been fully tested by Sun. For a list of the JDBC drivers currently supported by the Application Server, see the *GlassFish v3 Application Server Release Notes*. For configurations of supported and other drivers, see "Configuration Specifics for JDBC Drivers" in *GlassFish v3 Application Server Administration Guide*.

Note – Because the drivers and databases supported by the Application Server are constantly being updated, and because database vendors continue to upgrade their products, always check with Sun technical support for the latest database support information.

Making the JDBC Driver JAR Files Accessible

To integrate the JDBC driver into a Application Server domain, copy the JAR files into the *domain-dir*`/lib` directory, then restart the server. This makes classes accessible to all applications or modules deployed on servers that share the same configuration. For more information about Application Server class loaders, see Chapter 2, "Class Loaders."

Creating a Connection Pool

When you create a connection pool that uses JDBC technology (a *JDBC connection pool*) in the Application Server, you can define many of the characteristics of your database connections.

You can create a JDBC connection pool in one of these ways:

- In the Admin Console, open the Resources component and select Connection Pools. For details, click the Help button in the Admin Console.
- Use the `asadmin create-jdbc-connection-pool` command. For details, see the *GlassFish v3 Application Server Reference Manual*.

For a complete description of JDBC connection pool features, see the *GlassFish v3 Application Server Administration Guide*

Testing a JDBC Connection Pool

You can test a JDBC connection pool for usability in one of these ways:

- In the Admin Console, open the Resources 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 Admin Console.
- Use the `asadmin ping-connection-pool` command. For details, see the *GlassFish v3 Application Server Reference Manual*.

Both these commands fail and display an error message unless they successfully connect to the connection pool.

Creating a JDBC Resource

A JDBC resource, also called a data source, lets you make connections to a database using `getConnection()`. Create a JDBC resource in one of these ways:

- In the Admin Console, open the Resources component and select JDBC Resources. For details, click the Help button in the Admin Console.
- Use the `asadmin create-jdbc-resource` command. For details, see the *GlassFish v3 Application Server Reference Manual*.

Creating Web Applications That Use the JDBC API

A web application that uses the JDBC API is an application that looks up and connects to one or more databases. This section covers these topics:

Sharing Connections

When multiple connections acquired by an application use the same JDBC resource, the connection pool provides connection sharing within the same transaction scope. For example, suppose Bean A starts a transaction and obtains a connection, then calls a method in Bean B. If Bean B acquires a connection to the same JDBC resource with the same sign-on information, and if Bean A completes the transaction, the connection can be shared.

Connections obtained through a resource are shared only if the resource reference declared by the Java EE component allows it to be shareable. This is specified in a component's deployment descriptor by setting the `res-sharing-scope` element to `Shareable` for the particular resource reference. To turn off connection sharing, set `res-sharing-scope` to `Unshareable`.

For general information about connections and JDBC URLs, see Chapter 4, "Administering Database Connectivity ," in *GlassFish v3 Application Server Administration Guide*.

Obtaining a Physical Connection From a Wrapped Connection

The `DataSource` implementation in the Application Server provides a `getConnection` method that retrieves the JDBC driver's `SQLConnection` from the Application Server's `Connection` wrapper. The method signature is as follows:

```
public java.sql.Connection getConnection(java.sql.Connection con)
throws java.sql.SQLException
```

For example:

```
InitialContext ctx = new InitialContext();
com.sun.appserv.jdbc.DataSource ds = (com.sun.appserv.jdbc.DataSource)
   ctx.lookup("jdbc/MyBase");
Connection con = ds.getConnection();
Connection drivercon = ds.getConnection(con); //get physical connection from wrapper
// Do db operations.
// Do not close driver connection.
con.close(); // return wrapped connection to pool.
```

Using the `Connection.unwrap()` Method

If the JDK version 1.6 is used, the Application Server supports JDBC 4.0 if the JDBC driver is JDBC 4.0 compliant. Using the `Connection.unwrap()` method on a vendor-provided interface returns an object or a wrapper object implementing the vendor-provided interface, which the application can make use of to do vendor-specific database operations. Use the `Connection.isWrapperFor()` method on a vendor-provided interface to check whether the connection can provide an implementation of the vendor-provided interface. Check the JDBC driver vendor's documentation for information on these interfaces.

Marking Bad Connections

The `DataSource` implementation in the Application Server provides a `markConnectionAsBad` method. A marked bad connection is removed from its connection pool when it is closed. The method signature is as follows:

```
public void markConnectionAsBad(java.sql.Connection con)
```

For example:

```
com.sun.appserv.jdbc.DataSource ds=
   (com.sun.appserv.jdbc.DataSource)context.lookup("dataSource");
Connection con = ds.getConnection();
Statement stmt = null;
try{
   stmt = con.createStatement();
   stmt.executeUpdate("Update");
}
catch (BadConnectionException e){
   ds.markConnectionAsBad(con) //marking it as bad for removal
}
finally{
   stmt.close();
   con.close(); //Connection will be destroyed during close.
}
```

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 JDBC Connection Pools page in the Admin Console. The default is unchecked. For more information, click the Help button in the Admin Console.
- Specify the `--nontransactionalconnections` option in the `asadmin create-jdbc-connection-pool` command. For more information, see the *GlassFish v3 Application Server Reference Manual*.
- Use the `DataSource` implementation in the Application Server, which provides a `getNonTxConnection` method. This method retrieves a JDBC connection that is not in the scope of any transaction. There are two variants.

  ```
  public java.sql.Connection getNonTxConnection() throws java.sql.SQLException

  public java.sql.Connection getNonTxConnection(String user, String password)
     throws java.sql.SQLException
  ```

- Create a resource with the JNDI name ending in `__nontx`. This forces all connections looked up using this resource to be non transactional.

Typically, a connection is enlisted in the context of the transaction in which a `getConnection` call is invoked. However, a non-transactional connection is not enlisted in a transaction context even if a transaction is in progress.

The main advantage of using non-transactional connections is that the overhead incurred in enlisting and delisting connections in transaction contexts is avoided. However, use such connections carefully. For example, if a non-transactional connection is used to query the database while a transaction is in progress that modifies the database, the query retrieves the unmodified data in the database. This is because the in-progress transaction hasn't committed. For another example, if a non-transactional connection modifies the database and a transaction that is running simultaneously rolls back, the changes made by the non-transactional connection are not rolled back.

Here is a typical use case for a non-transactional connection: a component that is updating a database in a transaction context spanning over several iterations of a loop can refresh cached data by using a non-transactional connection to read data before the transaction commits.

Using JDBC Transaction Isolation Levels

For general information about transactions, see Chapter 10, "Using the Transaction Service."

Not all database vendors support all transaction isolation levels available in the JDBC API. The Application Server permits specifying any isolation level your database supports. The following table defines transaction isolation levels.

TABLE 9–1 Transaction Isolation Levels

Transaction Isolation Level	Description
`TRANSACTION_READ_UNCOMMITTED`	Dirty reads, non-repeatable reads, and phantom reads can occur.
`TRANSACTION_READ_COMMITTED`	Dirty reads are prevented; non-repeatable reads and phantom reads can occur.
`TRANSACTION_REPEATABLE_READ`	Dirty reads and non-repeatable reads are prevented; phantom reads can occur.
`TRANSACTION_SERIALIZABLE`	Dirty reads, non-repeatable reads and phantom reads are prevented.

You can specify the transaction isolation level in the following ways:

- Select the value from the Transaction Isolation drop-down list on the JDBC Connection Pools page in the Admin Console. For more information, click the Help button in the Admin Console.
- Specify the `--isolationlevel` option in the `asadmin create-jdbc-connection-pool` command. For more information, see the *GlassFish v3 Application Server Reference Manual*.

Note that you cannot call `setTransactionIsolation()` during a transaction.

You can set the default transaction isolation level for a JDBC connection pool. For details, see "Creating a Connection Pool" on page 112.

To verify that a level is supported by your database management system, test your database programmatically using the `supportsTransactionIsolationLevel()` method in `java.sql.DatabaseMetaData`, as shown in the following example:

```
InitialContext ctx = new InitialContext();
DataSource ds = (DataSource)
ctx.lookup("jdbc/MyBase");
Connection con = ds.getConnection();
DatabaseMetaData dbmd = con.getMetaData();
if (dbmd.supportsTransactionIsolationLevel(TRANSACTION_SERIALIZABLE)
{ Connection.setTransactionIsolation(TRANSACTION_SERIALIZABLE); }
```

For more information about these isolation levels and what they mean, see the JDBC API specification.

Note – Applications that change the isolation level on a pooled connection programmatically risk polluting the pool, which can lead to errors.

Allowing Non-Component Callers

You can allow non-Java-EE components, such as servlet filters and third party persistence managers, to use this JDBC connection pool. The returned connection is automatically enlisted with the transaction context obtained from the transaction manager. Standard Java EE components can also use such pools. Connections obtained by non-component callers are not automatically closed at the end of a transaction by the container. They must be explicitly closed by the caller.

You can enable non-component callers in the following ways:

- Check the Allow Non Component Callers box on the JDBC Connection Pools page in the Admin Console. The default is `false`. For more information, click the Help button in the Admin Console.
- Specify the `--allownoncomponentcallers` option in the `asadmin create-jdbc-connection-pool` command. For more information, see the *GlassFish v3 Application Server Reference Manual*.
- Create a JDBC resource with a `__pm` suffix.

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 GlassFish JDBC drivers for Sybase databases create a stored procedure for each parameterized `PreparedStatement`. On the Application Server, exceptions are thrown when primary key identity generation is attempted. To disable the creation of these stored procedures, set the property `PrepareMethod=direct` for the JDBC connection pool.

◆ ◆ ◆ CHAPTER 10

Using the Transaction Service

The Java EE platform provides several abstractions that simplify development of dependable transaction processing for applications. This chapter discusses Java EE transactions and transaction support in the GlassFish Application Server.

This chapter contains the following sections:

For more information about the Java™ Transaction API (JTA) and Java Transaction Service (JTS), see the following sites: `http://java.sun.com/products/jta/` and `http://java.sun.com/products/jts/`.

You might also want to read "Chapter 35: Transactions" in the Java EE 5 Tutorial (`http://java.sun.com/javaee/5/docs/tutorial/doc/index.html`).

For information about JDBC transaction isolation levels, see "Using JDBC Transaction Isolation Levels" on page 116.

Transaction Scope

A *local* transaction involves only one non-XA resource and requires that all participating application components execute within one process. Local transaction optimization is specific to the resource manager and is transparent to the Java EE application.

In the Application Server, a JDBC resource is non-XA if it meets any of the following criteria:

- In the JDBC connection pool configuration, the DataSource class does not implement the `javax.sql.XADataSource` interface.
- The Global Transaction Support box is not checked, or the Resource Type setting does not exist or is not set to `javax.sql.XADataSource`.

A transaction remains local if the following conditions remain true:

- One and only one non-XA resource is used. If any additional non-XA resource is used, the transaction is aborted.
- No transaction importing or exporting occurs.

The Transaction Manager, the Transaction Synchronization Registry, and UserTransaction

You can access the Application Server transaction manager, a `javax.transaction.TransactionManager` implementation, using the JNDI subcontext `java:comp/TransactionManager` or `java:appserver/TransactionManager`. You can access the Application Server transaction synchronization registry, a `javax.transaction.TransactionSynchronizationRegistry` implementation, using the JNDI subcontext `java:comp/TransactionSynchronizationRegistry` or `java:appserver/TransactionSynchronizationRegistry`. You can also request injection of a `TransactionManager` or `TransactionSynchronizationRegistry` object using the `@Resource` annotation. Accessing the transaction synchronization registry is recommended. For details, see Java Specification Request (JSR) 907 (`http://www.jcp.org/en/jsr/detail?id=907`).

You can also access `java:comp/UserTransaction`.

◆ ◆ ◆ CHAPTER 11

Using the Java Naming and Directory Interface

A *naming service* maintains a set of bindings, which relate names to objects. The Java EE naming service is based on the Java Naming and Directory Interface™ (JNDI) API. The JNDI API allows application components and clients to look up distributed resources, services, and EJB components. For general information about the JNDI API, see `http://java.sun.com/products/jndi/`.

You can also see the JNDI tutorial at `http://java.sun.com/products/jndi/tutorial/`.

This chapter contains the following sections:

Note – For GlassFish v3 Technology Preview 2, EJB modules are not supported unless the optional EJB container module is downloaded from the Update Center. For information about the Update Center, see the *GlassFish v3 Application Server Quick Start Guide*.

For GlassFish v3 Technology Preview 2, only stateless session beans with local interfaces and entity beans that use the Java Persistence API are supported. Stateful, message-driven, and EJB 2.0 and 2.1 entity beans are not supported. Remote interfaces and remote business interfaces for any of the bean types are not supported.

Accessing the Naming Context

The Application Server provides a naming environment, or *context*, which is compliant with standard Java EE requirements. A `Context` object provides the methods for binding names to objects, unbinding names from objects, renaming objects, and listing the bindings. The `InitialContext` is the handle to the Java EE naming service that application components and clients use for lookups.

The JNDI API also provides subcontext functionality. Much like a directory in a file system, a subcontext is a context within a context. This hierarchical structure permits better organization of information. For naming services that support subcontexts, the `Context` class also provides methods for creating and destroying subcontexts.

Note – 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.

Global JNDI Names

Global JNDI names are assigned according to the following precedence rules:

1. A global JNDI name assigned in the `sun-ejb-jar.xml`, `sun-web.xml` deployment descriptor file has the highest precedence. See "Mapping References" on page 122.
2. A global JNDI name assigned in a `mapped-name` element in the `ejb-jar.xml`, `web.xml` deployment descriptor file has the second highest precedence. The following elements have `mapped-name` subelements: `resource-ref`, `resource-env-ref`, `ejb-ref`, `session`, and `entity`.
3. A global JNDI name assigned in a `mappedName` attribute of an annotation has the third highest precedence. The following annotations have `mappedName` attributes: `@javax.annotation.Resource`, `@javax.ejb.EJB`, `@javax.ejb.Stateless`.
4. A default global JNDI name is assigned in some cases if no name is assigned in deployment descriptors or annotations.
   - For 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`.

Mapping References

The following XML elements in the Application Server deployment descriptors map resource references in EJB and web application components to JNDI names configured in the Application Server:

- `resource-env-ref` - Maps the `@Resource` or `@Resources` annotation (or the `resource-env-ref` element in the corresponding Java EE XML file) to the absolute JNDI name configured in the Application Server.
- `resource-ref` - Maps the `@Resource` or `@Resources` annotation (or the `resource-ref` element in the corresponding Java EE XML file) to the absolute JNDI name configured in the Application Server.

- `ejb-ref` - Maps the `@EJB` annotation (or the `ejb-ref` element in the corresponding Java EE XML file) to the absolute JNDI name configured in the Application Server.

  JNDI names for EJB components must be unique. For example, appending the application name and the module name to the EJB name is one way to guarantee unique names. In this case, `mycompany.pkging.pkgingEJB.MyEJB` would be the JNDI name for an EJB in the module `pkgingEJB.jar`, which is packaged in the `pkging.ear` application.

These elements are part of the `sun-web.xml` and `sun-ejb-ref.xml` deployment descriptor files. For more information about how these elements behave in each of the deployment descriptor files, see Appendix A, "Deployment Descriptor Files," in *GlassFish v3 Application Server Application Deployment Guide*.

The rest of this section uses an example of a JDBC resource lookup to describe how to reference resource factories.

The `@Resource` annotation in the application code looks like this:

```
@Resource(name="jdbc/helloDbDs") javax.sql.DataSource ds;
```

This references a resource with the JNDI name of `java:comp/env/jdbc/helloDbDs`. If this is the JNDI name of the JDBC resource configured in the Application Server, the annotation alone is enough to reference the resource.

However, you can use an Application Server specific deployment descriptor to override the annotation. For example, the `resource-ref` element in the `sun-web.xml` file maps the `res-ref-name` (the name specified in the annotation) to the JNDI name of another JDBC resource configured in the Application Server.

```
<resource-ref>
   <res-ref-name>jdbc/helloDbDs</res-ref-name>
   <jndi-name>jdbc/helloDbDataSource</jndi-name>
</resource-ref>
```

Index

G

H

I

J

T

U

V

W

X