Not For Publication

Sun Java System Application Server Platform Edition 9 Developer's Guide

Beta

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

Part No: 819–3659

Contents

Figures

Tables

Examples

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 Application Programming Interfaces (APIs) in the Sun Java System Application Server (Application Server) environment. Topics include developer tools, security, debugging, and creating lifecycle modules.

Who Should Use This Book

This book is intended for use by software developers who create, assemble, and deploy Java EE applications using Sun Java System servers and software. Application Server software developers should already understand the following technologies:

- Java technology
- Java EE platform version 5
- Hypertext Transfer Protocol (HTTP)
- Hypertext Markup Language (HTML)
- Extensible Markup Language (XML)

How This Book Is Organized

The *Developer's Guide* has three parts.

- Part I includes general development topics relevant to the Application Server, such as class loaders and debugging.
- Part II describes Java EE application components, such as servlets and message-driven beans, that can run on the Application Server.
- Part III describes services and APIs that provide Application Server resources, such as Java DataBase Connectivity (JDBC™).

The following table summarizes the chapters in this book.

TABLE P–1 How This Book Is Organized

Chapter	Description
Chapter 1	Describes setting up an application development environment in the Application Server.
Chapter 2	Describes Application Server class loaders.
Chapter 3	Describes how to use the `asant` utility, which provides Ant tasks specific to the Application Server.
Chapter 4	Provides guidelines for debugging applications in the Application Server.
Chapter 5	Explains how to write secure Java EE applications, which contain components that perform user authentication and access authorization.
Chapter 6	Describes Application Server support for web services.
Chapter 7	Describes Application Server support for Java persistence.
Chapter 8	Describes how web applications are supported in the Application Server.
Chapter 9	Describes how Enterprise JavaBeans™ (EJB™) technology is supported in the Application Server.
Chapter 10	Provides information on how container-managed persistence (CMP) works in the Application Server.
Chapter 11	Describes how to develop and assemble Java EE application clients.
Chapter 12	Describes Application Server support for the J2EE 1.5 Connector architecture.
Chapter 13	Describes how to create and use a lifecycle listener module.
Chapter 14	Describes Application Server support for custom MBeans.
Chapter 15	Explains how to use the JDBC API for database access with the Application Server.
Chapter 16	Describes Java EE transactions and transaction support in the Application Server.
Chapter 17	Explains how to use the Java Naming and Directory Interface™ (JNDI) API for naming and references.
Chapter 18	Explains how to use the Java Message Service (JMS) API, and describes the Application Server's fully integrated JMS provider: the Sun Java System Message Queue software.
Chapter 19	Explains how to use the JavaMail™ API.
Chapter 20	Explains how to use the Java Management Extensions (JMX™) API.

Application Server Documentation Set

The Application Server documentation set describes deployment planning and system installation. The Uniform Resource Locator (URL) for stand-alone Application Server documentation is `http://docs.sun.com/app/docs/coll/1343.3`. For an introduction to Application Server, refer to the books in the order in which they are listed in the following table.

TABLE P–2 Books in the Application Server Documentation Set

Book Title	Description
Documentation Center	Application Server documentation topics organized by task and subject.
Release Notes	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	How to get started with the Application Server product.
Installation Guide	Installing the software and its components.
Application Deployment Guide	Deployment of applications and application components to the Application Server. Includes information about deployment descriptors.
Developer's Guide	Creating and implementing Java Platform, Enterprise Edition (Java EE platform) applications intended to run on the Application Server that follow the open Java standards model for Java EE components and APIs. Includes information about developer tools, security, debugging, and creating lifecycle modules.
Java EE 5 Tutorial	Using Java EE 5 platform technologies and APIs to develop Java EE applications.
Administration Guide	Configuring, managing, and deploying Application Server subsystems and components from the Admin Console.
Administration Reference	Editing the Application Server configuration file, `domain.xml`.
Upgrade and Migration Guide	Migrating your applications to the new Application Server programming model, specifically from Application Server 6.x, and 7.x, and 8.x. This guide also describes differences between adjacent product releases and configuration options that can result in incompatibility with the product specifications.
Troubleshooting Guide	Solving Application Server problems.
Error Message Reference	Solving Application Server error messages.
Reference Manual	Utility commands available with the Application Server; written in man page style. Includes the `asadmin` command line interface.

Related Books

For documentation about other stand-alone Sun Java System server products, go to the following:

- Message Queue documentation (`http://docs.sun.com/app/docs/coll/1343.3`)
- Directory Server documentation (`http://docs.sun.com/app/docs/coll/1316.1`)
- Web Server documentation (`http://docs.sun.com/app/docs/coll/1308.1`)

You can find a directory of the official Java EE 5 specifications at `http://java.sun.com/javaee/5/javatech.html`. Additionally, the following resources might be useful.

General Java EE Online Information:

The Java EE 5 Tutorial (`http://java.sun.com/javaee/5/docs/tutorial/doc/index.html`)

The Java EE Blueprints (`http://java.sun.com/reference/blueprints/index.html`)

General Java EE Books:

Core J2EE Patterns: Best Practices and Design Strategies by Deepak Alur, John Crupi, & Dan Malks, Prentice Hall Publishing

Java Security, by Scott Oaks, O'Reilly Publishing

Books on Programming With servlets and JavaServer Pages™ (JSP™) Files:

Java Servlet Programming, by Jason Hunter, O'Reilly Publishing

Java Threads, 2nd Edition, by Scott Oaks & Henry Wong, O'Reilly Publishing

Books on Programming With EJB Components:

Enterprise JavaBeans, by Richard Monson-Haefel, O'Reilly Publishing

Books on Programming With JDBC:

Database Programming with JDBC and Java, by George Reese, O'Reilly Publishing

JDBC Database Access With Java: A Tutorial and Annotated Reference (Java Series), by Graham Hamilton, Rick Cattell, & Maydene Fisher

The Javadoc™ Tool:

A Javadoc tool reference for packages provided with the Application Server is located at `http://glassfish.dev.java.net/nonav/javaee5/api/index.html`.

Default Paths and File Names

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

TABLE P–3 Default Paths and File Names

Placeholder	Description	Default Value
install-dir	Represents the base installation directory for Application Server.	Solaris™ and Linux operating system installations, non-root user: *user's-home-directory*`/SUNWappserver` Solaris and Linux installations, root user: `/opt/SUNWappserver` Windows, all installations: *SystemDrive*`:\Sun\AppServer`
domain-root-dir	Represents the directory containing all domains.	*install-dir*`/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-dir*

Typographic Conventions

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

TABLE P–4 Typographic Conventions

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

TABLE P–4 Typographic Conventions *(Continued)*

Typeface	Meaning	Example
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–5 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.

Accessing Sun Resources Online

The docs.sun.com[SM] web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. Books are available as online files in PDF and HTML formats. Both formats are readable by assistive technologies for users with disabilities.

To access the following Sun resources, go to `http://www.sun.com`:

- Downloads of Sun products
- Services and solutions
- Support (including patches and updates)
- Training
- Research

- Communities (for example, Sun Developer Network)

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 819-3659.

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

For more information about stand-alone Application Server installation, see the *Sun Java System Application Server Platform Edition 9 Installation Guide*.

The following components are included in the full installation.

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

The NetBeans™ Integrated Development Environment (IDE) bundles the Platform 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 System Class Loader" on page 43 or "Using the Common Class Loader" on page 43.
- Set up debugging. For more information, see Chapter 4.
- Configure the Java Virtual Machine (JVM) software. For more information, see Chapter 19, "Java Virtual Machine and Advanced Settings," in *Sun Java System Application Server Platform Edition 9 Administration Guide*.

The GlassFish Project

Application Server Platform Edition 9 is developed through the GlassFish[SM] 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.

Development Tools

The following general tools are provided with the Application Server:

- "The `asadmin` Command" on page 33
- "The Admin Console" on page 33
- "The `asant` Utility" on page 33
- "The `verifier` Tool" on page 33

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

- "The NetBeans IDE" on page 34
- "The Migration Tool" on page 34

The following third-party tools might also be useful:

- "Debugging Tools" on page 34
- "Profiling Tools" on page 34
- "The Eclipse IDE" on page 34

The `asadmin` Command

The `asadmin` command allows you to configure a local or remote server and perform both administrative and development tasks at the command line. For general information about `asadmin`, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

The `asadmin` command is located in the *install-dir*`/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` 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 `asant` Utility

Apache Ant 1.6.5 is provided with the Application Server and can be launched from the `bin` directory using the command `asant`. The Application Server also provides server-specific tasks for administration and deployment; see Chapter 3. The sample applications that can be used with the Application Server use Ant `build.xml` files; see "Sample Applications" on page 35.

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

The `verifier` Tool

The `verifier` tool checks a Java EE application file, including Java classes and deployment descriptors, for compliance with Java EE specifications. Java EE application files are Java archive (JAR), web archive (WAR), resource adapter archive (RAR), or enterprise archive (EAR) files. Use the `verifier` tool to check whether an application complies with the Java EE specification and to make applications portable across application servers. The `verifier` tool can be launched from the command line. For more information, see "The verifier Utility" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

The NetBeans IDE

The NetBeans IDE allows you to create, assemble, and debug code from a single, easy-to-use interface. The Platform 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`.

For additional information on migration, see the *Sun Java System Application Server Platform Edition 9 Upgrade and Migration Guide*.

Debugging Tools

You can use several debugging tools with the Application Server. For more information, see Chapter 4.

Profiling Tools

You can use several profilers with the Application Server. For more information, see "Profiling Tools" on page 64.

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:// http://java.sun.com/j2ee/reference/codesamples/`) 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 `build.xml` file defines `asant` targets for the sample. See Chapter 3.
- 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:

The Class Loader Hierarchy

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

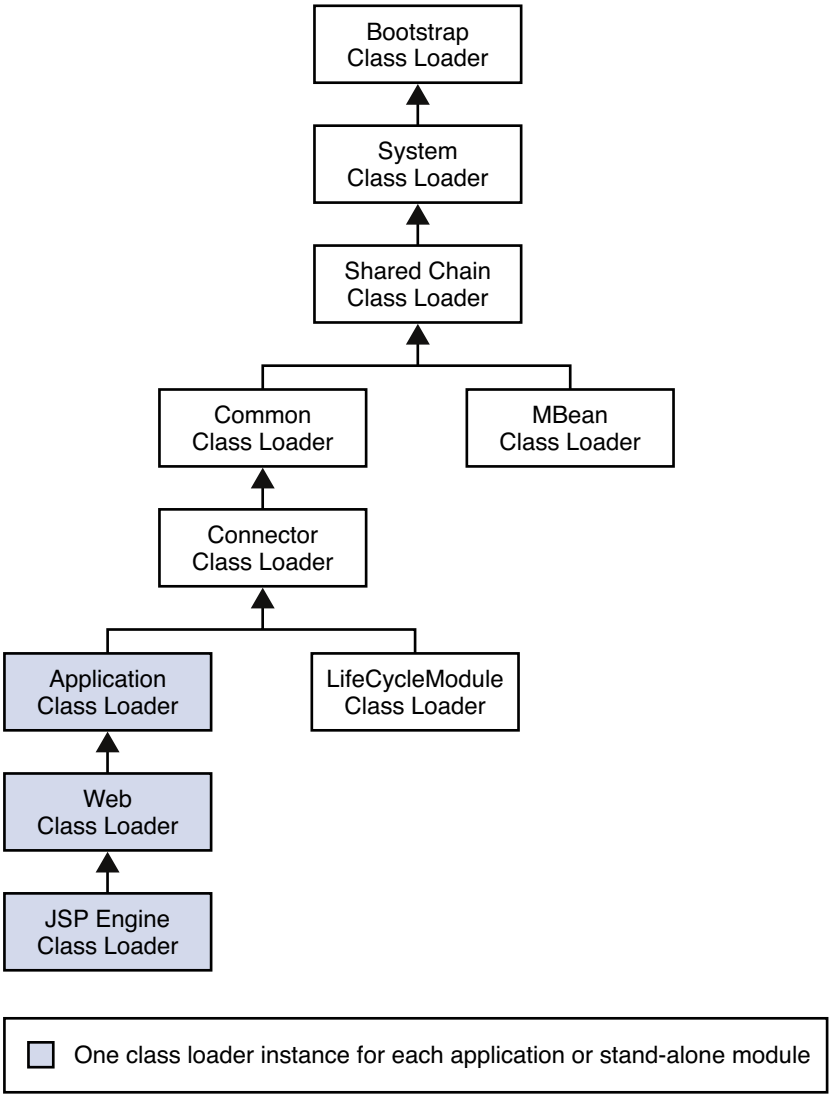

FIGURE 2–1 Class Loader Runtime Hierarchy

The following table describes the class loaders in the Application Server.

TABLE 2–1 Sun Java System Application Server Class Loaders

Class Loader	Description
Bootstrap	The Bootstrap class loader loads the basic runtime classes provided by the JVM, plus any classes from JAR files present in the system extensions directory. It is parent to the System class loader. To add JAR files to the system extensions, directory, see "Using the Java Optional Package Mechanism" on page 41.
System	The System class loader loads Application Server launch classes. It is parent to the Shared Chain class loader. It is created based on the `system-classpath` attribute of the `java-config` element in the `domain.xml` file. In the Admin Console, select the Application Server component, the JVM Settings tab, and the Path Settings tab, then edit the System Classpath field. See "Using the System Class Loader" on page 43 and "java-config" in *Sun Java System Application Server Platform Edition 9 Administration Reference*.
Shared Chain	The Shared Chain class loader loads most of the core Application Server classes. It is parent to the MBean class loader and the Common class loader. Classes specified by the `classpath-prefix` and `classpath-suffix` attributes of the `java-config` element in the `domain.xml` file are added to this class loader. In the Admin Console, select the Application Server component, the JVM Settings tab, and the Path Settings tab, then edit the Classpath Prefix or Classpath Suffix field. The environment classpath is included if `env-classpath-ignored="false"` is set in the `java-config` element. Use `classpath-prefix` to place libraries ahead of Application Server implementation classes in the shared chain. The `classpath-prefix` is ideal for placing development and diagnostic patches. Use `classpath-suffix` to place libraries after implementation classes in the shared chain.
MBean	The MBean class loader loads the MBean implementation classes. See "MBean Class Loading" on page 203.
Common	The Common 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 Connector class loader. No special classpath settings are required. The existence of these directories is optional; if they do not exist, the Common class loader is not created. See "Using the Common Class Loader" on page 43.
Connector	The Connector class loader is a single class loader instance that loads individually deployed connector modules, which are shared across all applications. It is parent to the LifeCycleModule class loader and the Application class loader.
LifeCycleModule	The LifeCycleModule class loader is created once per lifecycle module. Each `lifecycle-module` element's `classpath` attribute is used to construct its own class loader. For more information on lifecycle modules, see Chapter 13.

TABLE 2–1 Sun Java System Application Server Class Loaders *(Continued)*

Class Loader	Description
Application	The Application class loader loads the classes in a specific enabled individually deployed module or Java EE application. One instance of this class loader is present in each class loader universe; see "Class Loader Universes" on page 41. The Application 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 class loader loads classes in the following order: 1. Classes specified by the `library-directory` element in the `application.xml` deployment descriptor or the `--libraries` option during deployment; see "Application-Specific Class Loading" on page 42 2. Classes specified by the application's or module's `location` attribute in the `domain.xml` file, determined during deployment 3. Classes in the classpaths of the application's sub-modules 4. Classes in the application's or module's stubs directory The `location` attribute points to *domain-dir*`/applications/j2ee-apps/`*app-name* or *domain-dir*`/applications/j2ee-modules/`*module-name*. The stubs directory is *domain-dir*`/generated/ejb/j2ee-apps/`*app-name* or *domain-dir*`/generated/ejb/j2ee-modules/`*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 41. 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 41. The JSP Engine class loader is created with a list of URLs that point to the locations of the classes it needs to load.

Note that this 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 *Sun Java System Application Server Platform Edition 9 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 "The sun-web.xml File" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

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

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

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

- A directory pointed to by the Libraries field or `--libraries` option used during deployment
- A directory pointed to by the `library-directory` element in the `application.xml` deployment descriptor
- A directory pointed to by the 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`

Note – In iPlanet™ Application Server 6.x, individually deployed modules shared the same class loader. In subsequent Application Server versions, each individually deployed module has its own class loader universe.

Application-Specific Class Loading

You can specify application-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 application or module. Type the path in the Libraries field. For details, click the Help button in the Admin Console.
- Use the `asadmin deploy` command with the `--libraries` option. For details, see the *Sun Java System Application Server Platform Edition 9 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).

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

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

If multiple applications or modules refer to the same libraries, classes in those libraries are automatically shared. This can reduce the memory footprint and allow sharing of static information. However, applications or modules using application-specific libraries are not portable. Other ways to make libraries available are described in "Circumventing Class Loader Isolation" on page 43.

For general information about deployment, see the *Sun Java System Application Server Platform Edition 9 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 81.

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 accessed by more than one application, you can include the relevant path to the required classes in one of these ways:

- "Using the System Class Loader" on page 43
- "Using the Common Class Loader" on page 43
- "Packaging the Client JAR for One Application in Another Application" on page 44

Using the System class loader or Common class loader requires a server restart and makes a library accessible to any other application or module across the domain.

Using the System Class Loader

To use the System class loader, do one of the following, then restart the server:

- Use the Admin Console. Select the Application Server component, select the JVM Settings tab, select the Path Settings tab, and edit the System Classpath field. For details, click the Help button in the Admin Console.
- Edit the `system-classpath` attribute of the `java-config` element in the `domain.xml` file. For details about `domain.xml`, see the *Sun Java System Application Server Platform Edition 9 Administration Reference*.

Using the System class loader makes an application or module accessible to any other application or module across the domain.

Using the Common Class Loader

To use the Common 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 Common class loader makes an application or module accessible to any other application or module across the domain.

For example, using the Common 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 *Sun Java System Application Server Platform Edition 9 Release Notes*. For configurations of supported and other drivers, see "Configurations for Specific JDBC Drivers" in *Sun Java System Application Server Platform Edition 9 Administration Guide*.

Packaging the Client JAR for One Application in Another Application

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

As an alternative for a production environment, you can have the Common class loader load the client JAR of the dependent application as described in "Using the Common Class Loader" on page 43. Restart the server to make the dependent application accessible across the domain.

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

1 Deploy the dependent application.

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

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

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

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

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

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

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

4 Deploy the calling application.

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

You do not need to restart the server.

◆ ◆ ◆ CHAPTER 3

The asant Utility

Apache Ant 1.6.5 is provided with Application Server and can be launched from the `bin` directory using the command `asant`. The Application Server also provides server-specific tasks, which are described in this section.

Make sure you have done these things before using `asant`:

1. Include *install-dir*`/bin` in the PATH environment variable (`/usr/sfw/bin` for Sun Java Enterprise System on Solaris). The Ant script provided with the Application Server, `asant`, is located in this directory. For details on how to use `asant`, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.
2. If you are executing platform-specific applications, such as the `exec` or `cvs` task, the ANT_HOME environment variable must be set to the Ant installation directory.
   - The ANT_HOME environment variable for Sun Java Enterprise System must include the following paths.
     - `/usr/sfw/bin` – the Ant binaries (shell scripts)
     - `/usr/sfw/doc/ant` – HTML documentation
     - `/usr/sfw/lib/ant` – Java classes that implement Ant
   - The ANT_HOME environment variable for all other platforms is *install-dir*`/lib`.
3. Set up your password file. The argument for the `passwordfile` option of each Ant task is a file. This file contains the password in the following format.

   ```
   AS_ADMIN_PASSWORD=password
   ```

   For more information about password files, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

This section covers the following `asant`-related topics:

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

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

Application Server asant Tasks

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

The sun-appserv-deploy Task

Deploys any of the following.

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

Subelements of sun-appserv-deploy

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

TABLE 3–1 The sun-appserv-deploy Subelements

Element	Description
"The component Subelement" on page 59	A component to be deployed
"The fileset Subelement" on page 60	A set of component files that match specified parameters

Attributes of sun-appserv-deploy

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

TABLE 3–2 The sun-appserv-deploy Attributes

Attribute	Default	Description
file	none	(optional if a component or fileset subelement is present, otherwise required) The component to deploy. If this attribute refers to a file, it must be a valid archive. If this attribute refers to a directory, it must contain a valid archive in which all components have been exploded. If upload is set to false, this must be an absolute path on the server machine.
name	file name without extension	(optional) The display name for the component being deployed.
force	true	(optional) If true, the component is overwritten if it already exists on the server. If false, sun-appserv-deploy fails if the component exists.
retrievestubs	client stubs not saved	(optional) The directory where client stubs are saved. This attribute is inherited by nested component elements.
precompilejsp	false	(optional) If true, all JSP files found in an enterprise application (.ear) or web application (.war) are precompiled. This attribute is ignored for other component types. This attribute is inherited by nested component elements.
verify	false	(optional) If true, syntax and semantics for all deployment descriptors are automatically verified for correctness. This attribute is inherited by nested component elements.
contextroot	file name without extension	(optional) The context root for a web module (WAR file). This attribute is ignored if the component is not a WAR file.
dbvendorname	sun-ejb-jar.xml entry	(optional) The name of the database vendor for which tables can be created. Allowed values are javadb, db2, mssql, oracle, postgresql, pointbase, derby (also for CloudScape), and sybase, case-insensitive. If not specified, the value of the database-vendor-name attribute in sun-ejb-jar.xml is used. If no value is specified, a connection is made to the resource specified by the jndi-name subelement of the cmp-resource element in the sun-ejb-jar.xml file, and the database vendor name is read. If the connection cannot be established, or if the value is not recognized, SQL-92 compliance is presumed. For details, see "Generation Options for CMP" on page 158.
createtables	sun-ejb-jar.xml entry	(optional) If true, causes database tables to be created for beans that need them. If false, does not create tables. If not specified, the value of the create-tables-at-deploy attribute in sun-ejb-jar.xml is used. For details, see "Generation Options" on page 108 and "Generation Options for CMP" on page 158.

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

Attribute	Default	Description
dropandcreatetables	sun-ejb-jar.xml entry	(optional) If true, and if tables were automatically created when this application was last deployed, tables from the earlier deployment are dropped and fresh ones are created. If true, and if tables were *not* automatically created when this application was last deployed, no attempt is made to drop any tables. If tables with the same names as those that would have been automatically created are found, the deployment proceeds, but a warning indicates that tables could not be created. If false, settings of create-tables-at-deploy or drop-tables-at-undeploy in the sun-ejb-jar.xml file are overridden. For details, see "Generation Options" on page 108 and "Generation Options for CMP" on page 158.
uniquetablenames	sun-ejb-jar.xml entry	(optional) If true, specifies that table names are unique within each application server domain. If not specified, the value of the use-unique-table-names property in sun-ejb-jar.xml is used. For details, see "Generation Options for CMP" on page 158.
enabled	true	(optional) If true, enables the component.
deploymentplan	none	(optional) A deployment plan is a JAR file containing Sun-specific descriptors. Use this attribute when deploying an EAR file that lacks Sun-specific descriptors.
upload	true	(optional) If true, the component is transferred to the server for deployment. If the component is being deployed on the local machine, set upload to false to reduce deployment time. If a directory is specified for deployment, upload must be false.
virtualservers	default virtual server only	(optional) A comma-separated list of virtual servers to be deployment targets. This attribute applies only to application (.ear) or web (.war) components and is ignored for other component types.
user	admin	(optional) The user name used when logging into the application server.
passwordfile	none	(optional) File containing passwords. The password from this file is retrieved for communication with the application server.
host	localhost	(optional) Target server. When deploying to a remote server, use the fully qualified host name.
port	4848	(optional) The administration port on the target server.
asinstalldir	see description	(optional) The installation directory for the local Application Server installation, which is used to find the administrative classes. If not specified, the command checks if the asinstalldir parameter has been set. Otherwise, administrative classes must be in the system classpath.

Examples of sun-appserv-deploy

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

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

Here is an equivalent script showing all the implied attributes:

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

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

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

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

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

The `sun-appserv-undeploy` Task

Undeploys any of the following.

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

- Application client

Subelements of sun-appserv-undeploy

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

TABLE 3–3 The sun-appserv-undeploy Subelements

Element	Description
"The component Subelement" on page 59	A component to be deployed
"The fileset Subelement" on page 60	A set of component files that match specified parameters

Attributes of sun-appserv-undeploy

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

TABLE 3–4 The sun-appserv-undeploy Attributes

Attribute	Default	Description
name	file name without extension	(optional if a component or fileset subelement is present or the file attribute is specified, otherwise required) The display name for the component being undeployed.
file	none	(optional) The component to undeploy. If this attribute refers to a file, it must be a valid archive. If this attribute refers to a directory, it must contain a valid archive in which all components have been exploded.
droptables	sun-ejb-jar.xml entry	(optional) If true, causes database tables that were automatically created when the bean(s) were last deployed to be dropped when the bean(s) are undeployed. If false, does not drop tables. If not specified, the value of the drop-tables-at-undeploy attribute in sun-ejb-jar.xml is used. For details, see "Generation Options" on page 108 and "Generation Options for CMP" on page 158.
cascade	false	(optional) If true, deletes all connection pools and connector resources associated with the resource adapter being undeployed. If false, undeployment fails if any pools or resources are still associated with the resource adapter. This attribute is applicable to connectors (resource adapters) and applications with connector modules.
user	admin	(optional) The user name used when logging into the application server.

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

Attribute	Default	Description
passwordfile	none	(optional) File containing passwords. The password from this file is retrieved for communication with the application server.
host	localhost	(optional) Target server. When deploying to a remote server, use the fully qualified host name.
port	4848	(optional) The administration port on the target server.
asinstalldir	see description	(optional) The installation directory for the local Application Server installation, which is used to find the administrative classes. If not specified, the command checks to see if the asinstalldir parameter has been set. Otherwise, administrative classes must be in the system classpath.

Examples of sun-appserv-undeploy

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

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

Here is an equivalent script showing all the implied attributes:

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

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

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

The sun-appserv-component Task

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

- Enterprise application (EAR file)
- Web application (WAR file)

- Enterprise Java Bean (EJB-JAR file)
- Enterprise connector (RAR file)
- Application client

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

Subelements of sun-appserv-component

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

TABLE 3–5 The sun-appserv-component Subelements

Element	Description
"The component Subelement" on page 59	A component to be deployed
"The fileset Subelement" on page 60	A set of component files that match specified parameters

Attributes of sun-appserv-component

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

TABLE 3–6 The sun-appserv-component Attributes

Attribute	Default	Description
action	none	The control command for the target application server. Valid values are enable and disable.
name	file name without extension	(optional if a component or fileset subelement is present or the file attribute is specified, otherwise required) The display name for the component being enabled or disabled.
file	none	(optional) The component to enable or disable. If this attribute refers to a file, it must be a valid archive. If this attribute refers to a directory, it must contain a valid archive in which all components have been exploded.
user	admin	(optional) The user name used when logging into the application server.
passwordfile	none	(optional) File containing passwords. The password from this file is retrieved for communication with the application server.
host	localhost	(optional) Target server. When enabling or disabling a remote server, use the fully qualified host name.
port	4848	(optional) The administration port on the target server.
asinstalldir	see description	(optional) The installation directory for the local Application Server installation, which is used to find the administrative classes. If not specified, the command checks to see if the asinstalldir parameter has been set. Otherwise, administrative classes must be in the system classpath.

Examples of sun-appserv-component

Here is a simple example of disabling a component:

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

Here is a simple example of enabling a component:

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

Here is an equivalent script showing all the implied attributes:

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

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

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

The sun-appserv-admin Task

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

Attributes of sun-appserv-admin

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

TABLE 3–7 The sun-appserv-admin Attributes

Attribute	Default	Description
command	none	(exactly one of these is required: command or explicitcommand) The command to execute. If the user, passwordfile, host, or port attributes are also specified, they are automatically inserted into the command before execution. If any of these options are specified in the command string, the corresponding attribute values are ignored.
explicitcommand	none	(exactly one of these is required: command or explicitcommand) The exact command to execute. No command processing is done, and all other attributes are ignored.
user	admin	(optional) The user name used when logging into the application server.
passwordfile	none	(optional) File containing passwords. The password from this file is retrieved for communication with the application server.
host	localhost	(optional) Target server. If it is a remote server, use the fully qualified host name.
port	4848	(optional) The administration port on the target server.
asinstalldir	see description	(optional) The installation directory for the local Application Server installation, which is used to find the administrative classes. If not specified, the command checks if the asinstalldir parameter has been set. Otherwise, administrative classes must be in the system classpath.

Examples of sun-appserv-admin

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

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

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

```
<sun-appserv-admin command="create-jms-dest --user adminuser --host localhost
--port 4848 --desttype topic --target server1 simpleJmsDest">
```

The sun-appserv-jspc Task

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

Attributes of sun-appserv-jspc

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

TABLE 3–8 The sun-appserv-jspc Attributes

Attribute	Default	Description
destdir		The destination directory for the generated Java source files.
srcdir		(exactly one of these is required: srcdir or webapp) The source directory where the JSP files are located.
webapp		(exactly one of these is required: srcdir or webapp) The directory containing the web application. All JSP files within the directory are recursively parsed. The base directory must have a WEB-INF subdirectory beneath it. When webapp is used, sun-appserv-jspc hands off all dependency checking to the compiler.
verbose	2	(optional) The verbosity integer to be passed to the compiler.
classpath		(optional) The classpath for running the JSP compiler.
classpathref		(optional) A reference to the JSP compiler classpath.
uribase	/	(optional) The URI context of relative URI references in the JSP files. If this context does not exist, it is derived from the location of the JSP file relative to the declared or derived value of uriroot. Only pages translated from an explicitly declared JSP file are affected.
uriroot	see description	(optional) The root directory of the web application, against which URI files are resolved. If this directory is not specified, the first JSP file is used to derive it: each parent directory of the first JSP file is searched for a WEB-INF directory, and the directory closest to the JSP file that has one is used. If no WEB-INF directory is found, the directory from which sun-appserv-jspc was called is used. Only pages translated from an explicitly declared JSP file (including tag libraries) are affected.
package		(optional) The destination package for the generated Java classes.
asinstalldir	see description	(optional) The installation directory for the local Application Server installation, which is used to find the administrative classes. If not specified, the command checks if the asinstalldir parameter has been set. Otherwise, administrative classes must be in the system classpath.

Example of sun-appserv-jspc

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

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

```
   appserv-rt.jar:${asinstalldir}/lib/appserv-ext.jar">
 <include name="**/*.java"/>
</javac>
```

The `sun-appserv-update` Task

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

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

For more information about dynamic reloading, see the *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

Attributes of `sun-appserv-update`

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

TABLE 3–9 The `sun-appserv-update` Attributes

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

Example of `sun-appserv-update`

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

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

Reusable Subelements

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

- "The `component` Subelement" on page 59
- "The `fileset` Subelement" on page 60

The `component` Subelement

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

Attributes of `component`

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

TABLE 3–10 The `component` Attributes

Attribute	Default	Description
`file`	none	(optional if the parent task is "The `sun-appserv-undeploy` Task" on page 51 or "The `sun-appserv-component` Task" on page 53) The target component. If this attribute refers to a file, it must be a valid archive. If this attribute refers to a directory, it must contain a valid archive in which all components have been exploded. If `upload` is set to `false`, this must be an absolute path on the server machine.
`name`	file name without extension	(optional) The display name for the component.
`force`	`true`	(applies to "The `sun-appserv-deploy` Task" on page 48 only, optional) If `true`, the component is overwritten if it already exists on the server. If `false`, the containing element's operation fails if the component exists.
`precompilejsp`	`false`	(applies to "The `sun-appserv-deploy` Task" on page 48 only, optional) If `true`, all JSP files found in an enterprise application (`.ear`) or web application (`.war`) are precompiled. This attribute is ignored for other component types.
`retrievestubs`	client stubs not saved	(applies to "The `sun-appserv-deploy` Task" on page 48 only, optional) The directory where client stubs are saved.
`contextroot`	file name without extension	(applies to "The `sun-appserv-deploy` Task" on page 48 only, optional) The context root for a web module (WAR file). This attribute is ignored if the component is not a WAR file.
`verify`	`false`	(applies to "The `sun-appserv-deploy` Task" on page 48 only, optional) If `true`, syntax and semantics for all deployment descriptors is automatically verified for correctness.

Examples of `component`

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

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

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

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

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

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

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

The `fileset` Subelement

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

◆ ◆ ◆ CHAPTER 4

Debugging Applications

This chapter gives guidelines for debugging applications in the Sun Java System 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 62.

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. **Select the Application Server component and the JVM Settings tab in the Admin Console.**
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 62.**

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 *Sun Java System Application Server Platform Edition 9 Reference Manual*.

Sun Java System Message Queue Debugging

Sun Java System Message Queue has a broker logger, which can be useful for debugging Java Message Service (JMS) applications, including message-driven bean applications. You can adjust the logger's verbosity, and you can send the logger output to the broker's console using the broker's `-tty` option. For more information, see the *Sun Java System Message Queue 3 2006Q2 Administration Guide*.

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.

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-\ prints a thread dump. To start the server in verbose mode, use the `--verbose` option as follows:

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

Application Server Logging

You can use the Application Server's log files to help debug your applications. In the Admin Console, select the Application Server component, then click the View Log Files button in the General Information page. To change logging settings, select the Logging tab. For details about logging, click the Help button in the 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. **Using the Admin Console, select the Application Server component, the JVM Settings tab, and 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 Optimizeit Profiler

You can purchase Optimizeit from Borland at `http://www.borland.com/optimizeit`.

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

▼ To Enable Remote Profiling With Optimizeit

1 Configure your operating system:

- On Solaris, add *Optimizeit-dir*`/lib` to the LD_LIBRARY_PATH environment variable.
- On Windows, add *Optimizeit-dir*`/lib` to the PATH environment variable.

2 Configure the Application Server using the Admin Console:

a. Select the Application Server component, the JVM Settings tab, and the Profiler tab.

b. Edit the following fields:

- Profiler Name – `optimizeit`
- Profiler Enabled – `true`
- Classpath – *Optimizeit-dir*`/lib/optit.jar`
- Native Library Path – *Optimizeit-dir*`/lib`
- JVM Option – For each of these options, select Add, type the option in the Value field, then check its box

```
-DOPTITHOME=Optimizeit-dir  -Xrunpri:startAudit=t
-Xbootclasspath/p:/Optimizeit-dir/lib/oibcp.jar
```

3 In addition, you might have to set the following in your `server.policy` file.

For more information about the `server.policy` file, see "The `server.policy` File" on page 80.

```
grant codeBase "file:Optimizeit-dir/lib/optit.jar" {
    permission java.security.AllPermission;
};
```

4 Restart the Application Server.

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

See Also For further details, see the Optimizeit documentation.

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

The Wily Introscope Profiler

Information about Introscope® from Wily Technology is available at `http://www.wilytech.com/solutions_introscope.html`.

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

▼ To Enable Remote Profiling With Introscope

1 Using the Admin Console, select the Application Server component, the JVM Settings tab, and the Profiler tab.

2 Edit the following fields before clicking Save:

- Name – `wily`
- Enabled – `true`
- Classpath – *Wily-dir*`/ProbeBuilder.jar:`*Wily-dir*`/Agent.jar`
- Native Library Path – (leave blank)
- JVM Option – (leave blank)

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

See Also For further details, see the Introscope documentation.

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

The JProbe Profiler

Information about JProbe from Sitraka is available at `http://www.klgroup.com/software/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. Select the Application Server component, the JVM Settings tab, and 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 – *install-dir*
- Server class File – `com.sun.enterprise.server.J2EERunner`
- Working Directory – *install-dir*
- Classpath – *install-dir*`/lib/appserv-rt.jar`
- Source File Path – *source-code-dir* (in case you want to get the line level details)
- Server class arguments – (optional)
- Main Package – `com.sun.enterprise.server`

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

PART II

Developing Applications and Application Components

◆ ◆ ◆ CHAPTER 5

Securing Applications

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

For information about administrative security for the Application Server, see Chapter 8, "Configuring Security," in *Sun Java System Application Server Platform Edition 9 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 Sun Java System Application Server is to provide highly secure, interoperable, and distributed component computing based on the Java EE security model. Security goals include:

- Full compliance with the Java EE security model. This includes EJB and servlet role-based authorization.

- Support for single sign-on across all Application Server applications within a single security domain.
- Support for web services message security.
- Security support for application clients.
- Support for several underlying authentication realms, such as simple file and Lightweight Directory Access Protocol (LDAP). Certificate authentication is also supported for Secure Socket Layer (SSL) client authentication. For Solaris, OS platform authentication is supported in addition to these.
- Support for declarative security through 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`).

Application Server Specific Security Features

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

- Message security; see "Configuring Message Security for Web Services" on page 83
- Single sign-on across all Application Server applications within a single security domain; see "User Authentication for Single Sign-on" on page 94
- Programmatic login; see "Programmatic Login" on page 91

Container Security

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

- "Declarative Security" on page 72
- "Programmatic Security" on page 73

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 *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

There are two levels of declarative security, as follows:

- "Application Level Security" on page 73
- "Component Level Security" on page 73

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 74.

Component Level Security

Component level security encompasses web components and EJB components.

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

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

Programmatic Security

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

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

For more information on programmatic security, see the following:

- The Java EE Specification
- "Programmatic Login" on page 91

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 *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

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

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

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

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

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

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

You can define a default principal and a default principal to role mapping, each of which applies to the entire Application Server. 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.
- Use the `asadmin set` command. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*. For example, you can set the default principal as follows.

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

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

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

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.
- `ldap` – Stores user information in an LDAP directory.
- `certificate` – Sets up the user identity in the Application Server security context, and populates it with user data obtained from cryptographically verified client certificates.
- `solaris` – Allows authentication using Solaris `username+password` data. This realm is only supported on the Solaris operating system, version 9 and above.

For detailed information about configuring each of these realms, see "How to Configure a Realm" on page 76.

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 *Sun Java System Application Server Platform Edition 9 Reference Manual*.

How to Set a Realm for an Application or Module

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

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

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

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

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

For more information about the deployment descriptor files and elements, see Appendix A, "Deployment Descriptor Files," in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

Creating a Custom Realm

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

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

For general information about realms and login modules, see "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`).

For Javadoc tool pages relevant to custom realms, go to `http://glassfish.dev.java.net/nonav/javaee5/api/index.html` and click on the `com.sun.appserv.security` package.

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

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

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

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

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

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

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

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

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

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

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

```
public String getAuthType()
```

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

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

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

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. 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.

Pluggable Audit Module Support

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

- "Configuring an Audit Module" on page 78
- "The `AuditModule` Class" on page 79

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

Configuring an Audit Module

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

- To specify an audit module using the Admin Console, open the Security component under the relevant configuration, and select the Audit Modules component. For details, click the Help button in the Admin Console.
- You can use the `asadmin create-audit-module` command to configure an audit module. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

The `AuditModule` Class

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

For Javadoc tool pages relevant to audit modules, go to `http://glassfish.dev.java.net/nonav/javaee5/api/index.html` and click on the `com.sun.appserv.security` package.

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

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

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

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

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

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

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

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

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

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

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

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

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

The `server.policy` File

Each Application Server domain has its own standard 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/j2ee-apps/-" {
...
}
```

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/j2ee-apps/MyApp/-" {
...
}
```

For a module:

```
grant "file:${com.sun.aas.instanceRoot}/applications/j2ee-modules/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.

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 *Sun Java System Application Server Platform Edition 9 Administration Reference*.

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 information about Application Server system properties, see "system-property" in *Sun Java System Application Server Platform Edition 9 Administration Reference*.

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
  ```

  For more information about the `asadmin delete-jvm-options` command, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

Configuring Message Security for Web Services

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

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

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

For more information about message security, see the following:

- The Java EE 5 Tutorial (`http://java.sun.com/javaee/5/docs/tutorial/doc/index.html`) chapter titled "Chapter 29: Introduction to Security in Java EE"
- Chapter 9, "Configuring Message Security," in *Sun Java System Application Server Platform Edition 9 Administration Guide*
- The Liberty Alliance Project specifications at `http://www.projectliberty.org/resources/specifications.php`
- The Oasis Web Services Security (WSS) specification at `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf`

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

- "Message Security Providers" on page 84
- "Message Security Responsibilities" on page 85

- "Application-Specific Message Protection" on page 86
- "Understanding and Running the Sample Application" on page 89

Message Security Providers

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

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

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

For more information about the `asadmin set` command, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

The example described in "Understanding and Running the Sample Application" on page 89 uses the `ClientProvider` and `ServerProvider` providers, which are enabled when the `asant` targets are run. You don't need to enable these on the Application Server prior to running the example.

For information about configuring these providers in the Application Server, see Chapter 9, "Configuring Message Security," in *Sun Java System Application Server Platform Edition 9 Administration Guide*. For additional information about overriding provider settings, see "Application-Specific Message Protection" on page 86.

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

- To create a message security provider using the Admin Console, open the Security component under the relevant configuration, and select the Message Security component. For details, click the Help button in the Admin Console.
- You can use the `asadmin create-message-security-provider` command to create a message security provider. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

In addition, you can set a few optional provider properties. For more information, see the property descriptions under "provider-config" in *Sun Java System Application Server Platform Edition 9 Administration Reference*.

Message Security Responsibilities

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

- "Application Developer" on page 85
- "Application Deployer" on page 85
- "System Administrator" on page 85

Application Developer

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

The application developer is responsible for the following:

- Determining if an application-specific message protection policy is required by the application. If so, ensuring that the required policy is specified at application assembly which may be accomplished by communicating with the application deployer.
- Determining if message security is necessary at the Application Server level. If so, ensuring that this need is communicated to the system administrator, or taking care of implementing message security at the Application Server level.

Application Deployer

The application deployer is responsible for the following:

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

These security tasks are discussed in "Application-Specific Message Protection" on page 86. A sample application using message security is discussed in "Understanding and Running the Sample Application" on page 89.

System Administrator

The system administrator is responsible for the following:

- Configuring message security providers on the Application Server.
- Managing user databases.

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

A system administrator uses the Admin Console to manage server security settings and uses a command line tool to manage certificate databases. Certificates and private keys are stored in key stores and are managed with `keytool`. System administrator tasks are discussed in Chapter 9, "Configuring Message Security," in *Sun Java System Application Server Platform Edition 9 Administration Guide*.

Application-Specific Message Protection

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

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

For more information about message security providers, see "Message Security Providers" on page 84.

For more details on message security binding for EJB web services, servlet web services, and clients, see the XML file descriptions in Appendix A, "Deployment Descriptor Files," in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

- For `sun-ejb-jar.xml`, see "The sun-ejb-jar.xml File" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.
- For `sun-web.xml`, see "The sun-web.xml File" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.
- For `sun-application-client.xml`, see "The sun-application-client.xml file" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

This section contains the following topics:

Using a Signature to Enable Message Protection for All Methods

To enable message protection for all methods using digital signature, update the `message-security-binding` element for the EJB web service endpoint in the application's `sun-ejb-jar.xml` file. In this file, add `request-protection` and `response-protection` elements, which are analogous to the `request-policy` and `response-policy` elements discussed in Chapter 9, "Configuring Message Security," in *Sun Java System Application Server Platform Edition 9*

Administration Guide. To apply the same protection mechanisms for all methods, leave the method-name element blank. "Configuring Message Protection for a Specific Method Based on Digital Signatures" on page 87 discusses listing specific methods or using wildcard characters.

This section uses the sample application discussed in "Understanding and Running the Sample Application" on page 89 to apply application-level message security to show only the differences necessary for protecting web services using various mechanisms.

▼ To Enable Message Protection for All Methods Using Digital Signature

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

For the `xms` example, this file is located in the directory *app-dir*`/xms-ejb/src/conf`, where *app-dir* is defined in "To Set Up the Sample Application" on page 89.

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

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

3 Compile, deploy, and run the application as described in "To Run the Sample Application" on page 90.

Configuring Message Protection for a Specific Method Based on Digital Signatures

To enable message protection for a specific method, or for a set of methods that can be identified using a wildcard value, follow these steps. As in the example discussed in "Using a Signature to Enable Message Protection for All Methods" on page 86, to enable message protection for a specific method, update the `message-security-binding` element for the EJB web service endpoint in the application's `sun-ejb-jar.xml` file. To this file, add `request-protection` and

`response-protection` elements, which are analogous to the `request-policy` and `response-policy` elements discussed in Chapter 9, "Configuring Message Security," in *Sun Java System Application Server Platform Edition 9 Administration Guide*. The administration guide includes a table listing the set and order of security operations for different request and response policy configurations.

This section uses the sample application discussed in "Understanding and Running the Sample Application" on page 89 to apply application-level message security to show only the differences necessary for protecting web services using various mechanisms.

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

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

For the `xms` example, this file is located in the directory *app-dir*`/xms-ejb/src/conf`, where *app-dir* is defined in "To Set Up the Sample Application" on page 89.

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

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

3 **Compile, deploy, and run the application as described in "To Run the Sample Application" on page 90.**

Understanding and Running the Sample Application

This section discusses the WSS sample application. This sample application is installed on your system only if you installed the J2EE 1.4 samples. If you have not installed these samples, see "To Set Up the Sample Application" on page 89.

The objective of this sample application is to demonstrate how a web service can be secured with WSS. The web service in the `xms` example is a simple web service implemented using a Java EE EJB endpoint and a web service endpoint implemented using a servlet. In this example, a service endpoint interface is defined with one operation, `sayHello`, which takes a string then sends a response with `Hello` prefixed to the given string. You can view the WSDL file for the service endpoint interface at *app-dir*`/xms-ejb/src/conf/HelloWorld.wsdl`, where *app-dir* is defined in "To Set Up the Sample Application" on page 89.

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

This example shows how to use message security for web services at the Application Server level. For information about using message security at the application level, see "Application-Specific Message Protection" on page 86. The WSS message security mechanisms implement message-level authentication (for example, XML digital signature and encryption) of SOAP web services invocations using the X.509 and username/password profiles of the OASIS WS-Security standard, which can be viewed from the following URL: `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf`.

This section includes the following topics:

▼ To Set Up the Sample Application

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

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

1 **Go to the J2EE 1.4 download URL (`http://java.sun.com/j2ee/1.4/download.html`) in your browser.**

2 **Click on the Download button for the Samples Bundle.**

3. **Click on Accept License Agreement.**

4. **Click on the J2EE SDK Samples link.**

5. **Choose a location for the `j2eesdk-1_4_03-samples.zip` file.**

   Saving the file to *install-dir* is recommended.

6. **Unzip the file.**

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

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

▼ To Run the Sample Application

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

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

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

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

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

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

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

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

   a. **To compile samples:**

      ```
      asant
      ```

   b. **To deploy samples:**

      ```
      asant deploy
      ```

c. To run samples:

```
asant run
```

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

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

5 To undeploy the sample, run the following `asant` target:

```
asant undeploy
```

All of the web services security examples use the same web service name (`HelloWorld`) and web service ports. These examples show only the differences necessary for protecting web services using various mechanisms. Make sure to undeploy an application when you have completed running it. If you do not, you receive an `Already in Use` error and deployment failures when you try to deploy another web services example application.

Programmatic Login

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

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

This section contains the following topics:

Programmatic Login Precautions

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

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

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

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

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

Granting Programmatic Login Permission

The `ProgrammaticLoginPermission` permission is required to invoke the programmatic login mechanism for an application if the security manager is enabled. For information about the security manager, see "The `server.policy` File" on page 80. This permission is not granted by default to deployed applications because this is not a standard Java EE mechanism.

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

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

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

The `ProgrammaticLogin` Class

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

For Javadoc tool pages relevant to programmatic login, go to `http://glassfish.dev.java.net/nonav/javaee5/api/index.html` and click on the `com.sun.appserv.security` package.

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

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

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

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

The login methods for EJB components have the following signatures:

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

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

All of these `login` methods accomplish the following:

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

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

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

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

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

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

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

The logout methods for EJB components have the following signatures:

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

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

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

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

User Authentication for Single Sign-on

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

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

Single sign-on operates according to the following rules:

- Single sign-on applies to web applications configured for the same realm and virtual server. The realm is defined by the `realm-name` element in the `web.xml` file. For information about virtual servers, see Chapter 11, "Configuring the HTTP Service," in *Sun Java System Application Server Platform Edition 9 Administration Guide*.
- As long as users access only unprotected resources in any of the web applications on a virtual server, they are not challenged to authenticate themselves.
- As soon as a user accesses a protected resource in any web application associated with a virtual server, the user is challenged to authenticate himself or herself, using the login method defined for the web application currently being accessed.
- After authentication, the roles associated with this user are used for access control decisions across all associated web applications, without challenging the user to authenticate to each application individually.
- When the user logs out of one web application (for example, by invalidating the corresponding session), the user's sessions in all web applications are invalidated. Any subsequent attempt to access a protected resource in any application requires the user to authenticate again.

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

To configure single sign-on, set the following properties in the `virtual-server` element of the `domain.xml` file:

- `sso-enabled` - If `false`, single sign-on is disabled for this virtual server, and users must authenticate separately to every application on the virtual server. The default is `true`.
- `sso-max-inactive-seconds` - Specifies the time after which a user's single sign-on record becomes eligible for purging if no client activity is received. Since single sign-on applies across several applications on the same virtual server, access to any of the applications keeps the single

sign-on record active. The default value is 5 minutes (`300` seconds). Higher values provide longer single sign-on persistence for the users at the expense of more memory use on the server.

- `sso-reap-interval-seconds` - Specifies the interval between purges of expired single sign-on records. The default value is `60`.

Here is an example configuration with all default values:

```
<virtual-server id="server" ... >
     ...
    <property name="sso-enabled" value="true"/>
     <property name="sso-max-inactive-seconds" value="300"/>
     <property name="sso-reap-interval-seconds" value="60"/>
 </virtual-server>
```

◆ ◆ ◆ CHAPTER 6

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:

"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`).

For information about web services security, see "Configuring Message Security for Web Services" on page 83.

For information about web services administration, monitoring, logging, and registries, see Chapter 12, "Managing Web Services," in *Sun Java System Application Server Platform Edition 9 Administration Guide*.

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 *Sun Java System Application Server Platform Edition 9 Reference Manual*.

Deploying a Web Service

You deploy a web service endpoint to the Application Server just as you would any servlet, stateless session bean (SLSB), or application. 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 99.

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 files `sun-web.xml` and `sun-ejb-jar.xml` provide optional web service enhancements in their `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 100.

For more information about deployment, autodeployment, and deployment descriptors, see the *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*. For more information about the `asadmin deploy` command, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

Web Services Registry

You deploy a registry to the Application Server just as you would any connector 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 *Sun Java System Application Server Platform Edition 9 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 *Sun Java System Application Server Platform Edition 9 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 connector module that accesses UDDI registries is provided with the Application Server in the *install-dir*`/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 `application.xml` or `web.xml` file, and can be overridden in the `sun-application.xml` or `sun-web.xml` file. The *servlet-mapping-url-pattern* is defined in the `web.xml` file.

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

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

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

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

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

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

You can also test a service using the 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` and `sun-ejb-jar.xml` deployment descriptor to `false`. For more information, see the *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

Project Open ESB Starter Kit and JBI Support

An Enterprise Service Bus (ESB) is a distributed infrastructure used for enterprise integration. It consists of a set of service containers, which integrate various types of information technology assets. The containers are interconnected with a reliable messaging bus. Service containers adapt information technology assets to a standard services model, based on XML message exchange using standardized message exchange patterns. The ESB provides services for transforming and routing messages, as well as the ability to centrally administer the distributed system.

Project Open ESB Starter Kit implements an ESB runtime that incorporates the JSR 208 (`http://jcp.org/en/jsr/detail?id=208`) specification for Java Business Integration (JBI) and other open standards. Open ESB Starter Kit allows you to integrate web services and enterprise applications as loosely coupled composite applications within a Service-Oriented Architecture (SOA).

Open ESB Starter Kit is available from the Java EE 5 SDK SOA Starter Kit Preview distribution. It is also available as a separate download that can be installed into the Application Server. An enhanced version of Open ESB Starter Kit that allows you to install multiple instances of Open ESB is available from Project Open ESB (`http://open-esb.dev.java.net/`).

The distribution of Open ESB Starter Kit includes a Business Process Execution Language (BPEL) service engine, a Java EE service engine, an HTTP SOAP binding component, and a getting started tutorial. Additional components, tools, and documentation are available for download. Refer to Project Open ESB Starter Kit (`http://java.sun.com/integration/openesb/starterkit.jsp`) for more information on Open ESB and the additional components, tools, and documentation that are available.

When you install Open ESB Starter Kit, the Java EE Service Engine (`https://glassfish.dev.java.net/javaee5/jbi-se/ServiceEngine.html`) is automatically installed on the Application Server. The Java EE Service Engine acts as a bridge between the Application Server and the Open ESB environment for web service providers and web service consumers. The Java EE Service Engine provides better performance than the default SOAP over HTTP socket connection due to in-process communication between components and additional protocols such as JMS.

The Java EE Service Engine is enabled by default. To disable it without uninstalling it, set the `com.sun.enterprise.jbi.se.disable` JVM option to `true` using the `asadmin create-jvm-options` command as follows, then restart the server:

```
asadmin create-jvm-options --user adminuser -Dcom.sun.enterprise.jbi.se.disable=true
```

For more information about the `asadmin create-jvm-options` command, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

To determine whether a web service endpoint is enabled in the JBI environment, you can set a `jbi-enabled` attribute in the Application Server. This attribute is set to `true` (enabled) by default. To disable an endpoint for JBI, set the attribute to `false` using the `asadmin set` command. For example, if an endpoint is bundled as a WAR file named `my-ws.war` with an endpoint named `simple`, use the following command:

```
asadmin set --user adminuser
server.applications.web-module.my-ws.web-service-endpoint.simple.jbi-enabled=false
```

To determine whether requests from a web service consumer are routed through the Java EE Service Engine, you can set a `stub-property` named `jbi-enabled` in the consumer's `sun-web.xml` or `sun-ejb-jar.xml` file. This property is set to `false` (disabled) by default. Here is an example of the `sun-web.xml` file:

```
<sun-web-app>
  <service-ref>
    <service-ref-name>sun-web.serviceref/calculator</service-ref-name>
    <port-info>
      <wsdl-port>
        <namespaceURI>http://example.web.service/Calculator</namespaceURI>
        <localpart>CalculatorPort</localpart>
      </wsdl-port>
      <service-endpoint-interface>service.web.example.calculator.Calculator</service-endpoint-interface>
      <stub-property name="jbi-enabled" value="true"/>
    </port-info>
  </service-ref>
</sun-web-app>
```

For more information about the `sun-web.xml` and `sun-ejb-jar.xml` deployment descriptor files, see the *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

◆ ◆ ◆ CHAPTER 7

Using the Java Persistence API

Sun Java System 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:

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 TopLink persistence provider, the Application Server attempts to automatically detect the database based on the connection. You can specify the `toplink.platform.class.name` 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="toplink.platform.class.name"
                    value="oracle.toplink.essentials.platform.database.DB2Platform"/>
            </properties>
        </persistence-unit>
    </persistence>
```

The following `toplink.platform.class.name` property values are allowed:

```
//Supported platforms
oracle.toplink.essentials.platform.DerbyPlatform
oracle.toplink.essentials.platform.oracle.OraclePlatform
oracle.toplink.essentials.platform.SQLServerPlatform
oracle.toplink.essentials.platform.DB2Platform
oracle.toplink.essentials.platform.SybasePlatform
oracle.toplink.essentials.platform.CloudscapePlatform
oracle.toplink.essentials.platform.MySQL4Platform
oracle.toplink.essentials.platform.PointBasePlatform
oracle.toplink.essentials.platform.PotgreSQLPlatform

//Others available
oracle.toplink.essentials.platform.InformixPlatform
oracle.toplink.essentials.platform.TimesTenPlatform
oracle.toplink.essentials.platform.AttunityPlatform
oracle.toplink.essentials.platform.HSQLPlatform
oracle.toplink.essentials.platform.SQLAnyWherePlatform
oracle.toplink.essentials.platform.DBasePlatform
oracle.toplink.essentials.platform.DB2MainframePlatform
oracle.toplink.essentials.platform.AccessPlatform
```

To use the Java Persistence API outside the EJB container, do not specify the `jta-data-source` and `non-jta-data-source` elements. Instead, specify the `provider` element and any additional properties required by the JDBC driver or the database. If you are using the TopLink persistence provider, specify the `toplink.platform.class.name` property and the `toplink.jdbc.*` properties. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
    <persistence xmlns="http://java.sun.com/xml/ns/persistence">
        <persistence-unit name ="em2">
            <provider>oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider</provider>
```

```
            <transaction-type>RESOURCE_LOCAL<transaction-type>
            <non-jta-data-source>jdbc/MyDB2DB</non-jta-data-source>
            <properties>
                <property name="toplink.platform.class.name"
                    value="oracle.toplink.essentials.platform.database.DB2Platform"/>
                <!-- JDBC connection properties -->
                <property name="toplink.jdbc.driver" value="org.apache.derby.jdbc.ClientDriver"/>
                <property name="toplink.jdbc.url"
value="jdbc:derby://localhost:1527/testdb;retrieveMessagesFromServerOnGetMessage=true;create=true;"/>
                <property name="toplink.jdbc.user" value="APP"/>
                <property name="toplink.jdbc.password" value="APP"/>
            </properties>
        </persistence-unit>
    </persistence>
```

For more information about `toplink` properties, see "Database Properties" on page 105.

For a list of the JDBC drivers currently supported by the Application Server, see the *Sun Java System Application Server Platform Edition 9 Release Notes*. For configurations of supported and other drivers, see "Configurations for Specific JDBC Drivers" in *Sun Java System Application Server Platform Edition 9 Administration Guide*.

Note – 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 Properties

If you are using the TopLink persistence provider, you can specify the following database properties. For schema generation properties, see "Generation Options" on page 108. For query hints, see "Query Hints" on page 110.

TABLE 7–1 Database Properties

Property	Default	Description
`toplink.jdbc.driver`	`org.apache.derby.jdbc.ClientDriver`	Specifies the JDBC driver.
`toplink.jdbc.url`	depends on the JDBC driver and platform	Specifies the connection URL.
`toplink.jdbc.user`	none	Specifies the database user.
`toplink.jdbc.password`	none	Specifies the user's password.

TABLE 7–1 Database Properties *(Continued)*

Property	Default	Description
`toplink.platform.class.name`	`oracle.toplink.essentials.platform.DerbyPlatform`	Specifies the database platform. See "Specifying the Database" on page 103.
`toplink.logging.level`	`INFO`	Specifies the logging level to use. Use `FINE` to see the generated SQL statements.
`toplink.bind-all-parameters`	`true`	If `true`, generated prepared statements have parameter markers for the values to be bound. If `false`, parameter values are bound inline.
`toplink.cache.default-size`	`1000`	Specifies the maximum number of objects that can be stored in each cache. TopLink maintains a cache for each type of entity.
`toplink.max-write-connections`	`10`	Specifies the maximum number of write connections used by a TopLink created connection pool in an application managed entity manager.
`toplink.min-write-connections`	`5`	Specifies the minimum number of write connections used by a TopLink created connection pool in an application managed entity manager.
`toplink.max-read-connections`	`2`	Specifies the maximum number of read connections used by a TopLink created connection pool in an application managed entity manager.
`toplink.min-read-connections`	`2`	Specifies the minimum number of read connections used by a TopLink created connection pool in an application managed entity manager.
`toplink.weaving`	`true`	If `false`, turns off the TopLink weaving feature. Weaving optimizes lazy `@OneToOne` and `@OneToMany` mappings.

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

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.

Supported Data Types

The following table shows mappings of Java types to SQL types when the Java Persistence provider is Oracle TopLink and automatic schema generation is used.

TABLE 7–2 Java Type to SQL Type Mappings

Java Type	Java DB, Derby, CloudScape	Oracle	DB2	Sybase	MS-SQL Server	MySQL Server
`boolean`, `java.lang.Boolean`	`SMALLINT`	`NUMBER(1)`	`SMALLINT`	`BIT`	`BIT`	`TINYINT(1)`
`int`, `java.lang.Integer`	`INTEGER`	`NUMBER(10)`	`INTEGER`	`INTEGER`	`INTEGER`	`INTEGER`
`long`, `java.lang.Long`	`BIGINT`	`NUMBER(19)`	`INTEGER`	`NUMERIC(19)`	`NUMERIC(19)`	`BIGINT`
`float`, `java.lang.Float`	`FLOAT`	`NUMBER(19,4)`	`FLOAT`	`FLOAT(16)`	`FLOAT(16)`	`FLOAT`
`double`, `java.lang.Double`	`FLOAT`	`NUMBER(19,4)`	`FLOAT`	`FLOAT(32)`	`FLOAT(32)`	`DOUBLE`
`short`, `java.lang.Short`	`SMALLINT`	`NUMBER(5)`	`SMALLINT`	`SMALLINT`	`SMALLINT`	`SMALLINT`
`byte`, `java.lang.Byte`	`SMALLINT`	`NUMBER(3)`	`SMALLINT`	`SMALLINT`	`SMALLINT`	`SMALLINT`
`java.lang.Number`	`DECIMAL`	`NUMBER(38)`	`DECIMAL(15)`	`NUMERIC(38)`	`NUMERIC(28)`	`DECIMAL(38)`
`java.math.BigInteger`	`BIGINT`	`NUMBER(38)`	`BIGINT`	`NUMERIC(38)`	`NUMERIC(28)`	`BIGINT`
`java.math.BigDecimal`	`DECIMAL`	`NUMBER(38)`	`DECIMAL(15)`	`NUMERIC(38)`	`NUMERIC(28)`	`DECIMAL(38)`
`java.lang.String`	`VARCHAR(255)`	`VARCHAR(255)`	`VARCHAR(255)`	`VARCHAR(255)`	`VARCHAR(255)`	`VARCHAR(255)`
`char`, `java.lang.Character`	`CHAR(1)`	`CHAR(1)`	`CHAR(1)`	`CHAR(1)`	`CHAR(1)`	`CHAR(1)`

TABLE 7–2 Java Type to SQL Type Mappings *(Continued)*

Java Type	Java DB, Derby, CloudScape	Oracle	DB2	Sybase	MS-SQL Server	MySQL Server
`byte[]`, `java.lang.Byte[]`, `java.sql.Blob`	`BLOB(64000)`	`LONG RAW`	`BLOB(64000)`	`IMAGE`	`IMAGE`	`BLOB(64000)`
`char[]`, `java.lang.Character[]`, `java.sql.Clob`	`CLOB(64000)`	`LONG`	`CLOB(64000)`	`TEXT`	`TEXT`	`TEXT(64000)`
`java.sql.Date`	`DATE`	`DATE`	`DATE`	`DATETIME`	`DATETIME`	`DATE`
`java.sql.Time`	`TIME`	`DATE`	`TIME`	`DATETIME`	`DATETIME`	`TIME`
`java.sql.Timestamp`	`TIMESTAMP`	`DATE`	`TIMESTAMP`	`DATETIME`	`DATETIME`	`DATETIME`

Generation Options

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

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

Note – Before using these options, make sure you have a properly configured database. See "Specifying the Database" on page 103.

The following optional schema generation properties control the automatic creation of database tables at deployment. You can specify them in the `persistence.xml` file.

TABLE 7–3 Schema Generation Properties

Property	Default	Description
`toplink.ddl-generation`	`none`	Specifies whether tables and DDL files are created during deployment, and whether tables are dropped first if they already exist. Allowed values are `create-tables`, `drop-and-create-tables`, and `none`. If you are using persistence outside the EJB container and would like to create the DDL files without creating tables, additionally define a Java system property `INTERACT_WITH_DB` and set its value to `false`.
`toplink.create-ddl-jdbc-file-name`	`createDDL.jdbc`	Specifies the name of the JDBC file that contains the DDL statements required to create the required objects (tables, sequences, and constraints) in the database.

TABLE 7–3 Schema Generation Properties *(Continued)*

Property	Default	Description
`toplink.drop-ddl-jdbc-file-name`	`dropDDL.jdbc`	Specifies the name of the JDBC file that contains the DDL statements required to drop the required objects (tables, sequences, and constraints) from the database.
`toplink.application-location`	`.` for the current working directory	For persistence outside the EJB container, specifies the location where the DDL files are written. For persistence within the EJB container, if this property is set, the value is ignored, and DDL files are written to one of the following locations, for applications and modules, respectively: *domain-dir*`/generated/ejb/j2ee-apps/`*app-name* *domain-dir*`/generated/ejb/j2ee-modules/`*mod-name*

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

TABLE 7–4 The `asadmin deploy` and `asadmin deploydir` Generation Options

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

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

TABLE 7–5 The `asadmin undeploy` Generation Options

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

For more information about the `asadmin deploy`, `asadmin deploydir`, and `asadmin undeploy` commands, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

If one or more entities in the persistence unit are mapped to existing database tables and you use any of the `asadmin deploy` or `asadmin deploydir` options, the `asadmin` deployment options take precedence.

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

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

Query Hints

Query hints are additional, implementation-specific configuration settings. You can use Oracle TopLink 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("toplink.cache-usage", 3).
     getSingleResult();
```

Allowed values for each type of hint are summarized in the following table.

TABLE 7–6 Query Hints

Hint Name	Default	Description
`toplink.reference-class`	none	Specifies the reference class for the query.

TABLE 7–6 Query Hints *(Continued)*

Hint Name	Default	Description
`toplink.cache-usage`	`5`	Specifies cache usage. By default, only single result queries for the primary key check the cache before accessing the database. Any query can be configured to query against the cache completely, to query by key, or to ignore the cache check. Allowed values are as follows: ■ `-1` uses TopLink descriptor settings. ■ `0` doesn't check the cache, maintains the cache, and checks the database. ■ `1` checks the cache if the query is exactly and only on the object's primary key. ■ `2` checks the cache if the query contains the primary key and possible other values. ■ `3` checks the whole cache for any object matching the query; if none is found the database is accessed. ■ `4` checks the whole cache for any object matching the query; if none is found, `null` or an empty collection is returned. ■ `5` checks the results against the changes within the persistence context; objects no longer matching or deleted are removed, and matching new objects are added.
`toplink.refresh`	`0`	Specifies that attributes of the objects resulting from the query are refreshed if set to `1` (true). If cascading is used, the private attributes of the objects are also refreshed. The other allowed value is `0` (false).
`toplink.pessimistic-lock`	`0`	Specifies the pessimistic locking mode of the query. Allowed values are as follows: ■ `0` specifies `NO_LOCK` ■ `1` specifies `LOCK` ■ `2` specifies `LOCK_NOWAIT`
`toplink.expression`	none	Specifies an alternative way to define the selection criteria of the query.
`toplink.call`	none	Specifies an alternative to using native SQL queries with a new `SQLCall("SELECT...")`.
`toplink.cascade`	`2`	Specifies the cascading of the query. You must specify TopLink mappings in conjunction with Java Persistence API mappings for cascading to work. Allowed values are as follows: ■ `1` specifies `NoCascading` ■ `2` specifies `CascadePrivateParts` ■ `3` specifies `CascadeAllParts` ■ `4` specifies `CascadeDependentParts` ■ `5` specifies `CascadeAggregateDelete` ■ `6` specifies `CascadeByMapping`

Database Restrictions and Optimizations

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

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 8

Developing Web Applications

This chapter describes how web applications are supported in the Sun Java System 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`).

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 *Sun Java System Application Server Platform Edition 9 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 8–1 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` or `sun-application.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.

To change these defaults using the Admin Console, select the Application Server component, and the Logging tab, then check or uncheck this box:

- Write to System Log - If checked, `System.out` output is sent to the server log. If 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.

For information about JSP caching, see "JSP Caching" on page 122.

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 120.
- All non-ASCII request parameter values specified in cache key elements must be URL encoded. The caching subsystem attempts to match the raw parameter values in the request query string.
- Since newly updated classes impact what gets cached, the web container clears the cache during dynamic deployment or reloading of classes.
- The following `HttpServletRequest` request attributes are exposed.
  - `com.sun.appserv.web.cachedServletName`, the cached servlet target
  - `com.sun.appserv.web.cachedURLPattern`, the URL pattern being cached
- Results produced by resources that are the target of a `RequestDispatcher.include()` or `RequestDispatcher.forward()` call are cached if caching has been enabled for those resources. For details, see "cache-mapping" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide* and "dispatcher" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*. These are elements in the `sun-web.xml` file.

Default Cache Configuration

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

- The default cache timeout is 30 seconds.
- Only the HTTP GET method is eligible for caching.
- HTTP requests with cookies or sessions automatically disable caching.
- No special consideration is given to `Pragma:`, `Cache-control:`, or `Vary:` headers.
- The default key consists of the Servlet Path (minus `pathInfo` and the query string).
- A "least recently used" list is maintained to evict cache entries if the maximum cache size is exceeded.
- Key generation concatenates the servlet path with key field values, if any are specified.

- Results produced by resources that are the target of a `RequestDispatcher.include()` or `RequestDispatcher.forward()` call are never cached.

Caching Example

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

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

For more information about the `sun-web.xml` caching settings, see "cache" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

The `CacheKeyGenerator` Interface

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

The name of the context attribute is configurable as the `value` of the `cacheKeyGeneratorAttrName` property in the `default-helper` element of the `sun-web.xml` deployment descriptor. For more information, see "default-helper" in *Sun Java System Application Server Platform Edition 9 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" on page 120
- "Request Handling" on page 120

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`.

This section describes how to use JavaServer Pages (JSP files) as page templates in an Application Server web application. This section contains the following topics:

- "JSP Tag Libraries and Standard Portable Tags" on page 121
- "JSP Caching" on page 122
- "Options for Compiling JSP Files" on page 125

JSP Tag Libraries and Standard Portable Tags

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 *install-dir*`/lib`) to use JavaServer™ Faces technology or JSTL, respectively. These tag libraries are automatically available to all web applications.

However, the *install-dir*`/lib/appserv-tags.jar` tag library for JSP caching is not automatically available to web applications. See "JSP Caching" on page 122, next.

JSP Caching

JSP caching lets you cache tag invocation results within the Java engine. Each can be cached using different cache criteria. For example, suppose you have invocations to view stock quotes, weather information, and so on. The stock quote result can be cached for 10 minutes, the weather report result for 30 minutes, and so on. JSP caching is described in the following topics:

For more information about response caching as it pertains to servlets, see "Caching Servlet Results" on page 117.

The `appserv-tags.jar` File

JSP caching is implemented by a tag library packaged into the *install-dir*`/lib/appserv-tags.jar` file, which you can copy into the `WEB-INF/lib` directory of your web application. The `appserv-tags.tld` tag library descriptor file is in the `META-INF` directory of this JAR file.

Note – Web applications that use this tag library without bundling it are not portable.

To allow all web applications to share this tag library, change the following elements in the `domain.xml` file. Change this:

```
<jvm-options>
-Dcom.sun.enterprise.taglibs=appserv-jstl.jar,jsf-impl.jar
</jvm-options>
```

to this:

```
<jvm-options>
-Dcom.sun.enterprise.taglibs=appserv-jstl.jar,jsf-impl.jar,appserv-tags.jar
</jvm-options>
```

and this:

```
<jvm-options>
-Dcom.sun.enterprise.taglisteners=jsf-impl.jar
</jvm-options>
```

to this:

```
<jvm-options>
-Dcom.sun.enterprise.taglisteners=jsf-impl.jar,appserv-tags.jar
</jvm-options>
```

For more information about the `domain.xml` file, see the *Sun Java System Application Server Platform Edition 9 Administration Reference*.

Refer to these tags in JSP files as follows:

```
<%@ taglib prefix="prefix" uri="Sun ONE Application Server Tags" %>
```

Subsequently, the cache tags are available as <*prefix*:`cache`> and <*prefix*:`flush`>. For example, if your *prefix* is `mypfx`, the cache tags are available as <`mypfx:cache`> and <`mypfx:flush`>.

Caching Scope

JSP caching is available in three different scopes: `request`, `session`, and `application`. The default is `application`. To use a cache in `request` scope, a web application must specify the `com.sun.appserv.web.taglibs.cache.CacheRequestListener` in its `web.xml` deployment descriptor, as follows:

```
<listener>
   <listener-class>
      com.sun.appserv.web.taglibs.cache.CacheRequestListener
   </listener-class>
</listener>
```

Likewise, for a web application to utilize a cache in `session` scope, it must specify the `com.sun.appserv.web.taglibs.cache.CacheSessionListener` in its `web.xml` deployment descriptor, as follows:

```
<listener>
   <listener-class>
      com.sun.appserv.web.taglibs.cache.CacheSessionListener
   </listener-class>
</listener>
```

To utilize a cache in `application` scope, a web application need not specify any listener. The `com.sun.appserv.web.taglibs.cache.CacheContextListener` is already specified in the `appserv-tags.tld` file.

The `cache` Tag

The cache tag caches the body between the beginning and ending tags according to the attributes specified. The first time the tag is encountered, the body content is executed and cached. Each subsequent time it is run, the cached content is checked to see if it needs to be refreshed and if so, it is executed again, and the cached data is refreshed. Otherwise, the cached data is served.

Attributes of `cache`

The following table describes attributes for the `cache` tag.

TABLE 8–2 The cache Attributes

Attribute	Default	Description
key	*ServletPath_Suffix*	(optional) The name used by the container to access the cached entry. The cache key is suffixed to the servlet path to generate a key to access the cached entry. If no key is specified, a number is generated according to the position of the tag in the page.
timeout	60s	(optional) The time in seconds after which the body of the tag is executed and the cache is refreshed. By default, this value is interpreted in seconds. To specify a different unit of time, add a suffix to the timeout value as follows: s for seconds, m for minutes, h for hours, d for days. For example, 2h specifies two hours.
nocache	false	(optional) If set to true, the body content is executed and served as if there were no cache tag. This offers a way to programmatically decide whether the cached response is sent or whether the body has to be executed, though the response is not cached.
refresh	false	(optional) If set to true, the body content is executed and the response is cached again. This lets you programmatically refresh the cache immediately regardless of the timeout setting.
scope	application	(optional) The scope of the cache. Can be request, session, or application. See "Caching Scope" on page 123.

Example of cache

The following example represents a cached JSP file:

```
<%@ taglib prefix="mypfx" uri="Sun ONE Application Server Tags" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<mypfx:cache                key="${sessionScope.loginId}"
            nocache="${param.nocache}"
            refresh="${param.refresh}"
            timeout="10m">
<c:choose>
    <c:when test="${param.page == 'frontPage'}">
        <%-- get headlines from database --%>
    </c:when>
    <c:otherwise>
        ...
    </c:otherwise>
</c:choose>
</mypfx:cache>
<mypfx:cache timeout="1h">
<h2> Local News </h2>
    <%-- get the headline news and cache them --%>
</mypfx:cache>
```

The `flush` Tag

Forces the cache to be flushed. If a `key` is specified, only the entry with that key is flushed. If no key is specified, the entire cache is flushed.

Attributes of `flush`

The following table describes attributes for the `flush` tag.

TABLE 8–3 The `flush` Attributes

Attribute	Default	Description
`key`	*ServletPath_Suffix*	(optional) The name used by the container to access the cached entry. The cache key is suffixed to the servlet path to generate a key to access the cached entry. If no key is specified, a number is generated according to the position of the tag in the page.
`scope`	`application`	(optional) The scope of the cache. Can be `request`, `session`, or `application`. See "Caching Scope" on page 123.

Examples of `flush`

To flush the entry with `key="foobar"`:

```
<mypfx:flush key="foobar"/>
```

To flush the entire cache:

```
<c:if test="${empty sessionScope.clearCache}">
   <mypfx:flush />
</c:if>
```

Options for Compiling JSP Files

Application Server provides the following ways of compiling JSP 2.1 compliant source files into servlets:

- JSP files are automatically compiled at runtime.
- The `asadmin deploy` command has a `precompilejsp` option. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.
- The `sun-appserv-jspc` Ant task allows you to precompile JSP files; see "The `sun-appserv-jspc` Task" on page 56.
- The `jspc` command line tool allows you to precompile JSP files at the command line. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

Creating and Managing HTTP Sessions

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

This chapter contains the following sections:

Configuring Sessions

This section covers the following topics:

Sessions, Cookies, and URL Rewriting

To configure whether and how sessions use cookies and URL rewriting, edit the `session-properties` and `cookie-properties` elements in the `sun-web.xml` file for an individual web application. For more about the properties you can configure, see "session-properties" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide* and "cookie-properties" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

For information about configuring default session properties for the entire web container, see Chapter 7, "Java EE Containers," in *Sun Java System Application Server Platform Edition 9 Administration 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 127, the default
- "The file Persistence Type" on page 127, which uses a file to store session data

---

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

For more information, see "session-manager" in *Sun Java System Application Server Platform Edition 9 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 *Sun Java System Application Server Platform Edition 9 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 *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

For more information about the `sun-web.xml` file, see "The sun-web.xml File" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

The `file` Persistence Type

This persistence type provides session persistence to the local file system, and allows a single server domain to recover the session state after a failure and restart. The session state is persisted in the

background, and the rate at which this occurs is configurable. The store also provides passivation and activation of the session state to help control the amount of memory used. This option is not supported in a production environment. However, it is useful for a development system with a single server instance.

Note – Make sure the `delete` option is set in the `server.policy` file, or expired file-based sessions might not be deleted properly. For more information about `server.policy`, see "The `server.policy` File" on page 80.

To specify the `file` persistence type for the entire web container, use the `configure-ha-persistence` command. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

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

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

The `file` persistence type supports all the manager properties listed under "manager-properties" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide* except `sessionFilename`, and supports the `directory` store property listed under "store-properties" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

For more information about the `sun-web.xml` file, see "The sun-web.xml File" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

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. Select the Application Server component, the Advanced tab, and the Domain Attributes tab, then type a value in the Locale field. For details, click the Help button in the 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 *Sun Java System Application Server Platform Edition 9 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. Open the HTTP Service component under the relevant configuration in the Admin Console, select HTTP Listeners, and select or create an HTTP listener. Then select from the Default Virtual Server drop-down list. For details, click the Help button in the Admin Console.

Web applications and Java EE applications containing web components can be assigned to virtual servers.

▼ To Assign Virtual Servers

1 Deploy the application or web module and assign the desired virtual server to it.

For more information, see *Sun Java System Application Server Platform Edition 9 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 130.

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 130. 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 "The sun-web.xml File" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

For general information about class loaders, see Chapter 2.

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.

▼ To Use the `default-web.xml` File

1. **Place the JAR file for the filter, security constraint, or other feature in the *domain-dir*`/lib` directory.**
2. **Edit the *domain-dir*`/config/default-web.xml` file to refer to the JAR file.**
3. **Restart the server.**

Configuring Logging and Monitoring in the Web Container

For information about configuring logging and monitoring in the web container using the Admin Console, click the Help button in the Admin Console from the Logging or Monitor tab on the Application Server page.

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.

◆ ◆ ◆ CHAPTER 9

Using Enterprise JavaBeans Technology

This chapter describes how Enterprise JavaBeans™ (EJB) technology is supported in the Sun Java System 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`).

Summary of EJB 3.0 Changes

The Application Server supports and is compliant with the Sun Microsystems Enterprise JavaBeans (EJB) architecture as defined by the Enterprise JavaBeans Specification, v3.0, also known as JSR 220 (`http://jcp.org/en/jsr/detail?id=220`).

Note – The Application Server is backward compatible with 1.1, 2.0, and 2.1 enterprise beans. However, to take advantage of version 3.0 features, you should develop new beans as 3.0 enterprise beans.

The main changes in the Enterprise JavaBeans Specification, v3.0 that impact enterprise beans in the Application Server environment are as follows:

- Definition of the Java language metadata *annotations* that can be used to annotate EJB applications. These metadata annotations are targeted at simplifying the developer's task, at reducing the number of program classes and interfaces the developer is required to implement, at encapsulation of environmental dependencies and JNDI access, and at eliminating the need for the developer to provide an EJB deployment descriptor.

- Elimination of the requirement for home or EJB component interfaces for session beans. The required business interface for a session bean can be a plain Java interface rather than an `EJBObject`, `EJBLocalObject`, or `java.rmi.Remote` interface.
- Elimination of all required interfaces for persistent entities. Specification of Java language metadata annotations and XML deployment descriptor elements for the object/relational mapping of persistent entities. For details about Java Persistence in the Application Server, see Chapter 7.

  Container-managed persistence (CMP) is still supported for EJB 2.1 beans, for backward compatibility. For details, see Chapter 10.

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.

Read-Only Beans

Another feature that the Application Server provides is the *read-only bean*, an EJB 2.1 entity bean that is never modified by an EJB client. Read-only beans avoid database updates completely.

Note – Read-only beans are specific to the Application Server and are not part of the Enterprise JavaBeans Specification, v2.1. Use of this feature for an EJB 2.1 bean results in a non-portable application.

To make an EJB 3.0 entity bean read-only, use `@Column` annotations to mark its columns `insertable=false` and `updatable=false`.

A read-only bean can be used to cache a database entry that is frequently accessed but rarely updated (externally by other beans). When the data that is cached by a read-only bean is updated by another bean, the read-only bean can be notified to refresh its cached data.

The Application Server provides a number of ways by which a read-only bean's state can be refreshed. By setting the `refresh-period-in-seconds` element in the `sun-ejb-jar.xml` file and the `trans-attribute` element (or `@TransactionAttribute` annotation) in the `ejb-jar.xml` file, it is easy to configure a read-only bean that is one of the following:

- Always refreshed
- Periodically refreshed
- Never refreshed
- Programmatically refreshed

Read-only beans are best suited for situations where the underlying data never changes, or changes infrequently. For further information and usage guidelines, see "Using Read-Only Beans" on page 141.

The `pass-by-reference` Element

The `pass-by-reference` element in the `sun-ejb-jar.xml` file allows you to specify the parameter passing semantics for colocated remote EJB invocations. This is an opportunity to improve performance. However, use of this feature results in non-portable applications. See "pass-by-reference" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

Pooling and Caching

The EJB container of the Application Server pools anonymous instances (message-driven beans, stateless session beans, and entity beans) to reduce the overhead of creating and destroying objects. The EJB container maintains the free pool for each bean that is deployed. Bean instances in the free pool have no identity (that is, no primary key associated) and are used to serve method calls. The free beans are also used to serve all methods for stateless session beans.

Bean instances in the free pool transition from a Pooled state to a Cached state after `ejbCreate` and the business methods run. The size and behavior of each pool is controlled using pool-related properties in the EJB container or the `sun-ejb-jar.xml` file.

In addition, the Application Server supports a number of tunable parameters that can control the number of "stateful" instances (stateful session beans and entity beans) cached as well as the duration they are cached. Multiple bean instances that refer to the same database row in a table can be cached. The EJB container maintains a cache for each bean that is deployed.

To achieve scalability, the container selectively evicts some bean instances from the cache, usually when cache overflows. These evicted bean instances return to the free bean pool. The size and behavior of each cache can be controlled using the cache-related properties in the EJB container or the `sun-ejb-jar.xml` file.

Pooling and caching parameters for the `sun-ejb-jar.xml` file are described in "bean-cache" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

Pooling Parameters

One of the most important parameters of Application Server pooling is `steady-pool-size`. When `steady-pool-size` is set to greater than 0, the container not only pre-populates the bean pool with the specified number of beans, but also attempts to ensure that this number of beans is always available in the free pool. This ensures that there are enough beans in the ready to serve state to process user requests.

This parameter does not necessarily guarantee that no more than `steady-pool-size` instances exist at a given time. It only governs the number of instances that are pooled over a long period of time. For example, suppose an idle stateless session container has a fully-populated pool with a `steady-pool-size` of `10`. If 20 concurrent requests arrive for the EJB component, the container creates 10 additional instances to satisfy the burst of requests. The advantage of this is that it prevents the container from blocking any of the incoming requests. However, if the activity dies down to 10 or fewer concurrent requests, the additional 10 instances are discarded.

Another parameter, `pool-idle-timeout-in-seconds`, allows the administrator to specify the amount of time a bean instance can be idle in the pool. When `pool-idle-timeout-in-seconds` is set to greater than 0, the container removes or destroys any bean instance that is idle for this specified duration.

Caching Parameters

Application Server provides a way that completely avoids caching of entity beans, using commit option C. Commit option C is particularly useful if beans are accessed in large number but very rarely reused. For additional information, refer to "Commit Options" on page 149.

The Application Server caches can be either bounded or unbounded. *Bounded caches* have limits on the number of beans that they can hold beyond which beans are passivated. For stateful session beans, there are three ways (LRU, NRU and FIFO) of picking victim beans when cache overflow occurs. Caches can also passivate beans that are idle (not accessed for a specified duration).

Bean-Level Container-Managed Transaction Timeouts

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

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

Priority Based Scheduling of Remote Bean Invocations

You can create multiple thread pools, each having its own work queues. An optional element in the `sun-ejb-jar.xml` file, `use-thread-pool-id`, specifies the thread pool that processes the requests for the bean. The bean must have a remote interface, or `use-thread-pool-id` is ignored. You can create different thread pools and specify the appropriate thread pool ID for a bean that requires a quick response time. If there is no such thread pool configured or if the element is absent, the default thread pool is used.

Immediate Flushing

Normally, all entity bean updates within a transaction are batched and executed at the end of the transaction. The only exception is the database flush that precedes execution of a finder or select query.

Since a transaction often spans many method calls, you might want to find out if the updates made by a method succeeded or failed immediately after method execution. To force a flush at the end of a method's execution, use the `flush-at-end-of-method` element in the `sun-ejb-jar.xml` file. Only non-finder methods in an entity bean can be flush-enabled. (For an EJB 2.1 bean, these methods must be in the Local, Local Home, Remote, or Remote Home interface.) See "flush-at-end-of-method" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

Upon completion of the method, the EJB container updates the database. Any exception thrown by the underlying data store is wrapped as follows:

- If the method that triggered the flush is a `create` method, the exception is wrapped with `CreateException`.
- If the method that triggered the flush is a `remove` method, the exception is wrapped with `RemoveException`.
- For all other methods, the exception is wrapped with `EJBException`.

All normal end-of-transaction database synchronization steps occur regardless of whether the database has been flushed during the transaction.

EJB Timer Service

The EJB Timer Service uses a database to store persistent information about EJB timers. By default, the EJB Timer Service in Application Server is preconfigured to use an embedded version of the Java DB database. 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 *Sun Java System Application Server Platform Edition 9 Release Notes*. For configurations of supported and other drivers, see "Configurations for Specific JDBC Drivers" in *Sun Java System Application Server Platform Edition 9 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 *install-dir*`/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`.

For information about configuring EJB Timer Service settings, see Chapter 7, "Java EE Containers," in *Sun Java System Application Server Platform Edition 9 Administration Guide*. For information about the `asadmin list-timers` command, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

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 139
- "Session Bean Restrictions and Optimizations" on page 140

Extensive information on session beans is contained in Chapters 3 and 4 of the Enterprise JavaBeans Specification, v3.0, EJB Core Contracts and Requirements.

About the Session Bean Containers

Like an entity bean, a session bean can access a database through Java Database Connectivity (JDBC) calls. A session bean can also provide transaction settings. These transaction settings and JDBC calls are referenced by the session bean's container, allowing it to participate in transactions managed by the container.

A container managing stateless session beans has a different charter from a container managing stateful session beans. This section addresses the following topics:

- "Stateless Container" on page 139
- "Stateful Container" on page 139

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 *Sun Java System Application Server Platform Edition 9 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 *Sun Java System Application Server Platform Edition 9 Reference Manual*.

Stateful Container

The *stateful container* manages the stateful session beans, which, by definition, carry the client-specific state. There is a one-to-one relationship between the client and the stateful session beans. At creation, each stateful session bean (SFSB) is given a unique session ID that is used to access the session bean so that an instance of a stateful session bean is accessed by a single client only.

Stateful session beans are managed using cache. The size and behavior of stateful session beans cache are controlled by specifying the following `sun-ejb-jar.xml` parameters:

- `max-cache-size`
- `resize-quantity`
- `cache-idle-timeout-in-seconds`
- `removal-timeout-in-seconds`
- `victim-selection-policy`

The `max-cache-size` element specifies the maximum number of session beans that are held in cache. If the cache overflows (when the number of beans exceeds `max-cache-size`), the container then passivates some beans or writes out the serialized state of the bean into a file. The directory in which the file is created is obtained from the EJB container using the configuration APIs.

For more information about `sun-ejb-jar.xml`, see "The sun-ejb-jar.xml File" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

The passivated beans are stored on the file system. The Session Store Location setting in the EJB container allows the administrator to specify the directory where passivated beans are stored. By default, passivated stateful session beans are stored in application-specific subdirectories created under *domain-dir*`/session-store`.

Note – Make sure the `delete` option is set in the `server.policy` file, or expired file-based sessions might not be deleted properly. For more information about `server.policy`, see "The `server.policy` File" on page 80.

Session Bean Restrictions and Optimizations

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

Optimizing Session Bean Performance

For stateful session beans, colocating the stateful beans with their clients so that the client and bean are executing in the same process address space improves performance.

Restricting Transactions

The following restrictions on transactions are enforced by the container and must be observed as session beans are developed:

- A session bean can participate in, at most, a single transaction at a time.
- If a session bean is participating in a transaction, a client cannot invoke a method on the bean such that the `trans-attribute` element (or `@TransactionAttribute` annotation) in the `ejb-jar.xml` file would cause the container to execute the method in a different or unspecified transaction context or an exception is thrown.
- If a session bean instance is participating in a transaction, a client cannot invoke the `remove` method on the session object's home or business interface object, or an exception is thrown.

Using Read-Only Beans

A *read-only bean* is an EJB 2.1 entity bean that is never modified by an EJB client. The data that a read-only bean represents can be updated externally by other enterprise beans, or by other means, such as direct database updates.

Note – Read-only beans are specific to the Application Server and are not part of the Enterprise JavaBeans Specification, v2.1. Use of this feature for an EJB 2.1 bean results in a non-portable application.

To make an EJB 3.0 entity bean read-only, use `@Column` annotations to mark its columns `insertable=false` and `updatable=false`.

Read-only beans are best suited for situations where the underlying data never changes, or changes infrequently. The following topics are addressed in this section:

Read-Only Bean Characteristics and Life Cycle

Read-only beans are best suited for situations where the underlying data never changes, or changes infrequently. For example, a read-only bean can be used to represent a stock quote for a particular company, which is updated externally. In such a case, using a regular entity bean might incur the burden of calling `ejbStore`, which can be avoided by using a read-only bean.

Read-only beans have the following characteristics:

- Only entity beans can be read-only beans.
- Either bean-managed persistence (BMP) or container-managed persistence (CMP) is allowed. If CMP is used, do not create the database schema during deployment. Instead, work with your database administrator to populate the data into the tables. See Chapter 10.
- Only container-managed transactions are allowed; read-only beans cannot start their own transactions.
- Read-only beans don't update any bean state.
- `ejbStore` is never called by the container.
- `ejbLoad` is called only when a transactional method is called or when the bean is initially created (in the cache), or at regular intervals controlled by the bean's `refresh-period-in-seconds` element in the `sun-ejb-jar.xml` file.
- The home interface can have any number of find methods. The return type of the find methods must be the primary key for the same bean type (or a collection of primary keys).

- If the data that the bean represents can change, then `refresh-period-in-seconds` must be set to refresh the beans at regular intervals. `ejbLoad` is called at this regular interval.

A read-only bean comes into existence using the appropriate find methods.

Read-only beans are cached and have the same cache properties as entity beans. When a read-only bean is selected as a victim to make room in the cache, `ejbPassivate` is called and the bean is returned to the free pool. When in the free pool, the bean has no identity and is used only to serve any finder requests.

Read-only beans are bound to the naming service like regular read-write entity beans, and clients can look up read-only beans the same way read-write entity beans are looked up.

Read-Only Bean Good Practices

For best results, follow these guidelines when developing read-only beans:

- Avoid having any `create` or `remove` methods in the home interface.
- Use any of the valid EJB 3.0 transaction attributes for the `trans-attribute` element (or `@TransactionAttribute` annotation).

  The reason for having `TX_SUPPORTED` is to allow reading uncommitted data in the same transaction. Also, the transaction attributes can be used to force `ejbLoad`.

Refreshing Read-Only Beans

There are several ways of refreshing read-only beans as addressed in the following sections:

- "Invoking a Transactional Method" on page 142
- "Refreshing Periodically" on page 142
- "Refreshing Programmatically" on page 143

Invoking a Transactional Method

Invoking any transactional method invokes `ejbLoad`.

Refreshing Periodically

Use the `refresh-period-in-seconds` element in the `sun-ejb-jar.xml` file to refresh a read-only bean periodically.

- If the value specified in `refresh-period-in-seconds` is zero or not specified, which is the default, the bean is never refreshed (unless a transactional method is accessed).
- If the value is greater than zero, the bean is refreshed at the rate specified.

Note – This is the only way to refresh the bean state if the data can be modified external to the Application Server.

Refreshing Programmatically

Typically, beans that update any data that is cached by read-only beans need to notify the read-only beans to refresh their state. Use `ReadOnlyBeanNotifier` to force the refresh of read-only beans.

To do this, invoke the following methods on the `ReadOnlyBeanNotifier` bean:

```
public interface ReadOnlyBeanNotifier extends java.rmi.Remote {
   refresh(Object PrimaryKey) throws RemoteException;
 }
```

The implementation of the `ReadOnlyBeanNotifier` interface is provided by the container. The bean looks up `ReadOnlyBeanNotifier` using a fragment of code such as the following example:

```
com.sun.appserv.ejb.ReadOnlyBeanHelper helper =
  new com.sun.appserv.ejb.ReadOnlyBeanHelper();
com.sun.appserv.ejb.ReadOnlyBeanNotifier notifier =
  helper.getReadOnlyBeanNotifier("java:comp/env/ejb/ReadOnlyCustomer");
notifier.refresh(PrimaryKey);
```

For a local read-only bean notifier, the lookup has this modification:

```
helper.getReadOnlyBeanLocalNotifier("java:comp/env/ejb/LocalReadOnlyCustomer");
```

Beans that update any data that is cached by read-only beans need to call the `refresh` methods. The next (non-transactional) call to the read-only bean invokes `ejbLoad`.

For Javadoc tool pages relevant to read-only beans, go to `http://glassfish.dev.java.net/nonav/javaee5/api/index.html` and click on the `com.sun.appserv.ejb` package.

Deploying Read-Only Beans

Read-only beans are deployed in the same manner as other entity beans. However, in the entry for the bean in the `sun-ejb-jar.xml` file, the `is-read-only-bean` element must be set to true. That is:

```
<is-read-only-bean>true</is-read-only-bean>
```

Also, the `refresh-period-in-seconds` element in the `sun-ejb-jar.xml` file can be set to some value that specifies the rate at which the bean is refreshed. If this element is missing, no refresh occurs.

All requests in the same transaction context are routed to the same read-only bean instance. Set the `allow-concurrent-access` element to either `true` (to allow concurrent accesses) or `false` (to serialize concurrent access to the same read-only bean). The default is `false`.

For further information on these elements, refer to "The sun-ejb-jar.xml File" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

Using Message-Driven Beans

This section describes message-driven beans and explains the requirements for creating them in the Application Server environment. This section contains the following topics:

Message-Driven Bean Configuration

This section addresses the following configuration topics:

Connection Factory and Destination

A message-driven bean is a client to a Connector inbound resource adapter. The message-driven bean container uses the JMS service integrated into the Application Server for message-driven beans that are JMS clients. JMS clients use JMS Connection Factory- and Destination-administered objects. A JMS Connection Factory administered object is a resource manager Connection Factory object that is used to create connections to the JMS provider.

The `mdb-connection-factory` element in the `sun-ejb-jar.xml` file for a message-driven bean specifies the connection factory that creates the container connection to the JMS provider.

The `jndi-name` element of the `ejb` element in the `sun-ejb-jar.xml` file specifies the JNDI name of the administered object for the JMS `Queue` or `Topic` destination that is associated with the message-driven bean.

Message-Driven Bean Pool

The container manages a pool of message-driven beans for the concurrent processing of a stream of messages. The `sun-ejb-jar.xml` file contains the elements that define the pool (that is, the `bean-pool` element):

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

For more information about `sun-ejb-jar.xml`, see "The sun-ejb-jar.xml File" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

Domain-Level Settings

You can control the following domain-level message-driven bean settings in the EJB container:

- Initial and Minimum Pool Size - Specifies the initial and minimum number of beans maintained in the pool. The default is `0`.
- Maximum Pool Size - Specifies the maximum number of beans that can be created to satisfy client requests. The default is 32.
- Pool Resize Quantity - Specifies the number of beans to be created if a request arrives when the pool is empty (subject to the Initial and Minimum Pool Size), or the number of beans to remove if idle for more than the Idle Timeout. The default is `8`.
- Idle Timeout - Specifies the maximum time in seconds that a bean can remain idle in the pool. After this amount of time, the bean is destroyed. The default is `600` (10 minutes). A value of `0` means a bean can remain idle indefinitely.

For information on monitoring message-driven beans, select the Application Server component and the Monitor tab in the Admin Console, then click the Help button.

Note – Running monitoring when it is not needed might impact performance, so you might choose to turn monitoring off when it is not in use. For details, see Chapter 16, "Monitoring Components and Services," in *Sun Java System Application Server Platform Edition 9 Administration Guide*.

Message-Driven Bean Restrictions and Optimizations

This section discusses the following restrictions and performance optimizations that pertain to developing message-driven beans:

Pool Tuning and Monitoring

The message-driven bean pool is also a pool of threads, with each message-driven bean instance in the pool associating with a server session, and each server session associating with a thread. Therefore, a large pool size also means a high number of threads, which impacts performance and server resources.

When configuring message-driven bean pool properties, make sure to consider factors such as message arrival rate and pattern, `onMessage` method processing time, overall server resources (threads, memory, and so on), and any concurrency requirements and limitations from other resources that the message-driven bean accesses.

When tuning performance and resource usage, make sure to consider potential JMS provider properties for the connection factory used by the container (the `mdb-connection-factory` element

in the `sun-ejb-jar.xml` file). For example, you can tune the Sun Java System Message Queue flow control related properties for connection factory in situations where the message incoming rate is much higher than `max-pool-size` can handle.

Refer to Chapter 16, "Monitoring Components and Services," in *Sun Java System Application Server Platform Edition 9 Administration Guide* for information on how to get message-driven bean pool statistics.

The `onMessage` Runtime Exception

Message-driven beans, like other well-behaved MessageListeners, should not, in general, throw runtime exceptions. If a message-driven bean's `onMessage` method encounters a system-level exception or error that does not allow the method to successfully complete, the Enterprise JavaBeans Specification, v3.0 provides the following guidelines:

- If the bean method encounters a runtime exception or error, it should simply propagate the error from the bean method to the container.
- If the bean method performs an operation that results in a checked exception that the bean method cannot recover, the bean method should throw the `javax.ejb.EJBException` that wraps the original exception.
- Any other unexpected error conditions should be reported using `javax.ejb.EJBException` (`javax.ejb.EJBException` is a subclass of `java.lang.RuntimeException`).

Under container-managed transaction demarcation, upon receiving a runtime exception from a message-driven bean's `onMessage` method, the container rolls back the container-started transaction and the message is redelivered. This is because the message delivery itself is part of the container-started transaction. By default, the Application Server container closes the container's connection to the JMS provider when the first runtime exception is received from a message-driven bean instance's `onMessage` method. This avoids potential message redelivery looping and protects server resources if the message-driven bean's `onMessage` method continues misbehaving. To change this default container behavior, use the `cmt-max-runtime-exceptions` property of the `mdb-container` element in the `domain.xml` file.

The `cmt-max-runtime-exceptions` property specifies the maximum number of runtime exceptions allowed from a message-driven bean's `onMessage` method before the container starts to close the container's connection to the message source. By default this value is 1; -1 disables this container protection.

A message-driven bean's `onMessage` method can use the `javax.jms.Message` `getJMSRedelivered` method to check whether a received message is a redelivered message.

Note – The `cmt-max-runtime-exceptions` property might be deprecated in the future.

Sample Message-Driven Bean XML Files

This section includes the following sample files:

For general information on the `sun-ejb-jar.xml` file, see "The sun-ejb-jar.xml File" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

Sample `ejb-jar.xml` File

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
2.0//EN' 'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>
<ejb-jar>
<enterprise-beans>
    <message-driven>
        <ejb-name>MessageBean</ejb-name>
        <ejb-class>samples.mdb.ejb.MessageBean</ejb-class>
        <transaction-type>Container</transaction-type>
        <message-driven-destination>
            <destination-type>javax.jms.Queue</destination-type>
        </message-driven-destination>
        <resource-ref>
            <res-ref-name>jms/QueueConnectionFactory</res-ref-name>
            <res-type>javax.jms.QueueConnectionFactory</res-type>
            <res-auth>Container</res-auth>
        </resource-ref>
    </message-driven>
</enterprise-beans>
    <assembly-descriptor>
        <container-transaction>
            <method>
                <ejb-name>MessageBean</ejb-name>
                <method-intf>Bean</method-intf>
                <method-name>onMessage</method-name>
                <method-params>
                    <method-param>javax.jms.Message</method-param>
                </method-params>
            </method>
        <trans-attribute>NotSupported</trans-attribute>
    </container-transaction>
</assembly-descriptor
</ejb-jar>
```

Sample `sun-ejb-jar.xml` File

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD
Application Server 9.0 EJB 3.0//EN'
```

```
'http://www.sun.com/software/appserver/dtds/sun-ejb-jar_3_0-0.dtd'>
<sun-ejb-jar>
<enterprise-beans>
    <ejb>
        <ejb-name>MessageBean</ejb-name>
        <jndi-name>jms/sample/Queue</jndi-name>
        <resource-ref>
            <res-ref-name>jms/QueueConnectionFactory</res-ref-name>
            <jndi-name>jms/sample/QueueConnectionFactory</jndi-name>
            <default-resource-principal>
                <name>guest</name>
                <password>guest</password>
            </default-resource-principal>
        </resource-ref>
        <mdb-connection-factory>
            <jndi-name>jms/sample/QueueConnectionFactory</jndi-name>
            <default-resource-principal>
                <name>guest</name>
                <password>guest</password>
            </default-resource-principal>
        </mdb-connection-factory>
    </ejb>
</enterprise-beans>
</sun-ejb-jar>
```

Handling Transactions With Enterprise Beans

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

As a developer, you can write an application that updates data in multiple databases distributed across multiple sites. The site might use EJB servers from different vendors. This section provides overview information on the following topics:

Flat Transactions

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

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

Global and Local Transactions

Understanding the distinction between global and local transactions is crucial in understanding the Application Server support for transactions. See "Transaction Scope" on page 216.

Both local and global transactions are demarcated using the `javax.transaction.UserTransaction` interface, which the client must use. Local transactions bypass the transaction manager and are faster. For more information, see "The Transaction Manager, the Transaction Synchronization Registry, and `UserTransaction`" on page 217.

Commit Options

The EJB protocol is designed to give the container the flexibility to select the disposition of the instance state at the time a transaction is committed. This allows the container to best manage caching an entity object's state and associating an entity object identity with the EJB instances.

There are three commit-time options:

- **Option A** – The container caches a ready instance between transactions. The container ensures that the instance has exclusive access to the state of the object in persistent storage.

  In this case, the container does *not* have to synchronize the instance's state from the persistent storage at the beginning of the next transaction.

  Note – Commit option A is not supported for this Application Server release.

- **Option B** – The container caches a ready instance between transactions, but the container does *not* ensure that the instance has exclusive access to the state of the object in persistent storage. This is the default.

  In this case, the container must synchronize the instance's state by invoking `ejbLoad` from persistent storage at the beginning of the next transaction.

- **Option C** – The container does *not* cache a ready instance between transactions, but instead returns the instance to the pool of available instances after a transaction has completed.

  The life cycle for every business method invocation under commit option C looks like this.

  ```
  ejbActivate → ejbLoad → business method → ejbStore → ejbPassivate
  ```

  If there is more than one transactional client concurrently accessing the same entity, the first client gets the ready instance and subsequent concurrent clients get new instances from the pool.

The Application Server deployment descriptor has an element, `commit-option`, that specifies the commit option to be used. Based on the specified commit option, the appropriate handler is instantiated.

Administration and Monitoring

An administrator can control a number of domain-level Transaction Service settings. For details, see "Configuring the Transaction Service" on page 217.

The Transaction Timeout setting can be overridden by a bean. See "Bean-Level Container-Managed Transaction Timeouts" on page 136.

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. Also see Chapter 10, "Transactions," in *Sun Java System Application Server Platform Edition 9 Administration Guide*.

◆ ◆ ◆ CHAPTER 10

Using Container-Managed Persistence

This chapter contains information on how EJB 2.1 container-managed persistence (CMP) works in the Sun Java System Application Server in the following topics:

Application Server Support for CMP

Application Server support for EJB 2.1 CMP beans includes:

- Full support for the J2EE v1.4 specification's CMP model. Extensive information on CMP is contained in chapters 10, 11, and 14 of the Enterprise JavaBeans Specification, v2.1. This includes the following:
  - Support for commit options B and C for transactions. See "Commit Options" on page 149.
  - The primary key class must be a subclass of `java.lang.Object`. This ensures portability, and is noted because some vendors allow primitive types (such as `int`) to be used as the primary key class.
- The Application Server CMP implementation, which provides the following:
  - An Object/Relational (O/R) mapping tool that creates XML deployment descriptors for EJB JAR files that contain beans that use CMP.
  - Support for compound (multi-column) primary keys.
  - Support for sophisticated custom finder methods.
  - Standards-based query language (EJB QL).

- CMP runtime support. See "Configuring the CMP Resource" on page 163.

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

  For details, see "Performance-Related Features" on page 163.

CMP Mapping

Implementation for entity beans that use CMP is mostly a matter of mapping CMP fields and CMR fields (relationships) to the database. This section addresses the following topics:

Mapping Capabilities

Mapping refers to the ability to tie an object-based model to a relational model of data, usually the schema of a relational database. The CMP implementation provides the ability to tie a set of interrelated beans containing data and associated behaviors to the schema. This object representation of the database becomes part of the Java application. You can also customize this mapping to optimize these beans for the particular needs of an application. The result is a single data model through which both persistent database information and regular transient program data are accessed.

The mapping capabilities provided by the Application Server include:

- Mapping a CMP bean to one or more tables
- Mapping CMP fields to one or more columns
- Mapping CMP fields to different column types
- Mapping tables with compound primary keys
- Mapping tables with unknown primary keys
- Mapping CMP relationships to foreign keys
- Mapping tables with overlapping primary and foreign keys

The Mapping Deployment Descriptor File

Each module with CMP beans must have the following files:

- `ejb-jar.xml` – The J2EE standard file for assembling enterprise beans. For a detailed description, see the Enterprise JavaBeans Specification, v2.1.

- `sun-ejb-jar.xml` – The Application Server standard file for assembling enterprise beans. For a detailed description, see "The sun-ejb-jar.xml File" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.
- `sun-cmp-mappings.xml` – The *mapping deployment descriptor file*, which describes the mapping of CMP beans to tables in a database. For a detailed description, see "The sun-cmp-mappings.xml File" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

The `sun-cmp-mappings.xml` file can be automatically generated and does not have to exist prior to deployment. For details, see "Generation Options for CMP" on page 158.

The `sun-cmp-mappings.xml` file maps CMP fields and CMR fields (relationships) to the database. A primary table must be selected for each CMP bean, and optionally, multiple secondary tables. CMP fields are mapped to columns in either the primary or secondary table(s). CMR fields are mapped to pairs of column lists (normally, column lists are the lists of columns associated with primary and foreign keys).

Note – Table names in databases can be case-sensitive. Make sure that the table names in the `sun-cmp-mappings.xml` file match the names in the database.

Relationships should always be mapped to the primary key field(s) of the related table.

The `sun-cmp-mappings.xml` file conforms to the `sun-cmp-mapping_1_2.dtd` file and is packaged with the user-defined bean classes in the EJB JAR file under the `META-INF` directory.

The Application Server creates the mappings in the `sun-cmp-mappings.xml` file automatically during deployment if the file is not present.

To map the fields and relationships of your entity beans manually, edit the `sun-cmp-mappings.xml` deployment descriptor. Only do this if you are proficient in editing XML.

The mapping information is developed in conjunction with the database schema (`.dbschema`) file, which can be automatically captured when you deploy the bean (see "Automatic Database Schema Capture" on page 162). You can manually generate the schema using the `capture-schema` utility ("Using the `capture-schema` Utility" on page 162).

Mapping Considerations

This section addresses the following topics:

The data types used in automatic schema generation are also suggested for manual mapping. These data types are described in "Supported Data Types for CMP" on page 156.

Join Tables and Relationships

Use of join tables in the database schema is supported for all types of relationships, not just many-to-many relationships. For general information about relationships, see section 10.3.7 of the Enterprise JavaBeans Specification, v2.1.

Automatic Primary Key Generation

The Application Server supports automatic primary key generation for EJB 1.1, 2.0, and 2.1 CMP beans. To specify automatic primary key generation, give the `prim-key-class` element in the `ejb-jar-xml` file the value `java.lang.Object`. CMP beans with automatically generated primary keys can participate in relationships with other CMP beans. The Application Server does not support database-generated primary key values.

If the database schema is created during deployment, the Application Server creates the schema with the primary key column, then generates unique values for the primary key column at runtime.

If the database schema is not created during deployment, the primary key column in the mapped table must be of type `NUMERIC` with a precision of 19 or more, and must not be mapped to any CMP field. The Application Server generates unique values for the primary key column at runtime.

Fixed Length CHAR Primary Keys

If an existing database table has a primary key column in which the values vary in length, but the type is `CHAR` instead of `VARCHAR`, the Application Server automatically trims any extra spaces when retrieving primary key values. It is not a good practice to use a fixed length `CHAR` column as a primary key. Use this feature with schemas that cannot be changed, such as a schema inherited from a legacy application.

Managed Fields

A managed field is a CMP or CMR field that is mapped to the same database column as another CMP or CMR field. CMP fields mapped to the same column and CMR fields mapped to exactly the same column lists always have the same value in memory. For CMR fields that share only a subset of their mapped columns, changes to the columns affect the relationship fields in memory differently. Basically, the Application Server always tries to keep the state of the objects in memory synchronized with the database.

A managed field can have any `fetched-with` subelement except `<default/>`. See "fetched-with" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

BLOB Support

Binary Large Object (BLOB) is a data type used to store values that do not correspond to other types such as numbers, strings, or dates. Java fields whose types implement `java.io.Serializable` or are represented as `byte[]` can be stored as BLOBs.

If a CMP field is defined as `Serializable`, it is serialized into a `byte[]` before being stored in the database. Similarly, the value fetched from the database is deserialized. However, if a CMP field is defined as `byte[]`, it is stored directly instead of being serialized and deserialized when stored and fetched, respectively.

To enable BLOB support in the Application Server environment, define a CMP field of type `byte[]` or a user-defined type that implements the `java.io.Serializable` interface. If you map the CMP bean to an existing database schema, map the field to a column of type BLOB.

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

For a list of the JDBC drivers currently supported by the Application Server, see the *Sun Java System Application Server Platform Edition 9 Release Notes*. For configurations of supported and other drivers, see "Configurations for Specific JDBC Drivers" in *Sun Java System Application Server Platform Edition 9 Administration Guide*.

For automatic mapping, you might need to change the default BLOB column length for the generated schema using the `schema-generator-properties` element in `sun-ejb-jar.xml`. See your database vendor documentation to determine whether you need to specify the length. For example:

```
<schema-generator-properties>
   <property>
      <name>Employee.voiceGreeting.jdbc-type</name>
      <value>BLOB</value>
   </property>
   <property>
      <name>Employee.voiceGreeting.jdbc-maximum-length</name>
      <value>10240</value>
   </property>
   ...
</schema-generator-properties>
```

CLOB Support

Character Large Object (CLOB) is a data type used to store and retrieve very long text fields. CLOBs translate into long strings.

To enable CLOB support in the Application Server environment, define a CMP field of type `java.lang.String`. If you map the CMP bean to an existing database schema, map the field to a column of type CLOB.

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

For a list of the JDBC drivers currently supported by the Application Server, see the *Sun Java System Application Server Platform Edition 9 Release Notes*. For configurations of supported and other drivers, see "Configurations for Specific JDBC Drivers" in *Sun Java System Application Server Platform Edition 9 Administration Guide*.

For automatic mapping, you might need to change the default CLOB column length for the generated schema using the `schema-generator-properties` element in `sun-ejb-jar.xml`. See your database vendor documentation to determine whether you need to specify the length. For example:

```
<schema-generator-properties>
   <property>
      <name>Employee.resume.jdbc-type</name>
      <value>CLOB</value>
   </property>
   <property>
      <name>Employee.resume.jdbc-maximum-length</name>
      <value>10240</value>
   </property>
   ...
</schema-generator-properties>
```

Automatic Schema Generation for CMP

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

This section addresses the following topics:

- "Supported Data Types for CMP" on page 156
- "Generation Options for CMP" on page 158

Supported Data Types for CMP

CMP supports a set of JDBC data types that are used in mapping Java data fields to SQL types. Supported JDBC data types are as follows: BIGINT, BIT, BLOB, CHAR, CLOB, DATE, DECIMAL, DOUBLE, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT, TIME, TIMESTAMP, TINYINT, VARCHAR.

The following table contains the mappings of Java types to JDBC types when automatic mapping is used.

TABLE 10–1 Java Type to JDBC Type Mappings for CMP

Java Type	JDBC Type	Nullability
`boolean`	`BIT`	No

TABLE 10–1 Java Type to JDBC Type Mappings for CMP *(Continued)*

Java Type	JDBC Type	Nullability
`java.lang.Boolean`	`BIT`	Yes
`byte`	`TINYINT`	No
`java.lang.Byte`	`TINYINT`	Yes
`double`	`DOUBLE`	No
`java.lang.Double`	`DOUBLE`	Yes
`float`	`REAL`	No
`java.lang.Float`	`REAL`	Yes
`int`	`INTEGER`	No
`java.lang.Integer`	`INTEGER`	Yes
`long`	`BIGINT`	No
`java.lang.Long`	`BIGINT`	Yes
`short`	`SMALLINT`	No
`java.lang.Short`	`SMALLINT`	Yes
`java.math.BigDecimal`	`DECIMAL`	Yes
`java.math.BigInteger`	`DECIMAL`	Yes
`char`	`CHAR`	No
`java.lang.Character`	`CHAR`	Yes
`java.lang.String`	`VARCHAR` or `CLOB`	Yes
`Serializable`	`BLOB`	Yes
`byte[]`	`BLOB`	Yes
`java.util.Date`	`DATE` (Oracle only) `TIMESTAMP` (all other databases)	Yes
`java.sql.Date`	`DATE`	Yes
`java.sql.Time`	`TIME`	Yes
`java.sql.Timestamp`	`TIMESTAMP`	Yes

Note – Java types assigned to CMP fields must be restricted to Java primitive types, Java `Serializable` types, `java.util.Date`, `java.sql.Date`, `java.sql.Time`, or `java.sql.Timestamp`. An entity bean local interface type (or a collection of such) can be the type of a CMR field.

The following table contains the mappings of JDBC types to database vendor-specific types when automatic mapping is used. For a list of the JDBC drivers currently supported by the Application Server, see the *Sun Java System Application Server Platform Edition 9 Release Notes*. For configurations of supported and other drivers, see "Configurations for Specific JDBC Drivers" in *Sun Java System Application Server Platform Edition 9 Administration Guide*.

TABLE 10–2 Mappings of JDBC Types to Database Vendor Specific Types for CMP

JDBC Type	Java DB, Derby, CloudScape	Oracle	DB2	Sybase ASE 12.5	MS-SQL Server
BIT	SMALLINT	SMALLINT	SMALLINT	TINYINT	BIT
TINYINT	SMALLINT	SMALLINT	SMALLINT	TINYINT	TINYINT
SMALLINT	SMALLINT	SMALLINT	SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER	INTEGER	INTEGER	INTEGER
BIGINT	BIGINT	NUMBER	BIGINT	NUMERIC	NUMERIC
REAL	REAL	REAL	FLOAT	FLOAT	REAL
DOUBLE	DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE	DOUBLE PRECISION	FLOAT
DECIMAL(p,s)	DECIMAL(p,s)	NUMBER(p,s)	DECIMAL(p,s)	DECIMAL(p,s)	DECIMAL(p,s)
VARCHAR	VARCHAR	VARCHAR2	VARCHAR	VARCHAR	VARCHAR
DATE	DATE	DATE	DATE	DATETIME	DATETIME
TIME	TIME	DATE	TIME	DATETIME	DATETIME
TIMESTAMP	TIMESTAMP	TIMESTAMP(9)	TIMESTAMP	DATETIME	DATETIME
BLOB	BLOB	BLOB	BLOB	IMAGE	IMAGE
CLOB	CLOB	CLOB	CLOB	TEXT	NTEXT

Generation Options for CMP

Deployment descriptor elements or `asadmin` command line options can control automatic schema generation by the following:

- Creating tables during deployment
- Dropping tables during undeployment
- Dropping and creating tables during redeployment

- Specifying the database vendor
- Specifying that table names are unique
- Specifying type mappings for individual CMP fields

Note – Before using these options, make sure you have a properly configured CMP resource. See "Configuring the CMP Resource" on page 163.

For a read-only bean, do not create the database schema during deployment. Instead, work with your database administrator to populate the data into the tables. See "Using Read-Only Beans" on page 141.

Automatic schema generation is not supported for beans with version column consistency checking. Instead, work with your database administrator to create the schema and add the required triggers. See "Version Column Consistency Checking" on page 163.

The following optional data subelements of the `cmp-resource` element in the `sun-ejb-jar.xml` file control the automatic creation of database tables at deployment. For more information about the `cmp-resource` element, see "cmp-resource" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide* and "Configuring the CMP Resource" on page 163.

TABLE 10–3 The `sun-ejb-jar.xml` Generation Elements

Element	Default	Description
`create-tables-at-deploy`	`false`	If `true`, causes database tables to be created for beans that are automatically mapped by the EJB container. If `false`, does not create tables.
`drop-tables-at-undeploy`	`false`	If `true`, causes database tables that were automatically created when the bean(s) were last deployed to be dropped when the bean(s) are undeployed. If `false`, does not drop tables.
`database-vendor-name`	none	Specifies the name of the database vendor for which tables are created. Allowed values are `javadb`, `db2`, `mssql`, `oracle`, `postgresql`, `pointbase`, `derby` (also for CloudScape), and `sybase`, case-insensitive. If no value is specified, a connection is made to the resource specified by the `jndi-name` subelement of the `cmp-resource` element in the `sun-ejb-jar.xml` file, and the database vendor name is read. If the connection cannot be established, or if the value is not recognized, SQL-92 compliance is presumed.

TABLE 10–3 The `sun-ejb-jar.xml` Generation Elements *(Continued)*

Element	Default	Description
`schema-generator-properties`	none	Specifies field-specific column attributes in `property` subelements. Each property name is of the following format: *bean-name*`.`*field-name*`.`*attribute* For example: `Employee.firstName.jdbc-type` Also allows you to set the `use-unique-table-names` property. If `true`, this property specifies that generated table names are unique within each application server domain. The default is `false`. For further information and an example, see "schema-generator-properties" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

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

TABLE 10–4 The `asadmin deploy` and `asadmin deploydir` Generation Options for CMP

Option	Default	Description
`--createtables`	none	If `true`, causes database tables to be created for beans that need them. If `false`, does not create tables. If not specified, the value of the `create-tables-at-deploy` attribute in `sun-ejb-jar.xml` is used.
`--dropandcreatetables`	none	If `true`, and if tables were automatically created when this application was last deployed, tables from the earlier deployment are dropped and fresh ones are created. If `true`, and if tables were *not* automatically created when this application was last deployed, no attempt is made to drop any tables. If tables with the same names as those that would have been automatically created are found, the deployment proceeds, but a warning indicates that tables could not be created. If `false`, settings of `create-tables-at-deploy` or `drop-tables-at-undeploy` in the `sun-ejb-jar.xml` file are overridden.
`--uniquetablenames`	none	If `true`, specifies that table names are unique within each application server domain. If not specified, the value of the `use-unique-table-names` property in `sun-ejb-jar.xml` is used.

TABLE 10–4 The `asadmin deploy` and `asadmin deploydir` Generation Options for CMP *(Continued)*

Option	Default	Description
`--dbvendorname`	none	Specifies the name of the database vendor for which tables are created. Allowed values are `javadb`, `db2`, `mssql`, `oracle`, `postgresql`, `pointbase`, `derby` (also for CloudScape), and `sybase`, case-insensitive. If not specified, the value of the `database-vendor-name` attribute in `sun-ejb-jar.xml` is used. If no value is specified, a connection is made to the resource specified by the `jndi-name` subelement of the `cmp-resource` element in the `sun-ejb-jar.xml` file, and the database vendor name is read. If the connection cannot be established, or if the value is not recognized, SQL-92 compliance is presumed.

If one or more of the beans in the module are manually mapped and you use any of the `asadmin deploy` or `asadmin deploydir` options, the deployment is not harmed in any way, but the options have no effect, and a warning is written to the server log.

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

TABLE 10–5 The `asadmin undeploy` Generation Options for CMP

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

For more information about the `asadmin deploy`, `asadmin deploydir`, and `asadmin undeploy` commands, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

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

The `asant` tasks `sun-appserv-deploy` and `sun-appserv-undeploy` are equivalent to `asadmin deploy` and `asadmin undeploy`, respectively. These `asant` tasks also override the `sun-ejb-jar.xml` options. For details, see Chapter 3.

Schema Capture

This section addresses the following topics:

Automatic Database Schema Capture

You can configure a CMP bean in Application Server to automatically capture the database metadata and save it in a `.dbschema` file during deployment. If the `sun-cmp-mappings.xml` file contains an empty `<schema/>` entry, the `cmp-resource` entry in the `sun-ejb-jar.xml` file is used to get a connection to the database, and automatic generation of the schema is performed.

Note – Before capturing the database schema automatically, make sure you have a properly configured CMP resource. See "Configuring the CMP Resource" on page 163.

Using the `capture-schema` Utility

You can use the `capture-schema` command to manually generate the database metadata (`.dbschema`) file. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

The `capture-schema` utility does *not* modify the schema in any way. Its only purpose is to provide the persistence engine with information about the structure of the database (the schema).

Keep the following in mind when using the `capture-schema` command:

- The name of a `.dbschema` file must be unique across all deployed modules in a domain.
- If more than one schema is accessible for the schema user, more than one table with the same name might be captured if the `-schemaname` parameter of `capture-schema` is not set.
- The schema name must be upper case.
- Table names in databases are case-sensitive. Make sure that the table name matches the name in the database.
- PostgreSQL databases internally convert all names to lower case. Before running the `capture-schema` command on a PostgreSQL database, make sure table and column names are lower case in the `sun-cmp-mappings.xml` file.
- An Oracle database user running the `capture-schema` command needs ANALYZE ANY TABLE privileges if that user does not own the schema. These privileges are granted to the user by the database administrator.

Configuring the CMP Resource

An EJB module that contains CMP beans requires the JNDI name of a JDBC resource in the `jndi-name` subelement of the `cmp-resource` element in the `sun-ejb-jar.xml` file. Set `PersistenceManagerFactory` properties as properties of the `cmp-resource` element in the `sun-ejb-jar.xml` file. See "cmp-resource" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

In the Admin Console, open the Resources component, then select JDBC. Click the Help button in the Admin Console for information on creating a new CMP resource.

For a list of the JDBC drivers currently supported by the Application Server, see the *Sun Java System Application Server Platform Edition 9 Release Notes*. For configurations of supported and other drivers, see "Configurations for Specific JDBC Drivers" in *Sun Java System Application Server Platform Edition 9 Administration Guide*.

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

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

Performance-Related Features

The Application Server provides the following features to enhance performance or allow more fine-grained data checking. These features are supported only for entity beans with container managed persistence.

Note – Use of any of these features results in a non-portable application.

Version Column Consistency Checking

The version consistency feature saves the bean state at first transactional access and caches it between transactions. The state is copied from the cache instead of being read from the database. The bean state is verified by primary key and version column values at flush for custom queries (for dirty instances only) and at commit (for clean and dirty instances).

▼ To Use Version Consistency

1. **Create the version column in the primary table.**
2. **Give the version column a numeric data type.**
3. **Provide appropriate update triggers on the version column.**

   These triggers must increment the version column on each update of the specified row.
4. **Specify the version column.**

   This is specified in the `check-version-of-accessed-instances` subelement of the `consistency` element in the `sun-cmp-mappings.xml` file. See "consistency" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.
5. **Map the CMP bean to an existing schema.**

   Automatic schema generation is not supported for beans with version column consistency checking. Instead, work with your database administrator to create the schema and add the required triggers.

Relationship Prefetching

In many cases when an entity bean's state is fetched from the database, its relationship fields are always accessed in the same transaction. Relationship prefetching saves database round trips by fetching data for an entity bean and those beans referenced by its CMR fields in a single database round trip.

To enable relationship prefetching for a CMR field, use the `default` subelement of the `fetched-with` element in the `sun-cmp-mappings.xml` file. By default, these CMR fields are prefetched whenever `findByPrimaryKey` or a custom finder is executed for the entity, or when the entity is navigated to from a relationship. (Recursive prefetching is not supported, because it does not usually enhance performance.) See "fetched-with" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

To disable prefetching for specific custom finders, use the `prefetch-disabled` element in the `sun-ejb-jar.xml` file. See "prefetch-disabled" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

Read-Only Beans

Another feature that the Application Server provides is the *read-only bean*, an entity bean that is never modified by an EJB client. Read-only beans avoid database updates completely.

Note – Read-only beans are specific to the Application Server and are not part of the Enterprise JavaBeans Specification, v2.1. Use of this feature for an EJB 2.1 bean results in a non-portable application.

A read-only bean can be used to cache a database entry that is frequently accessed but rarely updated (externally by other beans). When the data that is cached by a read-only bean is updated by another bean, the read-only bean can be notified to refresh its cached data.

The Application Server provides a number of ways by which a read-only bean's state can be refreshed. By setting the `refresh-period-in-seconds` element in the `sun-ejb-jar.xml` file and the `trans-attribute` element (or `@TransactionAttribute` annotation) in the `ejb-jar.xml` file, it is easy to configure a read-only bean that is one of the following:

- Always refreshed
- Periodically refreshed
- Never refreshed
- Programmatically refreshed

Access to CMR fields of read-only beans is not supported. Deployment will succeed, but an exception will be thrown at runtime if a get or set method is invoked.

Read-only beans are best suited for situations where the underlying data never changes, or changes infrequently. For further information and usage guidelines, see "Using Read-Only Beans" on page 141.

Configuring Queries for 1.1 Finders

This section contains the following topics:

About JDOQL Queries

The Enterprise JavaBeans Specification, v1.1 does not specify the format of the finder method description. The Application Server uses an extension of Java Data Objects Query Language (JDOQL) queries to implement finder and selector methods. You can specify the following elements of the underlying JDOQL query:

- **Filter expression** - A Java-like expression that specifies a condition that each object returned by the query must satisfy. Corresponds to the WHERE clause in EJB QL.
- **Query parameter declaration** - Specifies the name and the type of one or more query input parameters. Follows the syntax for formal parameters in the Java language.

- **Query variable declaration** - Specifies the name and type of one or more query variables. Follows the syntax for local variables in the Java language. A query filter might use query variables to implement joins.
- **Query ordering declaration** - Specifies the ordering expression of the query. Corresponds to the ORDER BY clause of EJB QL.

The Application Server specific deployment descriptor (`sun-ejb-jar.xml`) provides the following elements to store the EJB 1.1 finder method settings:

```
query-filter
query-params
query-variables
query-ordering
```

The bean developer uses these elements to construct a query. When the finder method that uses these elements executes, the values of these elements are used to execute a query in the database. The objects from the JDOQL query result set are converted into primary key instances to be returned by the EJB 1.1 `ejbFind` method.

The JDO specification, JSR 12 (`http://jcp.org/en/jsr/detail?id=12`), provides a comprehensive description of JDOQL. The following information summarizes the elements used to define EJB 1.1 finders.

Query Filter Expression

The filter expression is a String containing a Boolean expression evaluated for each instance of the candidate class. If the filter is not specified, it defaults to true. Rules for constructing valid expressions follow the Java language, with the following differences:

- Equality and ordering comparisons between primitives and instances of wrapper classes are valid.
- Equality and ordering comparisons of Date fields and Date parameters are valid.
- Equality and ordering comparisons of String fields and String parameters are valid.
- White space (non-printing characters space, tab, carriage return, and line feed) is a separator and is otherwise ignored.
- The following assignment operators are not supported.
  - Comparison operators such as =, +=, and so on
  - Pre- and post-increment
  - Pre- and post-decrement
- Methods, including object construction, are not supported, except for these methods.

```
Collection.contains(Object o)
Collection.isEmpty()
String.startsWith(String s)
String.endsWith(String e)
```

In addition, the Application Server supports the following nonstandard JDOQL methods.

```
String.like(String pattern)
String.like(String pattern, char escape)
String.substring(int start, int length)
String.indexOf(String str)
String.indexOf(String str, int start)
String.length()
Math.abs(numeric n)
Math.sqrt(double d)
```

- Navigation through a null-valued field, which throws a `NullPointerException`, is treated as if the sub-expression returned `false`.

Note – Comparisons between floating point values are by nature inexact. Therefore, equality comparisons (== and !=) with floating point values should be used with caution. Identifiers in the expression are considered to be in the name space of the candidate class, with the addition of declared parameters and variables. As in the Java language, `this` is a reserved word, and refers to the current instance being evaluated.

The following expressions are supported.

- Relational operators (==, !=, >, <, >=, <=)
- Boolean operators (&, &&, |, ||, ~, !)
- Arithmetic operators (+, -, *, /)
- String concatenation, only for String + String
- Parentheses to explicitly mark operator precedence
- Cast operator
- Promotion of numeric operands for comparisons and arithmetic operations

The rules for promotion follow the Java rules extended by BigDecimal, BigInteger, and numeric wrapper classes. See the numeric promotions of the Java language specification.

Query Parameters

The parameter declaration is a String containing one or more parameter type declarations separated by commas. This follows the Java syntax for method signatures.

Query Variables

The type declarations follow the Java syntax for local variable declarations.

JDOQL Examples

This section provides a few query examples.

Example 1

The following query returns all players called Michael. It defines a filter that compares the name field with a string literal:

```
name == "Michael"
```

The `finder` element of the `sun-ejb-jar.xml` file looks like this:

```
<finder>
   <method-name>findPlayerByName</method-name>
   <query-filter>name == "Michael"</query-filter>
</finder>
```

Example 2

This query returns all products in a specified price range. It defines two query parameters which are the lower and upper bound for the price: double low, double high. The filter compares the query parameters with the price field:

```
low < price && price < high
```

Query ordering is set to `price ascending`.

The `finder` element of the `sun-ejb-jar.xml` file looks like this:

```
<finder>
   <method-name>findInRange</method-name>
   <query-params>double low, double high</query-params>
   <query-filter>low &lt; price &amp;&amp; price &lt high</query-filter>
   <query-ordering>price ascending</query-ordering>
</finder>
```

Example 3

This query returns all players having a higher salary than the player with the specified name. It defines a query parameter for the name `java.lang.String name`. Furthermore, it defines a variable to which the player's salary is compared. It has the type of the persistence capable class that corresponds to the bean:

```
    mypackage.PlayerEJB_170160966_JDOState player
```

The filter compares the salary of the current player denoted by the `this` keyword with the salary of the player with the specified name:

```
    (this.salary > player.salary) && (player.name == name)
```

The `finder` element of the `sun-ejb-jar.xml` file looks like this:

```
<finder>
   <method-name>findByHigherSalary</method-name>
   <query-params>java.lang.String name</query-params>
   <query-filter>
      (this.salary &gt; player.salary) &amp;&amp; (player.name == name)
   </query-filter>
   <query-variables>
      mypackage.PlayerEJB_170160966_JDOState player
   </query-variables>
</finder>
```

CMP Restrictions and Optimizations

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

Eager Loading of Field State

By default, the EJB container loads the state for all persistent fields (excluding relationship, BLOB, and CLOB fields) before invoking the `ejbLoad` method of the abstract bean. This approach might not be optimal for entity objects with large state if most business methods require access to only parts of the state.

Use the `fetched-with` element in `sun-cmp-mappings.xml` for fields that are used infrequently. See "fetched-with" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

Restrictions on Remote Interfaces

The following restrictions apply to the remote interface of an EJB 2.1 bean that uses CMP:

- Do not expose the `get` and `set` methods for CMR fields or the persistence collection classes that are used in container-managed relationships through the remote interface of the bean.

  However, you are free to expose the `get` and `set` methods that correspond to the CMP fields of the entity bean through the bean's remote interface.
- Do not expose the container-managed collection classes that are used for relationships through the remote interface of the bean.

- Do not expose local interface types or local home interface types through the remote interface or remote home interface of the bean.

Dependent value classes can be exposed in the remote interface or remote home interface, and can be included in the client EJB JAR file.

PostgreSQL Case Insensitivity

Case-sensitive behavior cannot be achieved for PostgresSQL databases. PostgreSQL databases internally convert all names to lower case, which makes the following workarounds necessary:

- In the CMP 2.1 runtime, PostgreSQL table and column names are not quoted, which makes these names case insensitive.
- Before running the `capture-schema` command on a PostgreSQL database, make sure table and column names are lower case in the `sun-cmp-mappings.xml` file.

No Support for `lock-when-loaded` on Sybase and DB2

For EJB 2.1 beans, the `lock-when-loaded` consistency level is implemented by placing update locks on the data corresponding to a bean when the data is loaded from the database. There is no suitable mechanism available on Sybase and DB2 databases to implement this feature. Therefore, the `lock-when-loaded` consistency level is not supported on Sybase and DB2 databases. See "consistency" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

Sybase Finder Limitation

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

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

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

Date and Time Fields

If a field type is a Java date or time type (`java.util.Date`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`), make sure that the field value exactly matches the value in the database.

For example, the following code uses a `java.sql.Date` type as a primary key field:

```
java.sql.Date myDate = new java.sql.Date(System.currentTimeMillis())
BeanA.create(myDate, ...);
```

For some databases, this code results in only the year, month, and date portion of the field value being stored in the database. Later if the client tries to find this bean by primary key as follows, the bean is not found in the database because the value does not match the one that is stored in the database.

```
myBean = BeanA.findByPrimaryKey(myDate);
```

Similar problems can happen if the database truncates the timestamp value while storing it, or if a custom query has a date or time value comparison in its WHERE clause.

For automatic mapping to an Oracle database, fields of type `java.util.Date`, `java.sql.Date`, and `java.sql.Time` are mapped to Oracle's DATE data type. Fields of type `java.sql.Timestamp` are mapped to Oracle's `TIMESTAMP(9)` data type.

Set `RECURSIVE_TRIGGERS` to `false` on MSSQL

For version consistency triggers on MSSQL, the property `RECURSIVE_TRIGGERS` must be set to `false`, which is the default. If set to `true`, triggers throw a `java.sql.SQLException`.

Set this property as follows:

```
EXEC sp_dboption 'database-name', 'recursive triggers', 'FALSE'
go
```

You can test this property as follows:

```
SELECT DATABASEPROPERTYEX('database-name', 'IsRecursiveTriggersEnabled')
go
```

MySQL Database Restrictions

The following restrictions apply when you use a MySQL database with the 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.

- Change the trigger create format from the following:

```
CREATE TRIGGER T_UNKNOWNPKVC1
BEFORE UPDATE ON UNKNOWNPKVC1
FOR EACH ROW
        WHEN (NEW.VERSION = OLD.VERSION)
BEGIN
        :NEW.VERSION := :OLD.VERSION + 1;
END;
/
```

  To the following:

```
DELIMITER |
CREATE TRIGGER T_UNKNOWNPKVC1
BEFORE UPDATE ON UNKNOWNPKVC1
FOR EACH ROW
        WHEN (NEW.VERSION = OLD.VERSION)
BEGIN
        :NEW.VERSION := :OLD.VERSION + 1;
END
```

```
|
DELIMITER ;
```

For more information, see http://dev.mysql.com/doc/mysql/en/create-trigger.html.

- MySQL does not allow a DELETE on a row that contains a reference to itself. Here is an example that illustrates the issue.

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

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

This example fails with the following error message.

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

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

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

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

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

- When an SQL script has foreign key constraints defined, `capture-schema` fails to capture the table information correctly. To work around the problem, remove the constraints and then run `capture-schema`. Here is an example that illustrates the issue.

```
CREATE TABLE ADDRESSBOOKBEANTABLE (ADDRESSBOOKNAME VARCHAR(255)
    NOT NULL PRIMARY KEY,
CONNECTEDUSERS              BLOB NULL,
OWNER                       VARCHAR(256),
```

```
FK_FOR_ACCESSPRIVILEGES     VARCHAR(256),
CONSTRAINT FK_ACCESSPRIVILEGE FOREIGN KEY (FK_FOR_ACCESSPRIVILEGES)
    REFERENCES ACCESSPRIVILEGESBEANTABLE (ROOT)
) ENGINE=InnoDB;
```

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

```
CREATE TABLE ADDRESSBOOKBEANTABLE (ADDRESSBOOKNAME VARCHAR(255)
    NOT NULL PRIMARY KEY,
CONNECTEDUSERS              BLOB NULL,
OWNER                       VARCHAR(256),
FK_FOR_ACCESSPRIVILEGES     VARCHAR(256)
) ENGINE=InnoDB;
```

◆ ◆ ◆ CHAPTER 11

Developing Java Clients

This chapter describes how to develop, assemble, and deploy Java clients in the following sections:

Introducing the Application Client Container

The Application Client Container (ACC) includes a set of Java classes, libraries, and other files that are required for and distributed with Java client programs that execute in their own Java Virtual Machine (JVM). The ACC manages the execution of Java EE application client components (application clients), which are used to access a variety of Java EE services (such as JMS resources, EJB components, web services, security, and so on.) from a JVM outside the Sun Java System Application Server.

The ACC communicates with the Application Server using RMI-IIOP protocol and manages the details of RMI-IIOP communication using the client ORB that is bundled with it. Compared to other Java EE containers, the ACC is lightweight.

ACC Security

The ACC is responsible for collecting authentication data such as the username and password and sending the collected data to the Application Server. The Application Server then processes the authentication data.

Authentication techniques are provided by the client container, and are not under the control of the application client component. The container integrates with the platform's authentication system. When you execute a client application, it displays a login window and collects authentication data from the user. It also supports SSL (Secure Socket Layer)/IIOP if configured and when necessary; see "Using RMI/IIOP Over SSL" on page 184.

Application clients can use "Programmatic Login" on page 91.

For more information about security for application clients, see the Java EE 5 Specification, Section EE.9.7, "Java EE Application Client XML Schema."

ACC Naming

The client container enables the application clients to use the Java Naming and Directory Interface (JNDI) to look up Java EE services (such as JMS resources, EJB components, web services, security, and so on.) and to reference configurable parameters set at the time of deployment.

ACC Annotation

Annotation is supported for application clients. For more information, see section 9.4 of the Java EE 5 Specification and "Java EE Standard Annotation" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

Java Web Start

Java Web Start allows your application client to be easily launched and automatically downloaded and updated. It is enabled for all application clients by default. For more information, see "Using Java Web Start" on page 179.

Developing Clients Using the ACC

This section describes the procedure to develop, assemble, and deploy client applications using the ACC. This section describes the following topics:

▼ To Access an EJB Component From an Application Client

1 In your client code, reference the EJB component by using an `@EJB` annotation or by looking up the JNDI name as defined in the `ejb-jar.xml` file.

For more information about annotations in application clients, see section 9.4 of the Java EE 5 Specification.

For more information about naming and lookups, see "Accessing the Naming Context" on page 221.

2 Define the `@EJB` annotations or the `ejb-ref` elements in the `application-client.xml` file. Define the corresponding `ejb-ref` elements in the `sun-application-client.xml` file.

For more information on the `application-client.xml` file, see the Java EE 5 Specification, Section EE.9.7, "Java EE Application Client XML Schema."

For more information on the `sun-application-client.xml` file, see "The sun-application-client.xml file" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*. For a general explanation of how to map JNDI names using reference elements, see "Mapping References" on page 225.

3 Deploy the application client and EJB component together in an application.

For more information on deployment, see the *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*. To get the client JAR file, use the `--retrieve` option of the `asadmin deploy` command.

To retrieve the stubs and ties whether or not you requested their generation during deployment, use the `asadmin get-client-stubs` command. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

4 Ensure that the client JAR file includes the following files:

- A Java class to access the bean.
- `application-client.xml` - (optional) Java EE application client deployment descriptor. For information on the `application-client.xml` file, see the Java EE 5 Specification, Section EE.9.7, "Java EE Application Client XML Schema."
- `sun-application-client.xml` - (optional) Application Server specific client deployment descriptor. For information on the `sun-application-client.xml` file, see "The sun-application-client.xml file" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.
- The `MANIFEST.MF` file. This file contains the `main` class, which states the complete package prefix and class name of the Java client.

If you are *not* using Java Web Start, you can package the application client using the `package-appclient` script. This is optional. See "Using the `package-appclient` Script" on page 183.

5 If you are *not* using Java Web Start, copy the following JAR files to the client machine and include them in the classpath on the client side:

- `appserv-rt.jar` - available at *install-dir*`/lib`
- `javaee.jar` - available at *install-dir*`/lib`
- The client JAR file

6 To access EJB components that are residing in a remote system, make the following changes to the `sun-acc.xml` file.

- Define the `target-server` element's `address` attribute to reference the remote server machine. See "target-server" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.
- Define the `target-server` element's `port` attribute to reference the ORB port on the remote server.

This information can be obtained from the `domain.xml` file on the remote system. For more information on `domain.xml` file, see the *Sun Java System Application Server Platform Edition 9 Administration Reference*.

For more information about the `sun-acc.xml` file, see "The sun-acc.xml File" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

7 Run the application client.

See "Using Java Web Start" on page 179 or "Running an Application Client Using the `appclient` Script" on page 183.

▼ To Access a JMS Resource From an Application Client

1 Create a JMS client.

For detailed instructions on developing a JMS client, see "Chapter 33: The Java Message Service API" in the Java EE 5 Tutorial (`http://java.sun.com/javaee/5/docs/tutorial/doc/index.html`).

2 Next, configure a JMS resource on the Application Server.

For information on configuring JMS resources, see "Creating JMS Resources: Destinations and Connection Factories" on page 231.

3 Define the `@Resource` or `@Resources` annotations or the `resource-ref` elements in the `application-client.xml` file. Define the corresponding `resource-ref` elements in the `sun-application-client.xml` file.

For more information on the `application-client.xml` file, see the Java EE 5 Specification, Section EE.9.7, "Java EE Application Client XML Schema."

For more information on the `sun-application-client.xml` file, see "The sun-application-client.xml file" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*. For a general explanation of how to map JNDI names using reference elements, see "Mapping References" on page 225.

4 Ensure that the client JAR file includes the following files:

- A Java class to access the resource.
- `application-client.xml` - (optional) Java EE application client deployment descriptor. For information on the `application-client.xml` file, see the Java EE 5 Specification, Section EE.9.7, "Java EE Application Client XML Schema."

- `sun-application-client.xml` - (optional) Application Server specific client deployment descriptor. For information on the `sun-application-client.xml` file, see "The sun-application-client.xml file" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.
- The `MANIFEST.MF` file. This file contains the `main` class, which states the complete package prefix and class name of the Java client.

If you are *not* using Java Web Start, you can package the application client using the `package-appclient` script. This is optional. See "Using the `package-appclient` Script" on page 183.

5 **If you are *not* using Java Web Start, copy the following JAR files to the client machine and include them in the classpath on the client side:**

- `appserv-rt.jar` - available at *install-dir*`/lib`
- `javaee.jar` - available at *install-dir*`/lib`
- `imqjmsra.jar` - available at *install-dir*`/lib/install/aplications/jmsra`
- The client JAR file

6 **Run the application client.**

See "Using Java Web Start" on page 179 or "Running an Application Client Using the `appclient` Script" on page 183.

Using Java Web Start

Java Web Start allows your application client to be easily launched and automatically downloaded and updated. General information about Java Web Start is available at `http://java.sun.com/products/javawebstart/reference/api/index.html`.

Java Web Start is discussed in the following topics:

Enabling and Disabling Java Web Start

Java Web Start is enabled for all application clients by default.

The application developer or deployer can specify that Java Web Start is always disabled for an application client by setting the value of the `eligible` element to `false` in the `sun-application-client.xml` file. See the *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

The Application Server administrator can disable Java Web Start for a previously deployed eligible application client using the `asadmin set` command.

To disable Java Web Start for all eligible application clients in an application, use the following command:

```
asadmin set --user adminuser
domain1.applications.j2ee-application.app-name.java-web-start-enabled="false"
```

To disable Java Web Start for one eligible application client in an application, use the following command:

```
asadmin set --user adminuser
domain1.applications.j2ee-application.app-name.module-name.java-web-start-enabled="false"
```

To disable Java Web Start for a stand-alone eligible application client, use the following command:

```
asadmin set --user adminuser
domain1.applications.appclient-module.module-name.java-web-start-enabled="false"
```

Setting `java-web-start-enabled="true"` re-enables Java Web Start for an eligible application client. For more information about the `asadmin set` command, see the *Sun Java System Application Server Platform Edition 9 Reference Manual.*

Downloading and Launching an Application Client

If Java Web Start is enabled for your deployed application client, you can launch it for testing. Simply click on the Launch button next to the application client or application's listing on the App Client Modules page in the Admin Console.

On other machines, you can download and launch the application client using Java Web Start in the following ways:

- Using a web browser, directly enter the URL for the application client. See "The Application Client URL" on page 181.
- Click on a link to the application client from a web page.
- Use the Java Web Start command `javaws`, specifying the URL of the application client as a command line argument.
- If the application has previously been downloaded using Java Web Start, you have additional alternatives.
  - Use the desktop icon that Java Web Start created for the application client. When Java Web Start downloads an application client for the first time it asks you if such an icon should be created.
  - Use the Java Web Start control panel to launch the application client.

When you launch an application client, Java Web Start contacts the server to see if a newer client version is available. This means you can redeploy an application client without having to worry about whether client machines have the latest version.

The Application Client URL

The default URL for an application is as follows:

`http://`*host*`:`*port*`/`*context-root*

The default URL for a stand-alone application client module is as follows:

`http://`*host*`:`*port*`/`*module-id*

If the *context-root* or *module-id* is not specified during deployment, the name of the EAR or JAR file without the extension is used. For an application, the relative path to the application client JAR file is also included. If the application or module is not in EAR or JAR file format, a *context-root* or *module-id* is generated.

Regardless of how the *context-root* or *module-id* is determined, it is written to the server log. For details about naming, see "Naming Standards" in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

To set a different URL for an application client, use the `context-root` subelement of the `java-web-start-access` element in the `sun-application-client.xml` file. See *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

You can also pass arguments to the ACC or to the application client's `main` method as query parameters in the URL. If multiple application client arguments are specified, they are passed in the order specified.

A question mark separates the context root from the arguments. Each argument and each value must begin with `arg=` and end with an ampersand (`&`). Here is an example URL with a `-color` argument for a stand-alone application client. The `-color` argument is passed to the application client's `main` method.

```
http://localhost:8080/testClient?arg=-color&arg=red
```

Note – If you are using the `javaws` *URL* command to launch Java Web Start with a URL that contains arguments, enclose the URL in double quotes (`"`) to avoid breaking the URL at the ampersand (`&`) symbol.

Ideally, you should build your production application clients with user-friendly interfaces that collect information which might otherwise be gathered as command-line arguments. This minimizes the degree to which users must customize the URLs that launch application clients using Java Web Start. Command-line argument support is useful in a development environment and for existing application clients that depend on it.

Signing JAR Files Used in Java Web Start

Java Web Start enforces a security sandbox. By default it grants any application, including application clients, only minimal privileges. Because Java Web Start applications can be so easily downloaded, Java Web Start provides protection from potentially harmful programs that might be

accessible over the network. If an application requires a higher privilege level than the sandbox permits, the code that needs privileges must be in a JAR file that was signed. When Java Web Start downloads such a signed JAR file, it displays information about the certificate that was used to sign the JAR, and it asks you whether you want to trust that signed code. If you agree, the code receives elevated permissions and runs. If you reject the signed code, Java Web Start does not start the downloaded application.

The Application Server serves two types of signed JAR files in response to Java Web Start requests. One type is a JAR file installed as part of the Application Server, which starts an application client during a Java Web Start launch: *install-dir*`/lib/appserv-jwsacc.jar`.

The other type is a generated application client JAR file. As part of deployment, the Application Server generates a new application client JAR file that contains classes, resources, and descriptors needed to run the application client on end-user systems. When you deploy an application with the `asadmin deploy` command's `--retrieve` option, use the `asadmin get-client-stubs` command, or select the Generate RMIStubs option in the Admin Console, this is the JAR file retrieved to your system. Because application clients need access beyond the minimal sandbox permissions to work in the Java Web Start environment, the generated application client JAR file must be signed before it can be downloaded to and executed on an end-user system.

A JAR file can be signed automatically or manually. The following sections describe the ways of signing JAR files.

- "Automatically Signing JAR Files" on page 182
- "Manually Signing `appserv-jwsacc.jar`" on page 183
- "Manually Signing the Generated Application Client JAR File" on page 183

Automatically Signing JAR Files

The Application Server automatically creates a signed version of the required JAR file if none exists. When a Java Web Start request for the `appserv-jwsacc.jar` file arrives, the Application Server looks for *domain-dir*`/java-web-start/appserv-jwsacc.jar`. When a request for an application's generated application client JAR file arrives, the Application Server looks in the directory *domain-dir*`/java-web-start/`*app-name* for a file with the same name as the generated JAR file created during deployment.

In either case, if the requested signed JAR file is absent or older than its unsigned counterpart, the Application Server creates a signed version of the JAR file automatically and deposits it in the relevant directory. Whether the Application Server just signed the JAR file or not, it serves the file from the *domain-dir*`/java-web-start` directory tree in response to the Java Web Start request.

To sign these JAR files, the Application Server uses its self-signed certificate. When you create a new domain, either by installing the Application Server or by using the `asadmin create-domain` command, the Application Server creates a self-signed certificate and adds it to the domain's key store.

A self-signed certificate is generally untrustworthy because no certification authority vouches for its authenticity. The automatic signing feature uses the same certificate to create all required signed JAR files. To sign different JAR files with different certificates, do the signing manually.

Manually Signing `appserv-jwsacc.jar`

You can sign the `appserv-jwsacc.jar` file manually any time after you have installed the Application Server. Copy the unsigned file from *install-dir*`/lib` to a different working directory and use the `jarsigner` command provided with the JDK to create a signed version of exactly the same name using your certificate. Then manually copy the signed file into *domain-dir*`/java-web-start`. From then on, the Application Server serves the JAR file signed with your certificate whenever a Java Web Start request asks that domain for the `appserv-jwsacc.jar` file. Note that you can sign each domain's `appserv-jwsacc.jar` file differently.

Remember that if you create a new domain and do not sign `appserv-jwsacc.jar` manually for that domain, the Application Server creates an auto-signed version of it for use by the new domain. Also, if you create a domain-specific signed `appserv-jwsacc.jar`, delete the domain, and then create a new domain with the same name as the just-deleted domain, the Application Server does not remember the earlier signed `appserv-jwsacc.jar`. You must recreate the manually signed version.

Manually Signing the Generated Application Client JAR File

You can sign the generated application client JAR file for an application any time after you have deployed the application. As you deploy the application, you can specify the `asadmin deploy` command's `--retrieve` option or select the Generate RMIStubs option in the Admin Console. Doing either of these tasks returns a copy of the generated application client JAR file to a directory you specify. Or, after you have deployed an application, you can download the generated application client JAR file using the `asadmin get-client-stubs` command.

Once you have a copy of the generated application client JAR file, you can sign it using the `jarsigner` tool and your certificate. Then place the signed JAR file in the *domain-dir*`/java-web-start/`*app-name* directory. You do not need to restart the server to start using the new signed JAR file.

Running an Application Client Using the `appclient` Script

To run an application client that does *not* have Java Web Start enabled, you can launch the ACC using the `appclient` script. This is optional. This script is located in the *install-dir*`/bin` directory. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

Using the `package-appclient` Script

You can package an application client that does *not* have Java Web Start enabled into a single `appclient.jar` file using the `package-appclient` script. This is optional. This script is located in the *install-dir*`/bin` directory. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

The `client.policy` File

The `client.policy` file is the J2SE policy file used by the application client. Each application client has a `client.policy` file. The default policy file limits the permissions of Java EE deployed application clients to the minimal set of permissions required for these applications to operate correctly. If an application client requires more than this default set of permissions, edit the `client.policy` file to add the custom permissions that your application client needs. Use the J2SE standard policy tool or any text editor to edit this file.

For more information on using the J2SE policy tool, see `http://java.sun.com/docs/books/tutorial/security1.2/tour2/index.html`.

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

Using RMI/IIOP Over SSL

You can configure RMI/IIOP over SSL in two ways: using a username and password, or using a client certificate.

To use a username and password, configure the `ior-security-config` element in the `sun-ejb-jar.xml` file. The following configuration establishes SSL between an application client and an EJB component using a username and password. The user has to login to the ACC using either the `sun-acc.xml` mechanism or the "Programmatic Login" on page 91 mechanism.

```
<ior-security-config>
  <transport-config>
    <integrity>required</integrity>
    <confidentiality>required</confidentiality>
    <establish-trust-in-target>supported</establish-trust-in-target>
    <establish-trust-in-client>none</establish-trust-in-client>
  </transport-config>
  <as-context>
    <auth-method>username_password</auth-method>
    <realm>default</realm>
    <required>true</required>
  </as-context>
 <sas-context>
    <caller-propagation>none</caller-propagation>
 </sas-context>
</ior-security-config>
```

For more information about the `sun-ejb-jar.xml` and `sun-acc.xml` files, see the *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

To use a client certificate, configure the `ior-security-config` element in the `sun-ejb-jar.xml` file. The following configuration establishes SSL between an application client and an EJB component using a client certificate.

```
<ior-security-config>
  <transport-config>
    <integrity>required</integrity>
    <confidentiality>required</confidentiality>
    <establish-trust-in-target>supported</establish-trust-in-target>
    <establish-trust-in-client>required</establish-trust-in-client>
  </transport-config>
  <as-context>
    <auth-method>none</auth-method>
    <realm>default</realm>
    <required>false</required>
  </as-context>
  <sas-context>
    <caller-propagation>none</caller-propagation>
  </sas-context>
</ior-security-config>
```

To use a client certificate, you must also specify the system properties for the keystore and truststore to be used in establishing SSL. To use SSL with the Application Client Container (ACC), you need to set `VMARGS` environment variable in one of the following ways:

- Set the environment variable `VMARGS` in the shell. For example, in the `ksh` or `bash` shell, the command to set this environment variable would be as follows:

  ```
  export VMARGS="-Djavax.net.ssl.keyStore=${keystore.db.file}
  -Djavax.net.ssl.trustStore=${truststore.db.file}
  -Djavax.net.ssl.keyStorePass word=${ssl.password}
  -Djavax.net.ssl.trustStorePassword=${ssl.password}"
  ```

- Set the `env` element in the `asant` script (see Chapter 3). For example:

  ```
  <target name="runclient">
    <exec executable="${S1AS_HOME}/bin/appclient">
      <env key="VMARGS" value=" -Djavax.net.ssl.keyStore=${keystore.db.file}
        -Djavax.net.ssl.trustStore=${truststore.db.file}
        -Djavax.net.ssl.keyStorePasword=${ssl.password}
        -Djavax.net.ssl.trustStorePassword=${ssl.password}"/>
      <arg value="-client"/>
      <arg value="${appClient.jar}"/>
    </exec>
  </target>
  ```

◆ ◆ ◆ CHAPTER 12

Developing Connectors

This chapter describes Sun Java System Application Server support for the J2EE™ 1.5 Connector Architecture (CA).

The J2EE Connector Architecture provides a Java solution to the problem of connectivity between multiple application servers and existing enterprise information systems (EISs). By using the J2EE Connector architecture, EIS vendors no longer need to customize their product for each application server. Application server vendors who conform to the J2EE Connector architecture do not need to write custom code to add connectivity to a new EIS.

This chapter uses the terms *connector* and *resource adapter* interchangeably. Both terms refer to a resource adapter module that is developed in conformance with the J2EE Connector Specification.

For more information about connectors, see J2EE Connector Architecture (`http://java.sun.com/j2ee/connector/`) and "Chapter 37: J2EE Connector Architecture" in the Java EE 5 Tutorial (`http://java.sun.com/javaee/5/docs/tutorial/doc/index.html`).

For connector examples, see `http://developers.sun.com/prodtech/appserver/reference/techart/as8_connectors`.

This chapter includes the following topics:

Connector Support in the Application Server

The Application Server supports the development and deployment of resource adapters that are compatible with Connector specification (and, for backward compatibility, the Connector 1.0 specification).

The Connector 1.0 specification defines the outbound connectivity system contracts between the resource adapter and the Application Server. The Connector 1.5 specification introduces major additions in defining system level contracts between the Application Server and the resource adapter with respect to the following:

- Inbound connectivity from an EIS - Defines the transaction and message inflow system contracts for achieving inbound connectivity from an EIS. The message inflow contract also serves as a standard message provider pluggability contract, thereby allowing various providers of messaging systems to seamlessly plug in their products with any application server that supports the message inflow contract.
- Resource adapter life cycle management and thread management - These features are available through the lifecycle and work management contracts.

Connector Architecture for JMS and JDBC

In the Admin Console, connector, JMS, and JDBC resources are handled differently, but they use the same underlying Connector architecture. In the Application Server, all communication to an EIS, whether to a message provider or an RDBMS, happens through the Connector architecture. To provide JMS infrastructure to clients, the Application Server uses the Sun Java System Message Queue software. To provide JDBC infrastructure to clients, the Application Server uses its own JDBC system resource adapters. The application server automatically makes these system resource adapters available to any client that requires them.

For more information about JMS in the Application Server, see Chapter 18. For more information about JDBC in the Application Server, see Chapter 15.

Connector Configuration

The Application Server does not need to use `sun-ra.xml`, which previous Application Server versions used, to store server-specific deployment information inside a Resource Adapter Archive (RAR) file. (However, the `sun-ra.xml` file is still supported for backward compatibility.) Instead, the information is stored in the server configuration. As a result, you can create multiple connector connection pools for a connection definition in a functional resource adapter instance, and you can create multiple user-accessible connector resources (that is, registering a resource with a JNDI name) for a connector connection pool. In addition, dynamic changes can be made to connector connection pools and the connector resource properties without restarting the Application Server.

Deploying and Configuring a Stand-Alone Connector Module

You can deploy a stand-alone connector module using the Admin Console or the `asadmin` command. For information about using the Admin Console, click the Help button in the Admin Console. For information about using the `asadmin` command, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

Deploying a stand-alone connector module allows multiple deployed Java EE applications to share the connector module. A resource adapter configuration is automatically created for the connector module.

▼ To Deploy and Configure a Stand-Alone Connector Module

1 Deploy the connector module in one of the following ways.

- In the Admin Console, open the Applications component and select Connector Modules. When you deploy the connector module, a resource adapter configuration is automatically created for the connector module.
- Use the `asadmin deploy` or `asadmin deploydir` command. To override the default configuration properties of a resource adapter, if necessary, use the `asadmin create-resource-adapter-config` command.

2 Configure connector connection pools for the deployed connector module in one of the following ways:

- In the Admin Console, open the Resources component, select Connectors, and select Connector Connection Pools.
- Use the `asadmin create-connector-connection-pool` command.

3 Configure connector resources for the connector connection pools in one of the following ways.

- In the Admin Console, open the Resources component, select Connectors, and select Connector Resources.
- Use the `asadmin create-connector-resource` command.

This associates a connector resource with a JNDI name.

4 Create an administered object for an inbound resource adapter, if necessary, in one of the following ways:

- In the Admin Console, open the Resources component, select Connectors, and select Admin Object Resources.
- Use the `asadmin create-admin-object` command.

Redeploying a Stand-Alone Connector Module

Redeployment of a connector module maintains all connector connection pools, connector resources, and administered objects defined for the previously deployed connector module. You need not reconfigure any of these resources.

However, you should redeploy any dependent modules. A dependent module uses or refers to a connector resource of the redeployed connector module. Redeployment of a connector module results in the shared class loader reloading the new classes. Other modules that refer to the old resource adapter classes must be redeployed to gain access to the new classes. For more information about class loaders, see Chapter 2.

During connector module redeployment, the server log provides a warning indicating that all dependent applications should be redeployed. Client applications or application components using the connector module's resources may throw class cast exceptions if dependent applications are not redeployed after connector module redeployment.

To disable automatic redeployment, set the `--force` option to `false`. In this case, if the connector module has already been deployed, the Application Server provides an error message.

Deploying and Configuring an Embedded Resource Adapter

A connector module can be deployed as a Java EE component in a Java EE application. Such connectors are only visible to components residing in the same Java EE application. Simply deploy this Java EE application as you would any other Java EE application.

You can create new connector connection pools and connector resources for a connector module embedded within a Java EE application by prefixing the connector name with *app-name#*. For example, if an application `appX.ear` has `jdbcra.rar` embedded within it, the connector connection pools and connector resources refer to the connector module as `appX#jdbcra`.

However, an embedded connector module cannot be undeployed using the name *app-name#connector-name*. To undeploy the connector module, you must undeploy the application in which it is embedded.

The association between the physical JNDI name for the connector module in the Application Server and the logical JNDI name used in the application component is specified in the Application Server specific XML descriptor `sun-ejb-jar.xml`.

Advanced Connector Configuration Options

You can use these advanced connector configuration options:

Thread Pools

Connectors can submit work instances to the Application Server for execution. By default, the Application Server services work requests for all connectors from its default thread pool. However, you can associate a specific user-created thread pool to service work requests from a connector. A thread pool can service work requests from multiple resource adapters. To create a thread pool:

- In the Admin Console, select Thread Pools under the relevant configuration. For details, click the Help button in the Admin Console.
- Use the `asadmin create-threadpool` command. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

To associate a connector with a thread pool:

- In the Admin Console, open the Applications component and select Connector Modules. Deploy the module, or select the previously deployed module. Specify the name of the thread pool in the Thread Pool ID field. For details, click the Help button in the Admin Console.
- Use the `--threadpoolid` option of the `asadmin create-resource-adapter-config` command. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

If you create a resource adapter configuration for a connector module that is already deployed, the connector module deployment is restarted with the new configuration properties.

Security Maps

Create a security map for a connector connection pool to map an application principal or a user group to a back end EIS principal. The security map is usually used in situations where one or more EIS back end principals are used to execute operations (on the EIS) initiated by various principals or user groups in the application.

To create or update security maps for a connector connection pool:

- In the Admin Console, open the Resources component, select Connectors, select Connector Connection Pools, and select the Security Maps tab. For details, click the Help button in the Admin Console.

- Use the `asadmin create-connector-security-map` command. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

If a security map already exists for a connector connection pool, the new security map is appended to the previous one. The connector security map configuration supports the use of the wildcard asterisk (*) to indicate all users or all user groups.

When an application principal initiates a request to an EIS, the Application Server first checks for an exact match to a mapped back end EIS principal using the security map defined for the connector connection pool. If there is no exact match, the Application Server uses the wild card character specification, if any, to determined the mapped back end EIS principal.

Overriding Configuration Properties

You can override the properties (`config-property` elements) specified in the `ra.xml` file of a resource adapter. Use the `asadmin create-resource-adapter-config` command to create a configuration for a resource adapter. Use this command's `--property` option to specify a name-value pair for a resource adapter property.

You can use the `asadmin create-resource-adapter-config` command either before or after resource adapter deployment. If it is executed after deploying the resource adapter, the existing resource adapter is restarted with the new properties. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

Testing a Connector Connection Pool

After configuring a connector connection pool, use the `asadmin ping-connection-pool` command to test the health of the underlying connections. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

Handling Invalid Connections

If a resource adapter generates a `ConnectionErrorOccured` event, the Application Server considers the connection invalid and removes the connection from the connection pool. Typically, a resource adapter generates a `ConnectionErrorOccured` event when it finds a `ManagedConnection` object unusable. Reasons can be network failure with the EIS, EIS failure, fatal problems with resource adapter, and so on. If the `fail-all-connections` property in the connection pool configuration is set to `true`, all connections are destroyed and the pool is recreated.

The `is-connection-validation-required` property specifies whether connections have to be validated before being given to the application. If a resource's validation fails, it is destroyed, and a new resource is created and returned.

You can set the `fail-all-connections` and `is-connection-validation-required` configuration properties during creation of a connector connection pool. Or, you can use the `asadmin set` command to dynamically reconfigure a previously set property. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

The interface `ValidatingManagedConnectionFactory` exposes the method `getInvalidConnections` to allow retrieval of the invalid connections. The Application Server checks if the resource adapter implements this interface, and if it does, invalid connections are removed when the connection pool is resized.

Setting the Shutdown Timeout

According to the Connector specification, while an application server shuts down, all resource adapters should be stopped. A resource adapter might hang during shutdown, since shutdown is typically a resource intensive operation. To avoid such a situation, you can set a timeout that aborts resource adapter shutdown if exceeded. The default timeout is 30 seconds per resource adapter module. To configure this timeout:

- In the Admin Console, select Connector Service under the relevant configuration and edit the shutdown Timeout field. For details, click the Help button in the Admin Console.
- Use the following command:

```
asadmin set --user adminuser server.connector-service.shutdown-timeout-in-seconds="num-secs"
```

For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

The Application Server deactivates all message-driven bean deployments before stopping a resource adapter.

Using Last Agent Optimization of Transactions

Transactions that involve multiple resources or multiple participant processes are *distributed* or *global* transactions. A global transaction can involve one non-XA resource if last agent optimization is enabled. Otherwise, all resources must be XA. For more information about transactions in the Application Server, see Chapter 16.

The Connector specification requires that if a resource adapter supports `XATransaction`, the `ManagedConnection` created from that resource adapter must support both distributed and local transactions. Therefore, even if a resource adapter supports `XATransaction`, you can configure its connector connection pools as non-XA or without transaction support for better performance. A non-XA resource adapter becomes the last agent in the transactions in which it participates.

The value of the connection pool configuration property `transaction-support` defaults to the value of the `transaction-support` property in the `ra.xml` file. The connection pool configuration property can override the `ra.xml` file property if the transaction level in the connection pool configuration property is lower. If the value in the connection pool configuration property is higher, it is ignored.

Inbound Communication Support

The Connector specification defines the transaction and message inflow system contracts for achieving inbound connectivity from an EIS. The message inflow contract also serves as a standard message provider pluggability contract, thereby allowing various message providers to seamlessly plug in their products with any application server that supports the message inflow contract. In the inbound communication model, the EIS initiates all communication to an application. An application can be composed of enterprise beans (session, entity, or message-driven beans), which reside in an EJB container.

Incoming messages are received through a message endpoint, which is a message-driven bean. This message-driven bean asynchronously consumes messages from a message provider. An application can also synchronously send and receive messages directly using messaging style APIs.

A resource adapter supporting inbound communication provides an instance of an `ActivationSpec` JavaBean class for each supported message listener type. Each class contains a set of configurable properties that specify endpoint activation configuration information during message-driven bean deployment. The required `config-property` element in the `ra.xml` file provides a list of configuration property names required for each activation specification. An endpoint activation fails if the required property values are not specified. Values for the properties that are overridden in the message-driven bean's deployment descriptor are applied to the `ActivationSpec` JavaBean when the message-driven bean is deployed.

Administered objects can also be specified for a resource adapter, and these JavaBeans are specific to a messaging style or message provider. For example, some messaging styles may need applications to use special administered objects (such as `Queue` and `Topic` objects in JMS). Applications use these objects to send and synchronously receive messages using connection objects using messaging style APIs. For more information about administered objects, see Chapter 18.

Configuring a Message Driven Bean to Use a Resource Adapter

The Connectors specification's message inflow contract provides a generic mechanism to plug in a wide-range of message providers, including JMS, into a Java-EE-compatible application server. Message providers use a resource adapter and dispatch messages to message endpoints, which are implemented as message-driven beans.

The message-driven bean developer provides activation configuration information in the message-driven bean's `ejb-jar.xml` file. Configuration information includes messaging-style-specific configuration details, and possibly message-provider-specific details as well. The message-driven bean deployer uses this configuration information to set up the activation specification JavaBean. The activation configuration properties specified in `ejb-jar.xml` override configuration properties in the activation specification definition in the `ra.xml` file.

According to the EJB specification, the messaging-style-specific descriptor elements contained within the activation configuration element are not specified because they are specific to a messaging

provider. In the following sample message-driven bean `ejb-jar.xml`, a message-driven bean has the following activation configuration property names: `destinationType`, `SubscriptionDurability`, and `MessageSelector`.

```
<!--  A sample MDB that listens to a JMS Topic -->
<!-- message-driven bean deployment descriptor -->
...
 <activation-config>
   <activation-config-property>
     <activation-config-property-name>
       destinationType
     </activation-config-property-name>
     <activation-config-property-value>
       javax.jms.Topic
     </activation-config-property-value>
  </activation-config-property>
  <activation-config-property>
     <activation-config-property-name>
       SubscriptionDurability
     </activation-config-property-name>
     <activation-config-property-value>
       Durable
     </activation-config-property-value>
  </activation-config-property>
  <activation-config-property>
     <activation-config-property-name>
       MessageSelector
     </activation-config-property-name>
     <activation-config-property-value>
       JMSType = 'car' AND color = 'blue'
     </activation-config-property-value>
  </activation-config-property>
 ...
 </activation-config>
...
```

When the message-driven bean is deployed, the value for the `resource-adapter-mid` element in the `sun-ejb-jar.xml` file is set to the resource adapter module name that delivers messages to the message endpoint (to the message-driven bean). In the following example, the `jmsra` JMS resource adapter, which is the bundled resource adapter for the Sun Java System Message Queue message provider, is specified as the resource adapter module identifier for the `SampleMDB` bean.

```
<sun-ejb-jar>
<enterprise-beans>
    <unique-id>1</unique-id>
    <ejb>
       <ejb-name>SampleMDB</ejb-name>
       <jndi-name>SampleQueue</jndi-name>
```

```
    <!-- JNDI name of the destination from which messages would be
         delivered from MDB needs to listen to -->
    ...
    </ejb>
    <mdb-resource-adapter>
    <resource-adapter-mid>jmsra</resource-adapter-mid>
    <!-- Resource Adapter Module Id that would deliver messages to
         this message endpoint -->
    </mdb-resource-adapter>
...
</sun-ejb-jar>
```

When the message-driven bean is deployed, the Application Server uses the `resourceadapter-mid` setting to associate the resource adapter with a message endpoint through the message inflow contract. This message inflow contract with the application server gives the resource adapter a handle to the `MessageEndpointFactory` and the `ActivationSpec` JavaBean, and the adapter uses this handle to deliver messages to the message endpoint instances (which are created by the `MessageEndpointFactory`).

When a message-driven bean first created for use on the Application Server 7 is deployed, the Connector runtime transparently transforms the previous deployment style to the current connector-based deployment style. If the deployer specifies neither a `resource-adapter-mid` property nor the Message Queue resource adapter's activation configuration properties, the Connector runtime maps the message-driven bean to the `jmsra` system resource adapter and converts the JMS-specific configuration to the Message Queue resource adapter's activation configuration properties.

◆ ◆ ◆ CHAPTER 13

Developing Lifecycle Listeners

Lifecycle listener modules provide a means of running short or long duration Java-based tasks within the application server environment, such as instantiation of singletons or RMI servers. These modules are automatically initiated at server startup and are notified at various phases of the server life cycle.

All lifecycle module classes and interfaces are in the *install-dir*`/lib/appserv-rt.jar` file.

For Javadoc tool pages relevant to lifecycle modules, go to `http://glassfish.dev.java.net/nonav/javaee5/api/index.html` and click on the `com.sun.appserv.server` package.

The following sections describe how to create and use a lifecycle listener module:

Server Life Cycle Events

A lifecycle module listens for and performs its tasks in response to the following events in the server life cycle:

- After the `INIT_EVENT`, the server reads the configuration, initializes built-in subsystems (such as security and logging services), and creates the containers.
- After the `STARTUP_EVENT`, the server loads and initializes deployed applications.
- After the `READY_EVENT`, the server is ready to service requests.
- After the `SHUTDOWN_EVENT`, the server destroys loaded applications and stops.
- After the `TERMINATION_EVENT`, the server closes the containers, the built-in subsystems, and the server runtime environment.

These events are defined in the `LifecycleEvent` class.

The lifecycle modules that listen for these events implement the `LifecycleListener` interface.

The `LifecycleListener` Interface

To create a lifecycle module is to configure a customized class that implements the `com.sun.appserv.server.LifecycleListener` interface. You can create and simultaneously execute multiple lifecycle modules.

The `LifecycleListener` interface defines this method:

```
public void handleEvent(com.sun.appserv.server.LifecycleEvent event)
throws ServerLifecycleException
```

This method responds to a lifecycle event and throws a `com.sun.appserv.server.ServerLifecycleException` if an error occurs.

A sample implementation of the `LifecycleListener` interface is the `LifecycleListenerImpl.java` file, which you can use for testing lifecycle events.

The `LifecycleEvent` Class

The `com.sun.appserv.server.LifecycleEvent` class defines a server life cycle event. The following methods are associated with the event:

- `public java.lang.Object getData()`

  This method returns an instance of `java.util.Properties` that contains the properties defined for the lifecycle module in the `domain.xml` file. For more information about the `domain.xml` file, see the *Sun Java System Application Server Platform Edition 9 Administration Reference*.

- `public int getEventType()`

  This method returns the type of the last event, which is `INIT_EVENT`, `STARTUP_EVENT`, `READY_EVENT`, `SHUTDOWN_EVENT`, or `TERMINATION_EVENT`.

- `public com.sun.appserv.server.LifecycleEventContext getLifecycleEventContext()`

  This method returns the lifecycle event context, described next.

A `LifecycleEvent` instance is passed to the `LifecycleListener.handleEvent` method.

The Server Lifecycle Event Context

The `com.sun.appserv.server.LifecycleEventContext` interface exposes runtime information about the server. The lifecycle event context is created when the `LifecycleEvent` class is instantiated at server initialization. The `LifecycleEventContext` interface defines these methods:

- `public java.lang.String[] getCmdLineArgs()`

  This method returns the server startup command-line arguments.
- `public java.lang.String getInstallRoot()`

  This method returns the server installation root directory.
- `public java.lang.String getInstanceName()`

  This method returns the server instance name.
- `public javax.naming.InitialContext getInitialContext()`

  This method returns the initial JNDI naming context. The naming environment for lifecycle modules is installed after the `STARTUP_EVENT`. A lifecycle module can look up any resource by its `jndi-name` attribute after the `READY_EVENT`.

If a lifecycle module needs to look up resources, it can do so after the `READY_EVENT`. It can use the `getInitialContext()` method to get the initial context to which all the resources are bound.

Deploying a Lifecycle Module

You can deploy a lifecycle module using the following tools:

- In the Admin Console, open the Applications component and go to the Lifecycle Modules page. For details, click the Help button in the Admin Console.
- Use the `asadmin create-lifecycle-module` command. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

You do not need to specify a classpath for the lifecycle module if you place it in the *domain-dir*`/lib` or *domain-dir*`/lib/classes` directory.

After you deploy a lifecycle module, you must restart the server to activate it. The server instantiates it and registers it as a lifecycle event listener at server initialization.

Note – If the `is-failure-fatal` setting is set to `true` (the default is `false`), lifecycle module failure prevents server initialization or startup, but not shutdown or termination.

Considerations for Lifecycle Modules

The resources allocated at initialization or startup should be freed at shutdown or termination. The lifecycle module classes are called synchronously from the main server thread, therefore it is important to ensure that these classes don't block the server. Lifecycle modules can create threads if appropriate, but these threads must be stopped in the shutdown and termination phases.

The LifeCycleModule class loader is the parent class loader for lifecycle modules. Each lifecycle module's `classpath` in `domain.xml` is used to construct its class loader. All the support classes needed by a lifecycle module must be available to the LifeCycleModule class loader or its parent, the Connector class loader.

You must ensure that the `server.policy` file is appropriately set up, or a lifecycle module trying to perform a `System.exec()` might cause a security access violation. For details, see "The `server.policy` File" on page 80.

The configured properties for a lifecycle module are passed as properties after the `INIT_EVENT`. The JNDI naming context is not available before the `STARTUP_EVENT`. If a lifecycle module requires the naming context, it can get this after the `STARTUP_EVENT`, `READY_EVENT`, or `SHUTDOWN_EVENT`.

◆ ◆ ◆ CHAPTER 14

Developing Custom MBeans

An MBean is a managed Java object, similar to a JavaBean™, that follows the design patterns set forth in the instrumentation level of the Java Management Extensions (JMX) specification. An MBean can represent a device, an application, or any resource that needs to be managed. MBeans expose a management interface: a set of readable and/or writable attributes and a set of invokable operations, along with a self-description. The actual runtime interface of an MBean depends on the type of that MBean. MBeans can also emit notifications when certain defined events occur. Unlike other components, MBeans have no annotations or deployment descriptors.

The Sun Java System Application Server supports the development of custom MBeans as part of the self-management infrastructure or as separate applications. All types of MBeans (standard, dynamic, open, and model) are supported. For more about self-management, see Chapter 20 and Chapter 17, "Configuring Management Rules," in *Sun Java System Application Server Platform Edition 9 Administration Guide*.

For general information about JMX technology, including how to download the JMX specification, see `http://java.sun.com/products/JavaManagement/index.jsp`.

For a useful overview of JMX technology, see `http://java.sun.com/j2se/1.5.0/docs/guide/jmx/overview/JMXoverviewTOC.html`.

For a tutorial of JMX technology, see `http://java.sun.com/j2se/1.5.0/docs/guide/jmx/tutorial/tutorialTOC.html`.

This chapter includes the following topics:

The MBean Life Cycle

The MBean life cycle proceeds as follows:

1. The MBean's class files are installed in the Application Server. See "MBean Class Loading" on page 203.
2. The MBean is deployed using the `asadmin create-mbean` command or the Admin Console. See "Creating, Deleting, and Listing MBeans" on page 203.
3. The MBean class is loaded. This also results in loading of other classes. The delegation model is used. See the class loader diagram in "The Class Loader Hierarchy" on page 37.
4. The MBean is instantiated. Its default constructor is invoked reflectively. This is why the MBean class must have a default constructor.
5. The MBean's `ObjectName` is determined according to the following algorithm.
   - If you specify the `ObjectName`, it is used as is. The domain must be `user:`. The property name `server` is reserved and cannot be used.

     The Application Server automatically appends `server=server` to the `ObjectName` when the MBean is registered.
   - If the MBean implements the `MBeanRegistration` interface, it must provide an `ObjectName` in its `preregister()` method that follows the same rules.
   - If the `ObjectName` is not specified directly or through the `MBeanRegistration` interface, the default is `user:type=`*impl-class-name*.
6. All attributes are set using `setAttribute` calls in the order in which the attributes are specified. Attempting to specify a read-only attribute results in an error.

   If attribute values are set during MBean deployment, these values are passed in as `String` objects. Therefore, attribute types must be Java classes having constructors that accept `String` objects. If you specify an attribute that does not have such a constructor, an error is reported.

   Attribute values specified during MBean deployment are persisted to the Application Server configuration. Changes to attributes after registration through a JMX connector such as JConsole do not affect the Application Server configuration. To change an attribute value in the Application Server configuration, use the `asadmin set` command. See "Handling MBean Attributes" on page 206.
7. If the MBean is enabled, the `MBeanServer.registerMBean(Object, ObjectName)` method is used to register the MBean in the `MBeanServer`. This is the only method called by the Application Server runtime. See "The `MBeanServer` in the Application Server" on page 205.

   MBeans are enabled by default. Disabling an MBean deregisters it. See "Enabling and Disabling MBeans" on page 206.
8. The MBean is automatically loaded, instantiated, and registered upon each server restart.
9. When the MBean is deleted using the `asadmin delete-mbean` command or the Admin Console, the MBean is first deregistered if it is enabled, then the MBean definition is deleted from the configuration. The class files are not deleted, however.

MBean Class Loading

After you develop a custom MBean, copy its class files (or JAR file) into the MBean class loader directory, *domain-dir*`/applications/mbeans`. You have two choices of where to place any dependent classes:

- Common class loader – Copy the classes as JAR files into the *domain-dir*`/lib` directory, or copy the classes as `.class` files into the *domain-dir*`/lib/classes` directory. The classes are loaded when you restart the Application Server. The classes are available to all other MBeans and all deployed applications and modules.
- MBean class loader – Copy the classes into the *domain-dir*`/applications/mbeans` directory. No restart is required. The classes are available to all other MBeans, but *not* to all deployed applications and modules.

After copying the classes, register the MBean using the `asadmin create-mbean` command. See "The `asadmin create-mbean` Command" on page 203.

For general information about Application Server class loaders, see Chapter 2.

Creating, Deleting, and Listing MBeans

This section describes the following commands:

- Use the `asadmin create-mbean` command to deploy, or *register*, an MBean.
- Use the `asadmin delete-mbean` command to undeploy an MBean.
- Use the `asadmin list-mbeans` command to list deployed MBeans.

To perform these tasks using the Admin Console, open the Custom MBeans component. For details, click the Help button in the Admin Console.

The `asadmin create-mbean` Command

After installing the MBean classes as explained in "MBean Class Loading" on page 203, use the `asadmin create-mbean` command to deploy the MBean. This registers the MBean in the `MBeanServer` that is part of the Application Server runtime environment. For more information about the `MBeanServer`, see "The `MBeanServer` in the Application Server" on page 205.

Here is a simple example of an `asadmin create-mbean` command in which `TextPatterns` is the implementation class. The `--attributes` option is not required.

```
asadmin create-mbean --user adminuser --attributes color=red:font=Times TextPatterns
```

Other options not included in the example are as follows:

- `--name` defaults to the implementation class name

- `--objectname` is explained in "The MBean Life Cycle" on page 202
- `--enabled` defaults to `true` and is explained in "Enabling and Disabling MBeans" on page 206

All options must precede the implementation class.

For full details on the `asadmin create-mbean` command, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

For more information about MBean attributes, see "Handling MBean Attributes" on page 206.

Note – To redeploy an MBean, simply install its new classes into the Application Server as described in "MBean Class Loading" on page 203. Then either restart the server or use `asadmin delete-mbean` followed by `asadmin create-mbean`.

The `asadmin delete-mbean` Command

To undeploy an MBean, use the `asadmin delete-mbean` command. This removes its registration from the `MBeanServer`, but does not delete its code. Here is an example `asadmin delete-mbean` command in which `TextPatterns` is the implementation class.

```
asadmin delete-mbean --user adminuser  TextPatterns
```

For full details on the `asadmin delete-mbean` command, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

The `asadmin list-mbeans` Command

To list MBeans that have been deployed, use the `asadmin list-mbeans` command. Note that this command only lists the MBean definitions and not the MBeans registered in the `MBeanServer`. Here is an example `asadmin list-mbeans` command.

```
asadmin list-mbeans --user adminuser
```

The output of the `asadmin list-mbeans` command lists the following information:

- Implementation class – The name of the implementation class without the extension.
- Name – The name of the registered MBean, which defaults to but may be different from the implementation class name.
- Object name – The `ObjectName` of the MBean, which is explained in "The MBean Life Cycle" on page 202.
- Object type – For custom MBeans, the object type is always `user`. System MBeans have other object types.
- Enabled – Whether the MBean is enabled. MBeans are enabled by default. See "Enabling and Disabling MBeans" on page 206.

For full details on the `asadmin list-mbeans` command, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

The `MBeanServer` in the Application Server

Custom MBeans are registered in the `PlatformMBeanServer` returned by the `java.lang.management.ManagementFactory.getPlatformMBeanServer()` method. This `MBeanServer` is associated with a standard JMX connector server.

You can use any JMX connector to look up MBeans in this `MBeanServer` just as you would any other `MBeanServer`. If your JMX connector is remote, you can connect to this `MBeanServer` using the following information:

- Host name of the Application Server machine
- `MBeanServer` port, which is `8686` by default
- Administrator username
- Administrator password

For example, if you use JConsole, you can enter this information under the Remote tab. JConsole is a generic JMX connector you can use to look up and manage MBeans. For more information about JConsole, see `http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html`, the JMX tutorial at `http://java.sun.com/j2se/1.5.0/docs/guide/jmx/tutorial/tutorialTOC.html`, and "Using JConsole" in *Sun Java System Application Server Platform Edition 9 Administration Guide*.

The connection to this `MBeanServer` is non-SSL by default.

If SSL is enabled, you must provide the location of the truststore that contains the server certificate that the JMX connector should trust. For example, if you are using JConsole, you supply this location at the command line as follows:

```
jconsole -J-Djavax.net.ssl.trustStore=home-directory/.asadmintruststore
```

Look up the MBean by its name. By default, the name is the same as the implementation class.

You can reconfigure the JMX connector server's naming service port in one of the following ways:

- In the Admin Console, open the Admin Service component under the relevant configuration, select the system subcomponent, edit the Port field, and select Save. For details, click the Help button in the Admin Console.
- Use the `asadmin set` command as in the following example:

  ```
  asadmin set --user adminuser server.admin-service.jmx-connector.system.port=8687
  ```

  For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

Enabling and Disabling MBeans

A custom MBean is enabled by default. You can disable an MBean during deployment by using the `asadmin create-mbean` command's optional `--enabled=false` option. See "The `asadmin create-mbean` Command" on page 203.

After deployment, you can disable an MBean using the `asadmin set` command. For example:

```
asadmin set --user adminuser server.applications.mbean.TextPatterns.enabled=false
```

If the MBean name is different from the implementation class, you must use the name in the `asadmin set` command. In this example, the name is `TextPatterns`.

For full details on the `asadmin set` command, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

Handling MBean Attributes

You can set MBean attribute values that are not read-only in the following ways:

- In the MBean code itself, which does not affect the Application Server configuration
- During deployment using the `asadmin create-mbean` command
- During deployment using the Custom MBeans component in the Admin Console
- Using the `asadmin set` command
- Using a JMX connector such as JConsole, which does not affect the Application Server configuration

In the Application Server configuration, MBean attributes are stored as properties. Therefore, using the `asadmin set` command means editing properties. For example:

```
asadmin set --user adminuser server.applications.mbean.TextPatterns.property.color=blue
```

If the MBean name is different from the implementation class, you must use the MBean name in the `asadmin set` command. In this example, the name is `TextPatterns`.

For full details on the `asadmin set` command, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

PART III

Using Services and APIs

◆ ◆ ◆ CHAPTER 15

Using the JDBC API for Database Access

This chapter describes how to use the Java™ Database Connectivity (JDBC) API for database access with the Sun Java System Application Server. This chapter also provides high level JDBC implementation instructions for servlets and EJB components using the Application Server. The Application Server supports the JDBC 3.0 API, which encompasses the JDBC 2.0 Optional Package 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 "Configurations for Specific JDBC Drivers" in *Sun Java System Application Server Platform Edition 9 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 210
- "Making the JDBC Driver JAR Files Accessible" on page 210

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 *Sun Java System Application Server Platform Edition 9 Release Notes*. For configurations of supported and other drivers, see "Configurations for Specific JDBC Drivers" in *Sun Java System Application Server Platform Edition 9 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 any application or module across the domain. For more information about Application Server class loaders, see Chapter 2.

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, open the JDBC 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 *Sun Java System Application Server Platform Edition 9 Reference Manual*.

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, open the JDBC 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 *Sun Java System Application Server Platform Edition 9 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, open the JDBC 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 *Sun Java System Application Server Platform Edition 9 Reference Manual*.

Creating Applications That Use the JDBC API

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

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 2, "JDBC Resources," in *Sun Java System Application Server Platform Edition 9 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.DataSource ds = (com.sun.appserv.DataSource)
   ctx.lookup("jdbc/MyBase");
Connection con = ds.getConnection();
Connection drivercon = ds.getConnection(con);
// Do db operations.
con.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 *Sun Java System Application Server Platform Edition 9 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 16 and Chapter 10, "Transactions," in *Sun Java System Application Server Platform Edition 9 Administration Guide*. For information about last agent optimization, which can improve performance, see "Transaction Scope" on page 216.

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 15–1 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.

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 210.

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

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

- Check the Allow Non Component Callers box on the 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 *Sun Java System Application Server Platform Edition 9 Reference Manual*.
- Create a JDBC resource with a `__pm` suffix.

◆ ◆ ◆ CHAPTER 16

Using the Transaction Service

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

This chapter contains the following sections:

For more information about the Java™ Transaction API (JTA) and Java Transaction Service (JTS), see Chapter 10, "Transactions," in *Sun Java System Application Server Platform Edition 9 Administration Guide* and the following sites: `http://java.sun.com/products/jta/` and `http://java.sun.com/products/jts/`.

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

Transaction Resource Managers

There are three types of transaction resource managers:

- Databases - Use of transactions prevents databases from being left in inconsistent states due to incomplete updates. For information about JDBC transaction isolation levels, see "Using JDBC Transaction Isolation Levels" on page 213.

The Application Server supports a variety of JDBC XA drivers. For a list of the JDBC drivers currently supported by the Application Server, see the *Sun Java System Application Server Platform Edition 9 Release Notes*. For configurations of supported and other drivers, see "Configurations for Specific JDBC Drivers" in *Sun Java System Application Server Platform Edition 9 Administration Guide*.

- Java Message Service (JMS) Providers - Use of transactions ensures that messages are reliably delivered. The Application Server is integrated with Sun Java System Message Queue, a fully capable JMS provider. For more information about transactions and the JMS API, see Chapter 18.
- J2EE Connector Architecture (CA) components - Use of transactions prevents legacy EIS systems from being left in inconsistent states due to incomplete updates. For more information about connectors, see Chapter 12.

For details about how transaction resource managers, the transaction service, and applications interact, see Chapter 10, "Transactions," in *Sun Java System Application Server Platform Edition 9 Administration Guide*.

Transaction Scope

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

In the 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.

Transactions that involve multiple resources or multiple participant processes are *distributed* or *global* transactions. A global transaction can involve one non-XA resource if last agent optimization is enabled. Otherwise, all resourced must be XA. The `use-last-agent-optimization` property is set to `true` by default. For details about how to set this property, see "Configuring the Transaction Service" on page 217.

If only one XA resource is used in a transaction, one-phase commit occurs, otherwise the transaction is coordinated with a two-phase commit protocol.

A two-phase commit protocol between the transaction manager and all the resources enlisted for a transaction ensures that either all the resource managers commit the transaction or they all abort.

When the application requests the commitment of a transaction, the transaction manager issues a PREPARE_TO_COMMIT request to all the resource managers involved. Each of these resources can in turn send a reply indicating whether it is ready for commit (PREPARED) or not (NO). Only when all the resource managers are ready for a commit does the transaction manager issue a commit request (COMMIT) to all the resource managers. Otherwise, the transaction manager issues a rollback request (ABORT) and the transaction is rolled back.

Configuring the Transaction Service

You can configure the transaction service in the Application Server in the following ways:

- To configure the transaction service using the Admin Console, open the Transaction Service component under the relevant configuration. For details, click the Help button in the Admin Console.
- To configure the transaction service, use the asadmin set command to set the following attributes.

```
server.transaction-service.automatic-recovery = false
server.transaction-service.heuristic-decision = rollback
server.transaction-service.keypoint-interval = 2048
server.transaction-service.retry-timeout-in-seconds = 600
server.transaction-service.timeout-in-seconds = 0
server.transaction-service.tx-log-dir = domain-dir/logs
```

You can also set these properties:

```
server.transaction-service.property.oracle-xa-recovery-workaround = false
server.transaction-service.property.disable-distributed-transaction-logging = false
server.transaction-service.property.xaresource-txn-timeout = 600
server.transaction-service.property.pending-txn-cleanup-interval = 60
server.transaction-service.property.use-last-agent-optimization = true
server.transaction-service.property.db-logging-resource = jdbc/TxnDS
```

You can use the asadmin get command to list all the transaction service attributes and properties. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

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:appserver/TransactionManager. You can access the Application Server transaction synchronization registry, a javax.transaction.TransactionSynchronizationRegistry implementation, using the JNDI

subcontext `java:appserver/TransactionSynchronizationRegistry`. 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`.

Transaction Logging

The transaction service writes transactional activity into transaction logs so that transactions can be recovered. You can control transaction logging in these ways:

- Set the location of the transaction log files using the Transaction Log Location setting in the Admin Console, or set the `tx-log-dir` attribute using the `asadmin set` command.
- Turn off transaction logging by setting the `disable-distributed-transaction-logging` property to `true` and the `automatic-recovery` attribute to `false`. Do this *only* if performance is more important than transaction recovery.

Storing Transaction Logs in a Database

For multi-core machines, logging transactions to a database may be more efficient.

To log transactions to a database, follow these steps:

1. Create a JDBC connection Pool, and set the `non-transactional-connections` attribute to `true`.
2. Create a JDBC resource that uses the connection pool and note the JNDI name of the JDBC resource.
3. Create a table named `txn_log_table` with the schema shown in Table 16–1.
4. Add the `db-logging-resource` property to the transaction service. For example:

```
asadmin set --user adminuser server.transaction-service.property.db-logging-resource="jdbc/TxnDS"
```

   The property's value should be the JNDI name of the JDBC resource configured previously.
5. To disable file synchronization, use the following `asadmin create-jvm-options` command:

   ```
   asadmin create-jvm-options --user adminuser -Dcom.sun.appserv.transaction.nofdsync
   ```
6. Restart the server.

For information about JDBC connection pools and resources, see Chapter 15. For more information about the `asadmin create-jvm-options` command, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

TABLE 16–1 Schema for `txn_log_table`

Column Name	JDBC Type
`LOCALTID`	`BIGINT`
`SERVERNAME`	`VARCHAR(n)`
`GTRID`	`VARBINARY`

The size of the `SERVERNAME` column should be at least the length of the Application Server host name plus 10 characters.

The size of the `GTRID` column should be at least 64 bytes.

Recovery Workarounds

The Application Server provides workarounds for some known issues with the recovery implementations of the following JDBC drivers. These workarounds are used unless explicitly disabled.

In the Oracle thin driver, the `XAResource.recover` method repeatedly returns the same set of in-doubt Xids regardless of the input flag. According to the XA specifications, the Transaction Manager initially calls this method with TMSTARTSCAN and then with TMNOFLAGS repeatedly until no Xids are returned. The `XAResource.commit` method also has some issues.

To disable the Application Server workaround, set the `oracle-xa-recovery-workaround` property value to `false`. For details about how to set this property, see "Configuring the Transaction Service" on page 217.

Note – These workarounds do not imply support for any particular JDBC driver.

◆ ◆ ◆ CHAPTER 17

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:

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.

The rest of this section covers these topics:

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`, or `sun-application-client.xml` deployment descriptor file has the highest precedence. See "Mapping References" on page 225.
2. A global JNDI name assigned in a `mapped-name` element in the `ejb-jar.xml`, `web.xml`, or `application-client.xml` deployment descriptor file has the second highest precedence. The following elements have `mapped-name` subelements: `resource-ref`, `resource-env-ref`, `ejb-ref`, `message-destination`, `message-destination-ref`, `session`, `message-driven`, and `entity`.
3. A global JNDI name assigned in a `mappedName` attribute of an annotation has the third highest precedence. The following annotations have `mappedName` attributes: `@javax.annotation.Resource`, `@javax.ejb.EJB`, `@javax.ejb.Stateless`, `@javax.ejb.Stateful`, and `@javax.ejb.MessageDriven`.
4. A default global JNDI name is assigned in some cases if no name is assigned in deployment descriptors or annotations.
   - For an EJB 2.x dependency or a session or entity bean with a remote interface, the default is the fully qualified name of the home interface.
   - For an EJB 3.0 dependency or a session bean with a remote interface, the default is the fully qualified name of the remote business interface.
   - If both EJB 2.x and EJB 3.0 remote interfaces are specified, or if more than one 3.0 remote interface is specified, there is no default, and the global JNDI name must be specified.
   - For all other component dependencies that must be mapped to global JNDI names, the default is the name of the dependency relative to `java:comp/env`. For example, in the `@Resource(name="jdbc/Foo") DataSource ds;` annotation, the global JNDI name is `jdbc/Foo`.

Accessing EJB Components Using the `CosNaming` Naming Context

The preferred way of accessing the naming service, even in code that runs outside of a Java EE container, is to use the no-argument `InitialContext` constructor. However, if EJB client code explicitly instantiates an `InitialContext` that points to the CosNaming naming service, it is necessary to set the `java.naming.factory.initial` property to `com.sun.jndi.cosnaming.CNCtxFactory` in the client JVM when accessing EJB components. You can set this property as a command-line argument, as follows:

```
-Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
```

Or you can set this property in the code, as follows:

```
Properties properties = null;
      try {
            properties = new Properties();
            properties.put("java.naming.factory.initial",
                  "com.sun.jndi.cosnaming.CNCtxFactory");
      ...
```

Accessing EJB Components in a Remote Application Server

The recommended approach for looking up an EJB component in a remote Application Server from a client that is a servlet or EJB component is to use the Interoperable Naming Service syntax. Host and port information is prepended to any global JNDI names and is automatically resolved during the lookup. The syntax for an interoperable global name is as follows:

`corbaname:iiop:`*host*`:`*port*`#a/b/`*name*

This makes the programming model for accessing EJB components in another Application Server exactly the same as accessing them in the same server. The deployer can change the way the EJB components are physically distributed without having to change the code.

For Java EE components, the code still performs a `java:comp/env` lookup on an EJB reference. The only difference is that the deployer maps the `ejb-reference` element to an interoperable name in an Application Server deployment descriptor file instead of to a simple global JNDI name.

For example, suppose a servlet looks up an EJB reference using `java:comp/env/ejb/Foo`, and the target EJB component has a global JNDI name of `a/b/Foo`.

The `ejb-ref` element in `sun-web.xml` looks like this:

```
<ejb-ref>
   <ejb-ref-name>ejb/Foo</ejb-ref-name>
   <jndi-name>corbaname:iiop:host:port#a/b/Foo</jndi-name>
<ejb-ref>
```

The code looks like this:

```
Context ic = new InitialContext();
Object o = ic.lookup("java:comp/env/ejb/Foo");
```

For a client that doesn't run within a Java EE container, the code just uses the interoperable global name instead of the simple global JNDI name. For example:

```
Context ic = new InitialContext();
Object o = ic.lookup("corbaname:iiop:host:port#a/b/Foo");
```

Objects stored in the interoperable naming context and component-specific (`java:comp/env`) naming contexts are transient. On each server startup or application reloading, all relevant objects are re-bound to the namespace.

Naming Environment for Lifecycle Modules

Lifecycle listener modules provide a means of running short or long duration tasks based on Java technology within the application server environment, such as instantiation of singletons or RMI servers. These modules are automatically initiated at server startup and are notified at various phases of the server life cycle. For details about lifecycle modules, see Chapter 13.

The configured properties for a lifecycle module are passed as properties during server initialization (the `INIT_EVENT`). The initial JNDI naming context is not available until server initialization is complete. A lifecycle module can get the `InitialContext` for lookups using the method `LifecycleEventContext.getInitialContext()` during, and only during, the `STARTUP_EVENT`, `READY_EVENT`, or `SHUTDOWN_EVENT` server life cycle events.

Configuring Resources

The Application Server exposes the following special resources in the naming environment. Full administration details are provided in the following sections:

- "External JNDI Resources" on page 224
- "Custom Resources" on page 225

External JNDI Resources

An external JNDI resource defines custom JNDI contexts and implements the `javax.naming.spi.InitialContextFactory` interface. There is no specific JNDI parent context for external JNDI resources, except for the standard `java:comp/env/`.

Create an external JNDI resource in one of these ways:

- To create an external JNDI resource using the Admin Console, open the Resources component, open the JNDI component, and select External Resources. For details, click the Help button in the Admin Console.
- To create an external JNDI resource, use the `asadmin create-jndi-resource` command. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

Custom Resources

A custom resource specifies a custom server-wide resource object factory that implements the `javax.naming.spi.ObjectFactory` interface. There is no specific JNDI parent context for external JNDI resources, except for the standard `java:comp/env/`.

Create a custom resource in one of these ways:

- To create a custom resource using the Admin Console, open the Resources component, open the JNDI component, and select Custom Resources. For details, click the Help button in the Admin Console.
- To create a custom resource, use the `asadmin create-custom-resource` command. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

Using a Custom `jndi.properties` File

To use a custom `jndi.properties` file, specify the path to the file in one of the following ways:

- Use the Admin Console. Select the Application Server component, select the JVM Settings tab, select the Path Settings tab, and edit the Classpath Prefix field. For details, click the Help button in the Admin Console.
- Edit the `classpath-prefix` attribute of the `java-config` element in the `domain.xml` file. For details about `domain.xml`, see the *Sun Java System Application Server Platform Edition 9 Administration Reference*.

This adds the `jndi.properties` file to the Shared Chain class loader. For more information about class loading, see Chapter 2.

For each property found in more than one `jndi.properties` file, the Java EE naming service either uses the first value found or concatenates all of the values, whichever makes sense.

Mapping References

The following XML elements in the Application Server deployment descriptors map resource references in application client, 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`, `sun-ejb-ref.xml`, and `sun-application-client.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 *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*.

The rest of this section uses an example of a JDBC resource lookup to describe how to reference resource factories. The same principle is applicable to all resources (such as JMS destinations, JavaMail sessions, and so on).

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

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

This references a resource with the JNDI name of `java:comp/env/jdbc/helloDbDs`. If this is the JNDI name of the JDBC resource configured in the 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>
```

◆ ◆ ◆ CHAPTER 18

Using the Java Message Service

This chapter describes how to use the Java™ Message Service (JMS) API. The Sun Java System Application Server has a fully integrated JMS provider: the Sun Java System Message Queue software.

For general information about the JMS API, see "Chapter 33: The Java Message Service API" in the Java EE 5 Tutorial (`http://java.sun.com/javaee/5/docs/tutorial/doc/index.html`).

For detailed information about JMS concepts and JMS support in the Application Server, see Chapter 3, "Configuring Java Message Service Resources," in *Sun Java System Application Server Platform Edition 9 Administration Guide*.

This chapter contains the following sections:

The JMS Provider

The Application Server support for JMS messaging, in general, and for message-driven beans, in particular, requires messaging middleware that implements the JMS specification: a JMS provider. The Application Server uses the Sun Java System Message Queue software as its native JMS provider. The Message Queue software is tightly integrated into theApplication Server, providing transparent JMS messaging support. This support is known within Application Server as the *JMS Service*. The JMS Service requires only minimal administration.

The relationship of the Message Queue software to the Application Server can be one of these types: `EMBEDDED`, `LOCAL`, or `REMOTE`. The effects of these choices on the Message Queue broker life cycle are as follows:

- If the type is `EMBEDDED`, the Application Server and Message Queue software run in the same JVM. The Message Queue broker is started and stopped automatically by the Application Server. This is the default.

  Lazy initialization starts the default embedded broker on the first access of JMS services rather than at Application Server startup.
- If the type is `LOCAL`, the Message Queue broker starts when the Application Server starts.
- If the type is `REMOTE`, the Message Queue broker must be started separately. For information about starting the broker, see the *Sun Java System Message Queue 3 2006Q2 Administration Guide*.

For more information about setting the type and the default JMS host, see "Configuring the JMS Service" on page 229.

For more information about the Message Queue software, refer to the documentation at `http://docs.sun.com/app/docs/coll/1343.3`.

For general information about the JMS API, see the JMS web page at `http://java.sun.com/products/jms/index.html`.

Message Queue Resource Adapter

The Sun Java System Message Queue software is integrated into the Application Server using a resource adapter that is compliant with the Connector specification. The module name of this system resource adapter is `jmsra`. Every JMS resource is converted to a corresponding connector resource of this resource adapter as follows:

- **Connection Factory** – A connector connection pool with a `max-pool-size` of `250` and a corresponding connector resource
- **Destination (Topic or Queue)** – A connector administered object

You use connector configuration tools to manage JMS resources. For more information, see Chapter 12.

Generic Resource Adapter

The Application Server provides a generic resource adapter for JMS. For details, see `http://genericjmsra.dev.java.net/` and "Configuring the Generic Resource Adapter for JMS" in *Sun Java System Application Server Platform Edition 9 Administration Guide*.

Administration of the JMS Service

To configure the JMS Service and prepare JMS resources for use in applications deployed to the Application Server, you must perform these tasks:

For more information about JMS administration tasks, see Chapter 3, "Configuring Java Message Service Resources," in *Sun Java System Application Server Platform Edition 9 Administration Guide* and the *Sun Java System Message Queue 3 2006Q2 Administration Guide*.

Configuring the JMS Service

The JMS Service configuration is available to all inbound and outbound connections pertaining to the Application Server. You can edit the JMS Service configuration in the following ways:

- To edit the JMS Service configuration using the Admin Console, open the Java Message Service component under the relevant configuration. For details, click the Help button in the Admin Console.
- To configure the JMS service, use the `asadmin set` command to set the following attributes:

```
server.jms-service.init-timeout-in-seconds = 60
server.jms-service.type = EMBEDDED
server.jms-service.start-args =
server.jms-service.default-jms-host = default_JMS_host
server.jms-service.reconnect-interval-in-seconds = 5
server.jms-service.reconnect-attempts = 3
server.jms-service.reconnect-enabled = true
server.jms-service.addresslist-behavior = random
server.jms-service.addresslist-iterations = 3
server.jms-service.mq-scheme = mq
server.jms-service.mq-service = jms
```

  You can also set these properties:

```
server.jms-service.property.instance-name = imqbroker
server.jms-service.property.instance-name-suffix =
server.jms-service.property.append-version = false
```

You can use the `asadmin get` command to list all the JMS service attributes and properties. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

You can override the JMS Service configuration using JMS connection factory settings. For details, see Chapter 3, "Configuring Java Message Service Resources," in *Sun Java System Application Server Platform Edition 9 Administration Guide*.

Note – The Application Server must be restarted after configuration of the JMS Service.

The Default JMS Host

A JMS host refers to a Sun Java System Message Queue broker. A default JMS host for the JMS service is provided, named `default_JMS_host`. This is the JMS host that the Application Server uses for performing all Message Queue broker administrative operations, such as creating and deleting JMS destinations.

If you have created a multi-broker cluster in the Message Queue software, delete the default JMS host, then add the Message Queue cluster's brokers as JMS hosts. In this case, the default JMS host becomes the first JMS host in the `AddressList`. For more information about the `AddressList`, see "JMS Connection Features" on page 232. You can also explicitly set the default JMS host; see "Configuring the JMS Service" on page 229.

When the Application Server uses a Message Queue cluster, it executes Message Queue specific commands on the default JMS host. For example, when a physical destination is created for a Message Queue cluster of three brokers, the command to create the physical destination is executed on the default JMS host, but the physical destination is used by all three brokers in the cluster.

Creating JMS Hosts

You can create additional JMS hosts in the following ways:

- Use the Admin Console. Open the Java Message Service component under the relevant configuration, then select the JMS Hosts component. For details, click the Help button in the Admin Console.
- Use the `asadmin create-jms-host` command. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

For machines having more than one host, use the Host field in the Admin Console or the `--mqhost` option of `create-jms-host` to specify the address to which the broker binds.

Checking Whether the JMS Provider Is Running

You can use the `asadmin jms-ping` command to check whether a Sun Java System Message Queue instance is running. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

Creating Physical Destinations

Produced messages are delivered for routing and subsequent delivery to consumers using *physical destinations* in the JMS provider. A physical destination is identified and encapsulated by an administered object (a `Topic` or `Queue` destination resource) that an application component uses to specify the destination of messages it is producing and the source of messages it is consuming.

If a message-driven bean is deployed and the physical destination it listens to doesn't exist, the Application Server automatically creates the physical destination. However, it is good practice to create the physical destination beforehand.

You can create a JMS physical destination in the following ways:

- Use the Admin Console. Open the Resources component, open the JMS Resources component, then select Physical Destinations. For details, click the Help button in the Admin Console.
- Use the `asadmin create-jmsdest` command. This command acts on the default JMS host. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

To purge all messages currently queued at a physical destination, use the `asadmin flush-jmsdest` command. This deletes the messages before they reach any message consumers. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

To create a destination resource, see "Creating JMS Resources: Destinations and Connection Factories" on page 231.

Creating JMS Resources: Destinations and Connection Factories

You can create two kinds of JMS resources in the Application Server:

- **Connection Factories** – administered objects that implement the `ConnectionFactory`, `QueueConnectionFactory`, or `TopicConnectionFactory` interfaces.
- **Destination Resources** – administered objects that implement the `Queue` or `Topic` interfaces.

In either case, the steps for creating a JMS resource are the same. You can create a JMS resource in the following ways:

- To create a JMS resource using the Admin Console, open the Resources component, then open the JMS Resources component. Click Connection Factories to create a connection factory, or click Destination Resources to create a queue or topic. For details, click the Help button in the Admin Console.
- A JMS resource is a type of connector. To create a JMS resource using the command line, see "Deploying and Configuring a Stand-Alone Connector Module" on page 189.

Note – All JMS resource properties that used to work with version 7 of the Application Server are supported for backward compatibility.

Restarting the JMS Client After JMS Configuration

When a JMS client accesses a JMS administered object for the first time, the client JVM retrieves the JMS service configuration from the Application Server. Further changes to the configuration are not available to the client JVM until the client is restarted.

JMS Connection Features

The Sun Java System Message Queue software supports the following JMS connection features:

- "Connection Pooling" on page 232
- "Connection Failover" on page 233

Both these features use the `AddressList` configuration, which is populated with the hosts and ports of the JMS hosts defined in the Application Server. The `AddressList` is updated whenever a JMS host configuration changes. The `AddressList` is inherited by any JMS resource when it is created and by any MDB when it is deployed.

Note – In the Sun Java System Message Queue software, the `AddressList` property is called `imqAddressList`.

Connection Pooling

The Application Server pools JMS connections automatically.

To dynamically modify connection pool properties using the Admin Console, go to either the Connection Factories page (see "Creating JMS Resources: Destinations and Connection Factories" on page 231) or the Connector Connection Pools page (see "Deploying and Configuring a Stand-Alone Connector Module" on page 189).

To use the command line, use the `asadmin create-connector-connection-pool` command to manage the pool (see "Deploying and Configuring a Stand-Alone Connector Module" on page 189.

The `addresslist-behavior` JMS service attribute is set to `random` by default. This means that each `ManagedConnection` (physical connection) created from the `ManagedConnectionFactory` selects its primary broker in a random way from the `AddressList`.

The other `addresslist-behavior` alternative is `priority`, which specifies that the first broker in the `AddressList` is selected first. This first broker is the local colocated Message Queue broker. If this broker is unavailable, connection attempts are made to brokers in the order in which they are listed in the `AddressList`.

When a JMS connection pool is created, there is one `ManagedConnectionFactory` instance associated with it. If you configure the `AddressList` as a `ManagedConnectionFactory` property, the `AddressList` configuration in the `ManagedConnectionFactory` takes precedence over the one defined in the Application Server.

Connection Failover

To specify whether the Application Server tries to reconnect to the primary broker if the connection is lost, set the `reconnect-enabled` attribute in the JMS service. To specify the number of retries and the time between retries, set the `reconnect-attempts` and `reconnect-interval-in-seconds` attributes, respectively.

If reconnection is enabled and the primary broker goes down, the Application Server tries to reconnect to another broker in the `AddressList`. The `AddressList` is updated whenever a JMS host configuration changes. The logic for scanning is decided by two JMS service attributes, `addresslist-behavior` and `addresslist-iterations`.

You can override these settings using JMS connection factory settings. For details, see Chapter 3, "Configuring Java Message Service Resources," in *Sun Java System Application Server Platform Edition 9 Administration Guide*.

The Sun Java System Message Queue software transparently transfers the load to another broker when the failover occurs. JMS semantics are maintained during failover.

Transactions and Non-Persistent Messages

During transaction recovery, non-persistent messages might be lost. If the broker fails between the transaction manager's prepare and commit operations, any non-persistent message in the transaction is lost and cannot be delivered. A message that is not saved to a persistent store is not available for transaction recovery.

Authentication With ConnectionFactory

If your web, EJB, or client module has `res-auth` set to `Container`, but you use the `ConnectionFactory.createConnection("user","password")` method to get a connection, the Application Server searches the container for authentication information before using the supplied user and password. Version 7 of the Application Server threw an exception in this situation.

Message Queue varhome Directory

The Sun Java System Message Queue software uses a default directory for storing data such as persistent messages and its log file. This directory is called `varhome`. The Application Server uses *domain-dir*`/imq` as the `varhome` directory. Thus, for the default Application Server domain, Message Queue data is stored in the following location:

install-dir`/domains/domain1/imq/var/instances/imqbroker`

Version 7 of the Application Server stored this data in the following location:

install-dir`/imq/var/instances/domain1_server`

When executing Message Queue scripts such as *install-dir*`/imq/bin/imqusermgr`, use the `-varhome` option. For example:

```
imqusermgr -varhome $AS_INSTALL/domains/domain1/imq add -u testuser
-p testpassword
```

Delivering SOAP Messages Using the JMS API

Web service clients use the Simple Object Access Protocol (SOAP) to communicate with web services. SOAP uses a combination of XML-based data structuring and Hyper Text Transfer Protocol (HTTP) to define a standardized way of invoking methods in objects distributed in diverse operating environments across the Internet.

For more information about SOAP, see the Apache SOAP web site at `http://xml.apache.org/soap/index.html`.

You can take advantage of the JMS provider's reliable messaging when delivering SOAP messages. You can convert a SOAP message into a JMS message, send the JMS message, then convert the JMS message back into a SOAP message. The following sections explain how to do these conversions:

- "To Send SOAP Messages Using the JMS API" on page 235
- "To Receive SOAP Messages Using the JMS API" on page 236

▼ To Send SOAP Messages Using the JMS API

1 Import the `MessageTransformer` library.

```
import com.sun.messaging.xml.MessageTransformer;
```

This is the utility whose methods you use to convert SOAP messages to JMS messages and the reverse. You can then send a JMS message containing a SOAP payload as if it were a normal JMS message.

2 Initialize the `TopicConnectionFactory`, `TopicConnection`, `TopicSession`, and publisher.

```
tcf = new TopicConnectionFactory();
tc = tcf.createTopicConnection();
session = tc.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);
topic = session.createTopic(topicName);
publisher = session.createPublisher(topic);
```

3 Construct a SOAP message using the SOAP with Attachments API for Java (SAAJ).

```
/*construct a default soap MessageFactory */
MessageFactory mf = MessageFactory.newInstance();
* Create a SOAP message object.*/
SOAPMessage soapMessage = mf.createMessage();
/** Get SOAP part.*/
SOAPPart soapPart = soapMessage.getSOAPPart();
/* Get SOAP envelope. */
SOAPEnvelope soapEnvelope = soapPart.getEnvelope();
/* Get SOAP body.*/
SOAPBody soapBody = soapEnvelope.getBody();
/* Create a name object. with name space */
/* http://www.sun.com/imq. */
Name name = soapEnvelope.createName("HelloWorld", "hw",
 "http://www.sun.com/imq");
* Add child element with the above name. */
SOAPElement element = soapBody.addChildElement(name)
/* Add another child element.*/
element.addTextNode( "Welcome to Sun Java System Web Services." );
/* Create an atachment with activation API.*/
URL url = new URL ("http://java.sun.com/webservices/");
DataHandler dh = new DataHandler (url);
AttachmentPart ap = soapMessage.createAttachmentPart(dh);
/*set content type/ID. */
ap.setContentType("text/html");
ap.setContentId("cid-001");
/**  add the attachment to the SOAP message.*/
soapMessage.addAttachmentPart(ap);
soapMessage.saveChanges();
```

4 **Convert the SOAP message to a JMS message by calling the** `MessageTransformer.SOAPMessageintoJMSMessage()` **method.**

```
Message m = MessageTransformer.SOAPMessageIntoJMSMessage (soapMessage,
session );
```

5 **Publish the JMS message.**

```
publisher.publish(m);
```

6 **Close the JMS connection.**

```
tc.close();
```

▼ To Receive SOAP Messages Using the JMS API

1 **Import the** `MessageTransformer` **library.**

```
import com.sun.messaging.xml.MessageTransformer;
```

This is the utility whose methods you use to convert SOAP messages to JMS messages and the reverse. The JMS message containing the SOAP payload is received as if it were a normal JMS message.

2 **Initialize the** `TopicConnectionFactory`, `TopicConnection`, `TopicSession`, `TopicSubscriber`, **and** `Topic`.

```
messageFactory = MessageFactory.newInstance();
tcf = new com.sun.messaging.TopicConnectionFactory();
tc = tcf.createTopicConnection();
session = tc.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
topic = session.createTopic(topicName);
subscriber = session.createSubscriber(topic);
subscriber.setMessageListener(this);
tc.start();
```

3 **Use the** `OnMessage` **method to receive the message. Use the** `SOAPMessageFromJMSMessage` **method to convert the JMS message to a SOAP message.**

```
public void onMessage (Message message) {
SOAPMessage soapMessage =
 MessageTransformer.SOAPMessageFromJMSMessage( message,
 messageFactory ); }
```

4 **Retrieve the content of the SOAP message.**

◆ ◆ ◆ CHAPTER 19

Using the JavaMail API

This chapter describes how to use the JavaMail™ API, which provides a set of abstract classes defining objects that comprise a mail system.

This chapter contains the following sections:

Introducing JavaMail

The JavaMail API defines classes such as `Message`, `Store`, and `Transport`. The API can be extended and can be subclassed to provide new protocols and to add functionality when necessary. In addition, the API provides concrete subclasses of the abstract classes. These subclasses, including `MimeMessage` and `MimeBodyPart`, implement widely used Internet mail protocols and conform to the RFC822 and RFC2045 specifications. The JavaMail API includes support for the IMAP4, POP3, and SMTP protocols.

The JavaMail architectural components are as follows:

- The *abstract layer* declares classes, interfaces, and abstract methods intended to support mail handling functions that all mail systems support.
- The *internet implementation layer* implements part of the abstract layer using the RFC822 and MIME internet standards.
- JavaMail uses the *JavaBeans Activation Framework* (JAF) to encapsulate message data and to handle commands intended to interact with that data.

For more information, see Chapter 4, "Configuring JavaMail Resources," in *Sun Java System Application Server Platform Edition 9 Administration Guide* and the JavaMail specification at `http://java.sun.com/products/javamail/`.

Creating a JavaMail Session

You can create a JavaMail session in the following ways:

- In the Admin Console, open the Resources component and select JavaMail Sessions. For details, click the Help button in the Admin Console.
- Use the `asadmin create-javamail-resource` command. For details, see the *Sun Java System Application Server Platform Edition 9 Reference Manual*.

JavaMail Session Properties

You can set properties for a JavaMail `Session` object. Every property name must start with a `mail-` prefix. The Application Server changes the dash (`-`) character to a period (`.`) in the name of the property and saves the property to the `MailConfiguration` and JavaMail `Session` objects. If the name of the property doesn't start with `mail-`, the property is ignored.

For example, if you want to define the property `mail.from` in a JavaMail `Session` object, first define the property as follows:

- Name – `mail-from`
- Value – `john.doe@sun.com`

Looking Up a JavaMail Session

The standard Java Naming and Directory Interface (JNDI) subcontext for JavaMail sessions is `java:comp/env/mail`.

Registering JavaMail sessions in the `mail` naming subcontext of a JNDI namespace, or in one of its child subcontexts, is standard. The JNDI namespace is hierarchical, like a file system's directory structure, so it is easy to find and nest references. A JavaMail session is bound to a logical JNDI name. The name identifies a subcontext, `mail`, of the root context, and a logical name. To change the JavaMail session, you can change its entry in the JNDI namespace without having to modify the application.

The resource lookup in the application code looks like this:

```
InitialContext ic = new InitialContext();
String snName = "java:comp/env/mail/MyMailSession";
Session session = (Session)ic.lookup(snName);
```

For more information about the JNDI API, see Chapter 17.

Sending and Reading Messages Using JavaMail

The following sections describe how to send and read messages using the JavaMail API:

▼ To Send a Message Using JavaMail

1 Import the packages that you need.

```
import java.util.*;
import javax.activation.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.naming.*;
```

2 Look up the JavaMail session.

```
InitialContext ic = new InitialContext();
String snName = "java:comp/env/mail/MyMailSession";
Session session = (Session)ic.lookup(snName);
```

For more information, see "Looking Up a JavaMail Session" on page 238.

3 Override the JavaMail session properties if necessary.

For example:

```
Properties props = session.getProperties();
props.put("mail.from", "user2@mailserver.com");
```

4 Create a `MimeMessage`.

The `msgRecipient`, `msgSubject`, and `msgTxt` variables in the following example contain input from the user:

```
Message msg = new MimeMessage(session);
msg.setSubject(msgSubject);
msg.setSentDate(new Date());
msg.setFrom();
msg.setRecipients(Message.RecipientType.TO,
   InternetAddress.parse(msgRecipient, false));
msg.setText(msgTxt);
```

5 Send the message.

```
Transport.send(msg);
```

▼ To Read a Message Using JavaMail

1 Import the packages that you need.

```
import java.util.*;
import javax.activation.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.naming.*;
```

2 Look up the JavaMail session.

```
InitialContext ic = new InitialContext();
String snName = "java:comp/env/mail/MyMailSession";
Session session = (javax.mail.Session)ic.lookup(snName);
```

For more information, see "Looking Up a JavaMail Session" on page 238.

3 Override the JavaMail session properties if necessary.

For example:

```
Properties props = session.getProperties();
props.put("mail.from", "user2@mailserver.com");
```

4 Get a `Store` object from the `Session`, then connect to the mail server using the Store object's `connect()` method.

You must supply a mail server name, a mail user name, and a password.

```
Store store = session.getStore();
store.connect("MailServer", "MailUser", "secret");
```

5 Get the INBOX folder.

```
Folder folder = store.getFolder("INBOX");
```

6 It is efficient to read the `Message` objects (which represent messages on the server) into an array.

```
Message[] messages = folder.getMessages();
```

◆ ◆ ◆ CHAPTER 20

Using the Application Server Management Extensions

Sun Java System Application Server uses Application Server Management eXtensions (AMX) (`http://glassfish.dev.java.net/javaee5/amx/index.html`) for management and monitoring purposes. AMX technology exposes managed resources for remote management as the Java Management eXtensions (JMX™) API.

The Application Server incorporates the JMX 1.2 Reference Implementation (`http://java.sun.com/products/JavaManagement/index.jsp`), which was developed by the Java Community Process as Java Specification Request (JSR) 3 (`http://jcp.org/en/jsr/detail?id=3`), and the JMX Remote API 1.0 Reference Implementation , which is JSR 160 (`http://jcp.org/en/jsr/detail?id=160`).

This chapter assumes some familiarity with the JMX technology, but the AMX interfaces can be used for the most part without understanding JMX. For more information about JMX, see the JMX specifications and Reference Implementations (`http://java.sun.com/products/JavaManagement/download.html`).

For information about creating custom MBeans, see Chapter 14.

This chapter contains the following topics:

About AMX

AMX is an API that exposes all of the Application Server configuration, monitoring and JSR 77 MBeans as easy-to-use client-side dynamic proxies implementing the AMX interfaces. To understand the design and implementation of the AMX API, you can get started with this white paper (`http://glassfish.dev.java.net/nonav/javaee5/amx/amx.html`).

Complete API documentation for AMX is provided in the Application Server package (`http://glassfish.dev.java.net/nonav/javaee5/amx/javadoc/index.html`).

`com.sun.appserv.management`

The code samples in this section are taken from the package:

`com.sun.appserv.management.sample`

The Application Server is based around the concept of *administration domains*. Each domain consists of one or more *managed resources*. A managed resource can be an Application Server or a manageable entity within a server. A managed resource is of a particular type, and each resource type exposes a set of attributes and administrative operations that change the resource's state.

Managed resources are exposed as JMX *management beans*, or *MBeans*. While the MBeans can be accessed using standard JMX APIs (for example, `MBeanServerConnection`), most users find the use of the AMX client-side dynamic proxies much more convenient.

Virtually all components of the Application Server are visible for monitoring and management through AMX. You can use third-party tools to perform all common administrative tasks programmatically, based on the JMX and JMX Remote API standards.

The AMX API consists of a set of interfaces. The interfaces are implemented by client-side dynamic proxies (`http://glassfish.dev.java.net/nonav/javaee5/amx/amx.html#AMXDynamicClientProxy`), each of which is associated with a server-side MBean in the Domain Administration Server (DAS). AMX provides routines to obtain proxies for MBeans, starting with the `DomainRoot` interface (see `http://glassfish.dev.java.net/nonav/javaee5/amx/javadoc/com/sun/appserv/management/DomainRoot.html`).

Note – The term AMX interface in the context of this document should be understood as synonymous with a client-side dynamic proxy implementing that interface.

You can navigate generically through the MBean hierarchy using the `com.sun.appserv.management.base.Container` interface (see `http://glassfish.dev.java.net/nonav/javaee5/amx/javadoc/com/sun/appserv/management/base/Container.html`). When using AMX, the interfaces defined are implemented by client-side dynamic proxies, but they also implicitly define the `MBeanInfo` that is made available by the MBean or MBeans corresponding to it. Certain operations defined in the interface might have a different return type or a slightly different name when accessed through the MBean directly. This results from the fact that direct access to JMX requires the use of `ObjectName`, whereas the AMX interfaces use strongly typed proxies implementing the interface(s).

AMX MBeans

All AMX MBeans are represented as interfaces in a subpackage of `com.sun.appserv.management` (see `http://glassfish.dev.java.net/nonav/javaee5/amx/javadoc/com/sun/appserv/management/package-summary.html`) and are implemented by dynamic proxies on the client-side. Note that client-side means any client, wherever it resides. AMX may be used within the server itself such as in a custom MBean. While you can access AMX MBeans directly through standard JMX APIs, most users find the use of AMX interface (proxy) classes to be most convenient.

An AMX MBean belongs to an Application Server domain. There is exactly one domain per DAS. Thus all MBeans accessible through the DAS belong to a single Application Server administrative domain. All MBeans in an Application Server administrative domain, and hence within the DAS, belong to the JMX domain `amx`. All AMX MBeans can be reached by navigating through the `DomainRoot`.

Note – Any MBeans that do not have the JMX domain `amx` are not part of AMX, and are neither documented nor supported for use by clients.

AMX defines different types of MBean, namely, *configuration* MBeans, *monitoring* MBeans, *utility* MBeans and Java EE management JSR 77 (`http://jcp.org/en/jsr/detail?id=77`) MBeans. These MBeans are logically related in the following ways:

- They all implement the `com.sun.appserv.management.base.AMX` interface (see `http://glassfish.dev.java.net/nonav/javaee5/amx/javadoc/com/sun/appserv/management/base/AMX.html`).
- They all have a `j2eeType` and `name` property within their `ObjectName`. See `com.sun.appserv.management.base.XTypes` (`http://glassfish.dev.java.net/nonav/javaee5/amx/javadoc/com/sun/appserv/management/base/XTypes.html`) and `com.sun.appserv.management.j2ee.J2EETypes` (`http://glassfish.dev.java.net/nonav/javaee5/amx/javadoc/com/sun/appserv/management/j2ee/J2EETypes.html`) for the available values of the `j2eeType` property.
- All MBeans that logically contain other MBeans implement the `com.sun.appserv.management.base.Container` interface.
- JSR 77 MBeans that have a corresponding configuration or monitoring peer expose it using `getConfigPeer()` or `getMonitoringPeer()`. However, there are many configuration and monitoring MBeans that do not correspond to JSR 77 MBeans.

Configuration MBeans

Configuration information for a given Application Server domain is stored in a central repository that is shared by all instances in that domain. The central repository can only be written to by the DAS. However, configuration information in the central repository is made available to administration clients through AMX MBeans.

The configuration MBeans are those that modify the underlying `domain.xml` or related files. Collectively, they form a model representing the configuration and deployment repository and the operations that can be performed on them.

The `Group` Attribute of configuration MBeans, obtained from `getGroup()`, has a value of `com.sun.appserv.management.base.AMX.GROUP_CONFIGURATION`.

Monitoring MBeans

Monitoring MBeans provide transient monitoring information about all the vital components of the Application Server.

The `Group` Attribute of monitoring MBeans, obtained from `getGroup()`, has a value of `com.sun.appserv.management.base.AMX.GROUP_MONITORING`.

Utility MBeans

Utility MBeans provide commonly used services to the Application Server.

The `Group` Attribute of utility MBeans, obtained from `getGroup()`, has a value of `com.sun.appserv.management.base.AMX.GROUP_UTILITY`.

Java EE Management MBeans

The Java EE management MBeans implement, and in some cases extend, the management hierarchy as defined by JSR 77 (`http://jcp.org/en/jsr/detail?id=77`), which specifies the management model for the whole Java EE platform.

The AMX JSR 77 MBeans offer access to configuration and monitoring MBeans using the `getMonitoringPeer()` and `getConfigPeer()` methods.

The `Group` Attribute of Java EE management MBeans, obtained from `getGroup()`, has a value of `com.sun.appserv.management.base.AMX.GROUP_JSR77`.

Other MBeans

MBeans that do not fit into one of the above four categories have the value `com.sun.appserv.management.base.AMX.GROUP_OTHER`. One such example is `com.sun.appserv.management.deploy.DeploymentMgr` (see `http://glassfish.dev.java.net/nonav/javaee5/amx/javadoc/com/sun/appserv/management/deploy/DeploymentMgr.html`).

MBean Notifications

All AMX MBeans that emit Notifications place a `java.util.Map` within the `UserData` field of a standard JMX Notification, which can be obtained using `Notification.getUserData()`. Within the map are one or more items, which vary according to the Notification type. Each Notification type, and the data available within the Notification, is defined in the Javadoc of the MBean (AMX interface) that emits it.

Note that certain standard Notifications, such as `javax.management.AttributeChangeNotification` (see `http://java.sun.com/j2se/1.5.0/docs/api/javax/management/AttributeChangeNotification.html`) do not and cannot follow this behavior.

Access to MBean Attributes

An AMX MBean `Attribute` is accessible in three ways:

- Dotted names using `MonitoringDottedNames` and `ConfigDottedNames`
- Attributes on MBeans using `getAttribute(s)` and `setAttributes(s)` (from the standard JMX API)
- Getters/setters within the MBean's interface class, for example, `getPort()`, `setPort()`, and so on

All dotted names that are accessible through the command line interface are available as Attributes within a single MBean. This includes properties, which are provided as Attributes beginning with the prefix `property.`, for example, `server.property.myproperty`.

Note – Certain attributes that ought to be of a specific type, such as `int`, are declared as `java.lang.String`. This is because the value of the attribute may be a template of a form such as `${HTTP_LISTENER_PORT}`.

Dynamic Client Proxies

Dynamic Client Proxies are an important part of the AMX API, and enhance ease-of-use for the programmer.

JMX MBeans can be used directly by an `MBeanServerConnection` (see `http://java.sun.com/j2se/1.5.0/docs/api/javax/management/MBeanServerConnection.html`) to the server. However, client proxies greatly simplify access to Attributes and operations on MBeans, offering get/set methods and type-safe invocation of operations. Compiling against the AMX interfaces means that compile-time checking is performed, as opposed to server-side runtime checking, when invoked generically through `MBeanServerConnection`.

See the API documentation for the `com.sun.appserv.management` package and its sub-packages for more information about using proxies. The API documentation explains the use of AMX with

proxies. If you are using JMX directly (for example, by using`MBeanServerConnection`), the return type, argument types, and method names might vary as needed for the difference between a strongly-typed proxy interface and generic `MBeanServerConnection/ObjectName` interface.

Connecting to the Domain Administration Server

As stated in "Configuration MBeans" on page 243, the AMX API allows client applications to connect to Application Server instances using the DAS. All AMX connections are established to the DAS only: AMX does not support direct connections to individual server instances. This makes it simple to interact with all servers, clusters, and so on, with a single connection.

Sample code for connecting to the DAS is shown in "Connecting to the DAS" on page 247. The `com.sun.appserv.management.helper.Connect` class (see `http://glassfish.dev.java.net/nonav/javaee5/amx/javadoc/com/sun/appserv/management/helper/Connect.html`) is also available.

Examining AMX Code Samples

An overview of the AMX API and code samples that demonstrate various uses of the AMX API can be found at
`http://glassfish.dev.java.net/nonav/javaee5/amx/samples/javadoc/index.html` and
`http://glassfish.dev.java.net/nonav/javaee5/amx/samples/javadoc/amxsamples/Samples.html`.

The sample implementation is based around the `SampleMain` class. The principal uses of AMX demonstrated by `SampleMain` are the following:

- "Starting an Application Server" on page 248
- "Deploying an Archive" on page 248
- "Displaying the AMX MBean Hierarchy" on page 248
- "Setting Monitoring States" on page 248
- "Accessing AMX MBeans" on page 248
- "Accessing and Displaying the Attributes of an AMX MBean" on page 249
- "Listing AMX MBean Properties" on page 249
- "Performing Queries" on page 249
- "Monitoring Attribute Changes" on page 249
- "Undeploying Modules" on page 249
- "Stopping an Application Server" on page 249

All of these actions are performed by commands that you give to `SampleMain`. Although these commands are executed by `SampleMain`, they are defined as methods of the class `Samples`, which is also found in the `com.sun.appserv.management.sample` package.

The `SampleMain` Class

The `SampleMain` class creates a connection to a DAS, and creates an interactive loop in which you can run the various commands defined in `Samples` that demonstrate different uses of AMX.

Connecting to the DAS

The connection to the DAS is shown in the following code.

EXAMPLE 20–1 Connecting to the DAS

```
[...]
public static AppserverConnectionSource
    connect(
        final String host,
        final int port,
        final String user,
        final String password,
        final TLSParams tlsParams )
        throws IOException
        {
            final String info = "host=" + host + ", port=" + port +
                ", user=" + user + ", password=" + password +
                ", tls=" + (tlsParams != null);

            SampleUtil.println( "Connecting...:" + info );

            final AppserverConnectionSource conn    =
                new AppserverConnectionSource(
                    AppserverConnectionSource.PROTOCOL_RMI,
                    host, port, user, password, tlsParams, null);

            conn.getJMXConnector( false );

            SampleUtil.println( "Connected: " + info );

            return( conn );
        }
[...]
```

A connection to the DAS is obtained using an instance of the `com.sun.appserv.management.client.AppserverConnectionSource` class. For the connection to be established, you must know the name of the host and port number on which the DAS is running, and have the correct user name, password and TLS parameters.

After the connection to the DAS is established, `DomainRoot` is obtained as follows:

```
DomainRoot domainRoot = appserverConnectionSource.getDomainRoot();
```

This `DomainRoot` instance is a client-side dynamic proxy to the MBean `amx:j2eeType=X-DomainRoot,name=amx`.

See the API documentation for `com.sun.appserv.management.client.AppserverConnectionSource` for further details about connecting to the DAS using the `AppserverConnectionSource` class.

However, if you prefer to work with standard JMX, instead of getting `DomainRoot`, you can get the `MBeanServerConnection` or `JMXConnector`, as shown:

```
MBeanServerConnection conn =
appserverConnectionSource.getMBeanServerConnection( false );
JMXConnector jmxConn =
appserverConnectionSource.getJMXConnector( false );
```

Starting an Application Server

The `Samples.startServer` method demonstrates how to start an Application Server.

In this sample AMX implementation, all the tasks are performed by the command `start-server` when you run `SampleMain`. See the `startServer` method to see how this command is implemented. Click the method name to see the source code.

Deploying an Archive

The `Samples.uploadArchive()` and `deploy` methods demonstrate how to upload and deploy a Java EE archive file.

Displaying the AMX MBean Hierarchy

The `Samples.displayHierarchy` method demonstrates how to display the AMX MBean hierarchy.

Setting Monitoring States

The `Samples.setMonitoring` method demonstrates how to set monitoring states.

Accessing AMX MBeans

The `Samples.handleList` method demonstrates how to access many (but not all) configuration elements.

Accessing and Displaying the Attributes of an AMX MBean

The `Samples.displayAllAttributes` method demonstrates how to access and display the attributes of an AMX MBean.

Listing AMX MBean Properties

The `Samples.displayAllProperties` method demonstrates how to list AMX MBean properties.

Performing Queries

The `Samples.demoQuery` method demonstrates how to perform queries.

The `demoQuery()` method uses other methods that are defined by `Samples`, namely `displayWild()`, and `displayJ2EEType()`.

Monitoring Attribute Changes

The `Samples.demoJMXMonitor` method demonstrates how to monitor attribute changes.

Undeploying Modules

The `Samples.undeploy` method demonstrates how to undeploy a module.

Stopping an Application Server

The `Samples.stopServer` method demonstrates how to stop an Application Server. The `stopServer` method simply calls the `Samples.getJ2EEServer` method on a given server instance, and then calls `J2EEServer.stop`.

Running the AMX Samples

The following section lists the steps to run the AMX samples.

▼ To Run the AMX Sample

1 **Ensure that the JAR file `appserv-ext.jar` has been added to your classpath. Some examples also require that `j2ee.jar` be present.**

2 Define a `SampleMain.properties` file, which provides the parameters required by `AppserverConnectionSource` to connect to the DAS.

The file `SampleMain.properties` file should use the following format:

```
connect.host=localhost
connect.port=8686
connect.user=admin
connect.password=admin123
connect.truststore=sample-truststore
connect.truststorePassword=changeme
connect.useTLS=true
```

3 Scripts are provided in the `com.sun.appserv.management.sample` package to run the AMX samples.

Start `SampleMain` by running the appropriate script for your platform:

- `run-samples.sh` on UNIX or Linux platforms
- `run-samples.bat` on Microsoft Windows platforms

4 After `SampleMain` is running, you can interact with it by typing the commands examined above:

- `Enter Command>` **`start-server`** *serverName*
- `Enter Command>` **`list-attributes`**

  You see output like this:

  ```
  --- Attributes for X-DomainRoot=amx ---
  AttributeNames=[...]
  BulkAccessObjectName=amx:j2eeType=X-BulkAccess,name=na
  DomainConfigObjectName=amx:j2eeType=X-DomainConfig,name=na
  MBeanInfoIsInvariant=true
  J2EEDomainObjectName=amx:j2eeType=J2EEDomain,name=amx
  AppserverDomainName=amx
  ObjectName=amx:j2eeType=X-DomainRoot,name=amx
  [...]
  ```

- `Enter Command>` **`show-hierarchy`**

  You see output like this:

  ```
  X-DomainRoot=amx
  X-ConfigDottedNames
  X-SystemInfo
  X-QueryMgr
  X-DeploymentMgr
  X-UploadDownloadMgr
  X-BulkAccess
  X-MonitoringDottedNames
  X-JMXMonitorMgr
  X-Sample
  X-DomainConfig
  ```

```
    X-WebModuleConfig=admingui
    X-WebModuleConfig=adminapp
    X-WebModuleConfig=com_sun_web_ui
    X-JDBCResourceConfig=jdbc/PointBase
    X-JDBCResourceConfig=jdbc/__TimerPool
    X-J2EEApplicationConfig=MEjbApp
    [...]
```

- `Enter Command>` **`list`**

  You see output like this:

```
    --- Top-level ---
    ConfigConfig: [server2-config, default-config, server-config,
    server3-config]
    ServerConfig: [server3, server, server2]
    StandaloneServerConfig: [server3, server, server2]
    ClusteredServerConfig: []
    ClusterConfig: []
    [...]
```

- `Enter Command>` **`list-properties`**

  You see output like this:

```
    Properties for:
    amx:j2eeType=X-JDBCConnectionPoolConfig,name=PointBasePool
    Password=pbPublic
DatabaseName=jdbc:pointbase:server://localhost:9092/sun-appserv-samples
    User=pbPublic
    [...]
```

- `Enter Command>` **`query`**

  You see output like this:

```
    --- Queried for j2eeType=X-*ResourceConfig ---
    j2eeType=X-JDBCResourceConfig,name=jdbc/PointBase
    j2eeType=X-JDBCResourceConfig,name=jdbc/__TimerPool
    [...]
```

- And so on for the other commands:

  `Enter Command>` **`demo-jmx-monitor`**

  `Enter Command>` **`set-monitoring`** *monitoringLevel* (one of `HIGH`, `LOW` or `OFF`)

  `Enter Command>` **`stop-server`** *serverName*

  `Enter Command>` **`quit`**

Index

B

C

D

E

K

L

M

N

O

P

Q

R

S

T

U

V

W

X