Developer's Guide to J2EE Features and Services

Sun™ ONE Application Server

Version 7

817-2177-10
March 2003

Contents

About This Guide

This guide describes how to create and run Java™ 2 Platform, Enterprise Edition (J2EE™ platform) applications that follow the new open Java standards model for the Java™ Database Connectivity (JDBC™), transaction, Java Naming and Directory Interface™ (JNDI), Java™ Message Service (JMS), and JavaMail™ APIs, on the Sun™ Open Net Environment (Sun ONE) Application Server 7.

This preface contains information about the following topics:

- Who Should Use This Guide
- Using the Documentation
- How This Guide Is Organized
- Related Information
- Documentation Conventions
- Product Support

Who Should Use This Guide

The intended audience for this guide is the person who develops, assembles, and deploys J2EE applications in a corporate enterprise.

This guide assumes you are familiar with the following topics:

- J2EE specification
- HTML
- Java programming

- Java APIs as defined in the Java™ Servlet, JavaServer Pages™ (JSP™), Enterprise JavaBeans™ (EJB™), and JDBC specifications
- Structured database query languages such as SQL
- Relational database concepts
- Software development processes, including debugging and source code control

Using the Documentation

The Sun ONE Application Server manuals are available as online files in Portable Document Format (PDF) and Hypertext Markup Language (HTML) formats, at:

`http://docs.sun.com/`

The following table lists tasks and concepts described in the Sun ONE Application Server manuals. The left column lists the tasks and concepts, and the right column lists the corresponding manuals.

Table 1 Sun ONE Application Server Documentation Roadmap

For information about	See the following
Late-breaking information about the software and the documentation	Release Notes
Supported platforms and environments	*Platform Summary*
Introduction to the application server, including new features, evaluation installation information, and architectural overview.	*Getting Started Guide*
Installing Sun ONE Application Server and its various components (sample applications, Administration interface, Sun ONE Message Queue).	*Installation Guide*
Creating and implementing J2EE applications that follow the open Java standards model on the Sun ONE Application Server 7. Includes general information about application design, developer tools, security, assembly, deployment, debugging, and creating lifecycle modules.	*Developer's Guide*

Table 1 Sun ONE Application Server Documentation Roadmap *(Continued)*

For information about	See the following
Creating and implementing J2EE applications that follow the open Java standards model for web applications on the Sun ONE Application Server 7. Discusses web application programming concepts and tasks, and provides sample code, implementation tips, and reference material.	*Developer's Guide to Web Applications*
Creating and implementing J2EE applications that follow the open Java standards model for enterprise beans on the Sun ONE Application Server 7. Discusses EJB programming concepts and tasks, and provides sample code, implementation tips, and reference material.	*Developer's Guide to Enterprise JavaBeans Technology*
Creating clients that access J2EE applications on the Sun ONE Application Server 7	*Developer's Guide to Clients*
Creating web services	*Developer's Guide to Web Services*
J2EE APIs such as JDBC, transactions, JNDI, JMS, and JavaMail	*Developer's Guide to J2EE Features and Services*
Creating custom NSAPI plugins	*Developer's Guide to NSAPI*
Performing the following administration tasks:	*Administrator's Guide*
• Using the Administration interface and the command line interface	
• Configuring server preferences	
• Using administrative domains	
• Using server instances	
• Monitoring and logging server activity	
• Configuring the web server plugin	
• Configuring the Java Messaging Service	
• Using J2EE features	
• Configuring support for CORBA-based clients	
• Configuring database connectivity	
• Configuring transaction management	
• Configuring the web container	
• Deploying applications	
• Managing virtual servers	

Table 1 Sun ONE Application Server Documentation Roadmap *(Continued)*

For information about	See the following
Editing server configuration files	*Administrator's Configuration File Reference*
Configuring and administering security for the Sun ONE Application Server 7 operational environment. Includes information on general security, certificates, and SSL/TLS encryption. HTTP server-based security is also addressed.	*Administrator's Guide to Security*
Configuring and administering service provider implementation for J2EE™ Connector Architecture (CA) connectors for the Sun ONE Application Server 7. Includes information about the Administration Tool, DTDs and provides sample XML files.	*J2EE CA Service Provider Implementation Administrator's Guide*
Migrating your applications to the new Sun ONE Application Server 7 programming model from the Netscape Application Server version 2.1, including a sample migration of an Online Bank application provided with Sun ONE Application Server	*Migration Guide*
Using the Sun™ Open Net Environment (Sun ONE) Message Queue software.	The Sun ONE Message Queue documentation at: `http://docs.sun.com/db/prod/s1.s1msgqu#hic`

How This Guide Is Organized

This guide provides a Sun ONE Application Server environment overview for designing programs, and includes the following topics:

- Chapter 1, "Overview of J2EE Features and Services"

  This chapter summarizes how to use the JDBC, transaction, JNDI, JMS, and JavaMail APIs with the Sun ONE Application Server.

- Chapter 2, "Using the JDBC™ API for Database Access"

  This chapter describes in detail how to use the JDBC API with the Sun ONE Application Server.

- Chapter 3, "Using the Transaction Service"

This chapter describes in detail how to use transactions with the Sun ONE Application Server.

- Chapter 4, “Using the Java Naming and Directory Interface™”

  This chapter describes in detail how to use the JNDI API with the Sun ONE Application Server.

- Chapter 5, “Using the Java™ Message Service”

  This chapter describes in detail how to use the JMS API with the Sun ONE Application Server.

- Chapter 6, “Using the JavaMail™ API”

  This chapter describes in detail how to use the JavaMail API with the Sun ONE Application Server.

Related Information

You can find a directory of URLs for the official specifications at *install_dir*`/docs/index.htm`. Additionally, the following resources may be helpful:

General J2EE Information:

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

Programming with the JDBC API:

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

JMS Programming Concepts:

Java Message Service by Richard Monson-Haefel, David A. Chappell, and Mike Loukides, O’Reilly Publishing

Documentation Conventions

This section describes the types of conventions used throughout this guide:

General Conventions

The following general conventions are used in this guide:

- **File and directory paths** are given in UNIX® format (with forward slashes separating directory names). For Windows versions, the directory paths are the same, except that backslashes are used to separate directories.
- **URLs** are given in the format:

  http://*server.domain*/*path*/*file*.html

  In these URLs, *server* is the server name where applications are run; *domain* is your Internet domain name; *path* is the server's directory structure; and *file* is an individual filename. Italic items in URLs are placeholders.
- **Font conventions** include:
  - The `monospace` font is used for sample code and code listings, API and language elements (such as function names and class names), file names, pathnames, directory names, and HTML tags.
  - *Italic* type is used for code variables.
  - *Italic* type is also used for book titles, emphasis, variables and placeholders, and words used in the literal sense.
  - **Bold** type is used as either a paragraph lead-in or to indicate words used in the literal sense.
- **Installation root directories** for most platforms are indicated by *install_dir* in this document. Exceptions are noted in "Conventions Referring to Directories" on page 13.

  By default, the location of *install_dir* on **most** platforms is:
  - Solaris™ 8 non-package-based Evaluation installations:

    user's home directory/sun/appserver7
  - Solaris unbundled, non-evaluation installations:

    `/opt/SUNWappserver7`
  - Windows, all installations:

```
C:\Sun\AppServer7
```

For the platforms listed above, *default_config_dir* and *install_config_dir* are identical to *install_dir*. See “Conventions Referring to Directories” on page 13 for exceptions and additional information.

- **Instance root directories** are indicated by *instance_dir* in this document, which is an abbreviation for the following:

  default_config_dir`/domains/`*domain*`/`*instance*

- **UNIX-specific descriptions** throughout this manual apply to the Linux operating system as well, except where Linux is specifically mentioned.

NOTE Forte for Java 4.0 has been renamed to Sun ONE Studio 4 throughout this manual.

Conventions Referring to Directories

By default, when using the Solaris 8 and 9 package-based installation and the Solaris 9 bundled installation, the application server files are spread across several root directories. These directories are described in this section.

- **For Solaris 9 bundled installations**, this guide uses the following document conventions to correspond to the various default installation directories provided:
  - *install_dir* refers to `/usr/appserver/`, which contains the static portion of the installation image. All utilities, executables, and libraries that make up the application server reside in this location.
  - *default_config_dir* refers to `/var/appserver/domains`, which is the default location for any domains that are created.
  - *install_config_dir* refers to `/etc/appserver/config`, which contains installation-wide configuration information such as licenses and the master list of administrative domains configured for this installation.
- **For Solaris 8 and 9 package-based, non-evaluation, unbundled installations**, this guide uses the following document conventions to correspond to the various default installation directories provided:
  - *install_dir* refers to `/opt/SUNWappserver7`, which contains the static portion of the installation image. All utilities, executables, and libraries that make up the application server reside in this location.

- *default_config_dir* refers to `/var/opt/SUNWappserver7/domains`which is the default location for any domains that are created.
- *install_config_dir* refers to `/etc/opt/SUNWappserver7/config`, which contains installation-wide configuration information such as licenses and the master list of administrative domains configured for this installation.

Product Support

If you have problems with your system, contact customer support using one of the following mechanisms:

- The online support web site at:

  `http://www.sun.com/supportraining/`

- The telephone dispatch number associated with your maintenance contract

Please have the following information available prior to contacting support. This helps to ensure that our support staff can best assist you in resolving problems:

- Description of the problem, including the situation where the problem occurs and its impact on your operation
- Machine type, operating system version, and product version, including any patches and other software that might be affecting the problem
- Detailed steps on the methods you have used to reproduce the problem
- Any error logs or core dumps

Chapter 1

Overview of J2EE Features and Services

The Java™ 2 Platform, Enterprise Edition (or J2EE™ Platform) includes features and services that are available as resources to all J2EE applications and modules. The Sun™ Open Net Environment (Sun ONE) Application Server 7, a J2EE 1.3 compliant server, provides access to these resources. This guide describes the following features:

- Java™ Database Connectivity (JDBC™) API
- Transaction Service
- Java Naming and Directory Interface™ (JNDI) API
- Java™ Message Service (JMS) API
- JavaMail™ API

Java™ Database Connectivity (JDBC™) API

The standard way to connect to a database from a J2EE application or module is through a JDBC driver. Sun ONE Application Server supports the core JDBC 3.0 API and the JDBC 2.0 extensions and works with a wide range of JDBC Compliant™ drivers. A JDBC resource associates a JDBC driver and database to a JNDI name that applications and modules can reference.

For information about the JDBC API in the Sun ONE Application Server, see Chapter 2, “Using the JDBC™ API for Database Access.”

Transaction Service

The purpose of a transaction is to ensure that data is updated in an all-or-nothing fashion in order to preserve data integrity. The transaction service provides transactional resource managers for the JDBC API, the JMS API, and resource adapters (connector modules). In the Sun ONE Application Server, you can configure transactions and reference them using the JNDI API.

For information about transactions in Sun ONE Application Server, see Chapter 3, “Using the Transaction Service.”

Java Naming and Directory Interface™ (JNDI) API

The JNDI API allows application components and clients to look up distributed resources, services, and EJB™ components. The J2EE resources described in this guide are made available through the JNDI API. External JNDI resources and custom resources are also configurable in the Sun ONE Application Server.

For information about the JNDI API in the Sun ONE Application Server, see Chapter 4, “Using the Java Naming and Directory Interface™.”

Java™ Message Service (JMS) API

The JMS API provides a common way for J2EE applications and modules to create, send, receive, and read messages in a distributed environment. The fully integrated JMS provider for Sun ONE Application Server is the Sun™ Open Net Environment (Sun ONE) Message Queue software. JMS queues, topics, and message destinations are made available through the JNDI API.

For information about the JMS API in the Sun ONE Application Server, see Chapter 5, “Using the Java™ Message Service.”

JavaMail™ API

The JavaMail API allows J2EE applications to create, send, receive, and read mail messages. The JavaMail API includes support for the IMAP4, POP3, and SMTP mail protocols. JavaMail sessions are made available through the JNDI API.

For information about the JavaMail API in the Sun ONE Application Server, see Chapter 6, “Using the JavaMail™ API.”

Chapter 2

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™ ONE Application Server. This chapter also provides high level JDBC implementation instructions for servlets and EJB™ components using the Sun ONE Application Server. The Sun ONE Application Server supports the core JDBC 3.0 API and the JDBC 2.0 extensions.

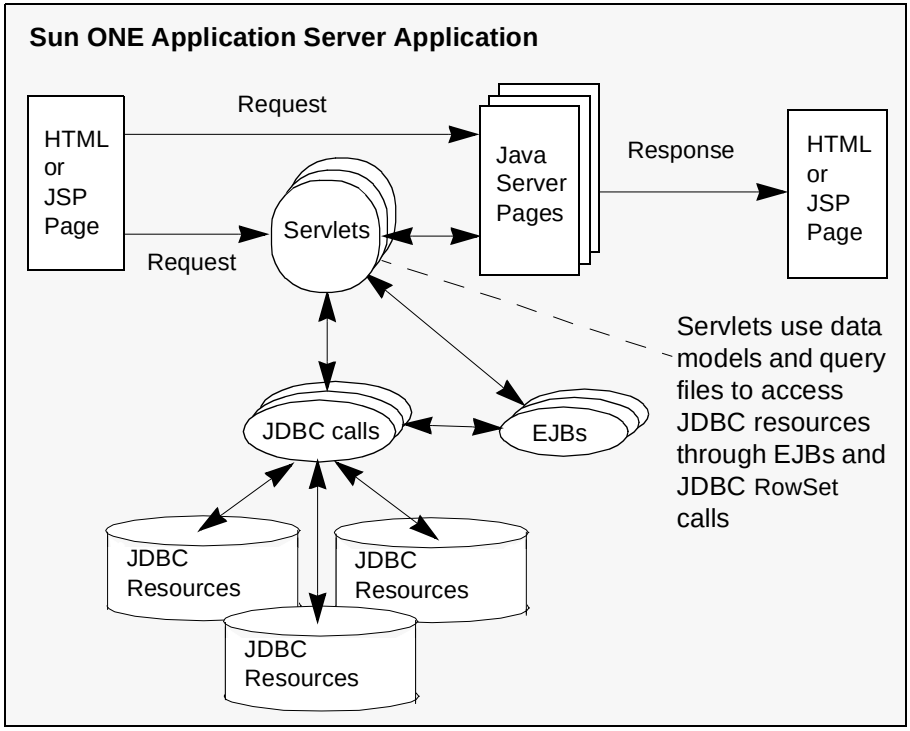
This chapter contains the following sections:

- Introducing the JDBC API
- General Steps for Creating a JDBC Resource
- Configurations for Specific JDBC Drivers
- Creating Applications That Use the JDBC API
- Using the JDBC API in Application Layers
- Sample Applications

Introducing the JDBC API

From a programming perspective, the JDBC API is a set of Java classes and methods that allow embedding of database calls in server applications. More specifically, the JDBC Specification is a set of interfaces that every JDBC driver vendor must implement. A driver processes the JDBC statements in your application and routes the SQL arguments they contain to your database engines. The Sun ONE Application Server supports a variety of JDBC drivers, which support a variety of Enterprise Information Systems (EIS) databases.

The following figure illustrates how application components use the JDBC API to interact with databases.



The JDBC API lets you write high-level, easy-to-use programs that operate seamlessly with and across many different databases without requiring you to know most of the low-level database implementation details.

For explanations of two-tier and three-tier database access models, see the *Sun ONE Application Server Administrator's Guide*.

The JDBC specifications are available here:

`http://java.sun.com/products/jdbc/download.html`

A useful JDBC tutorial is located here:

`http://java.sun.com/docs/books/tutorial/jdbc/index.html`

The rest of this section includes these topics:

- Supported Functionality

- Understanding Database Limitations

Supported Functionality

The JDBC specification is a broad, database-vendor-independent set of guidelines. The guidelines encompass the broadest database functionality range possible in a simple framework. At a minimum, the JDBC API assumes the database supports the SQL-3 database access language. Sun ONE Application Server supports these parts of the JDBC specification:

- The JDBC 3.0 **core database access and functionality** that a server vendor must implement to be JDBC compliant is supported. The Sun ONE Application Server fully meets the compliance standard. From a database vendor's perspective, the JDBC 3.0 API describes a database access model that permits full access to the standard SQL-3 language, the standard language portions each vendor supports, and the language extensions each vendor implements.
- The JDBC 2.0 **Standard Extension API**, which describes advanced features, many of which offer improved database performance, is supported.

NOTE Sun ONE Application Server does not support connection pooling or transactions for an application's database access if it does not use standard J2EE™ `DataSource` objects.

Understanding Database Limitations

When using the JDBC API in your server applications, you may encounter situations where the results are not what you desire or expect. You may think the problem lies in the JDBC API or driver implementation. However, the vast majority of these problems are limitations in your database engine.

Because the JDBC API covers the broadest possible database support, it enables you to attempt operations not every database supports. For example, most database vendors support most of the SQL-3 language, but no vendor provides fully unqualified support for all of the SQL-3 standard. Most vendors built SQL-3 support on top of their existing proprietary relational database management systems, and either those proprietary systems offer features not in SQL-3 or SQL-3 offers features not available in those systems. Most vendors have added non standard SQL-3 extensions to their SQL implementation to support their proprietary features. The JDBC API provides ways to access vendor-specific features, but these features may not be available for all databases you use.

Some JDBC access problems can result if you attempt to access JDBC features that are either partially supported or not supported by the JDBC driver. Check the JDBC driver documentation for details about which JDBC features are supported.

General Steps for Creating a JDBC Resource

To prepare a JDBC resource for use in J2EE applications deployed to the Sun ONE Application Server, perform the following tasks:

- Integrating the JDBC Driver
- Creating a Connection Pool
- Creating a JDBC Resource

For information about how to configure some specific JDBC drivers, see "Configurations for Specific JDBC Drivers" on page 28.

Integrating the JDBC Driver

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

- Supported Database Drivers
- Making the JDBC Driver JAR Files Accessible

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 Sun ONE Application Server, see the *Sun ONE Application Server 7 Platform Summary*.

For configurations of certified drivers, see "Configurations for Specific JDBC Drivers" on page 28.

NOTE Because the drivers and databases supported by the Sun ONE 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 Sun ONE Application Server instance, you can do either of the following:

- Make the driver's class files accessible to the Common Classloader. Copy the JAR and ZIP files into the *instance_dir*`/lib` directory or copy the `.class` files into the *instance_dir*`/lib/classes` directory, then restart the server.
- Make the driver's class files accessible to the System Classloader. Go to the server instance page in the Administration interface, click the JVM Settings tab, click the Path Settings option, edit the Classpath Suffix field, click Save, then restart the server.

Using either classloader makes classes accessible to any application or module across the server instance. For more information about Sun ONE Application Server classloaders, see the *Sun ONE Application Server Developer's Guide.*

Creating a Connection Pool

When you create a JDBC connection pool in the Sun ONE Application Server, you can define many of the characteristics of your database connections.

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

- Using the Administration Interface
- Using The Command Line Interface

The "Using The Administration Interface" section describes each connection pool setting. The "Using The Command Line Interface" section merely lists syntax and default values.

For additional information about connection pools, including connection pool monitoring, see the *Sun ONE Application Server Administrator's Guide*.

Using the Administration Interface

To create a JDBC connection pool using the Administration interface, perform the following tasks:

1. Open the JDBC component under your server instance.
2. Click Connection Pools.
3. Click the New button.
4. Enter the following information:

- Name (required) - Enter a name (or ID) for the connection pool.
- Database Vendor (required) - Select the database driver vendor from the list. You must select a JDBC driver that you have integrated as described in "Integrating the JDBC Driver" on page 20. You can select Other if your database driver is not listed.

5. If you want to enable global transactions, check the Global Transaction Support box.

   If you check this box, the Datasource Classname value you enter later must implement the `java.sql.XADataSource` interface.

6. Click the Next button. (You can click the Back button to return to this page.)

   Your Database Vendor selection determines what is displayed when you click the Next button.

7. Enter or edit the Datasource Classname value. This is the vendor-supplied `DataSource` class name.

8. Specify values for any properties your JDBC driver requires. If a property you need is not listed, use the Add button to add it. The following table lists some standard and commonly used properties.

Table 2-1 Common Connection Pool Properties

Property	Description
`User`	Specifies the user name for this connection pool.
`Password`	Specifies the password for this connection pool.
`databaseName`	Specifies the database for this connection pool.
`serverName`	Specifies the database server for this connection pool.
`port`	Specifies the port on which the database server listens for requests.
`networkProtocol`	Specifies the communication protocol.
`roleName`	Specifies the initial SQL role name.
`datasourceName`	Specifies an underlying `XADataSource`, or a `ConnectionPoolDataSource` if connection pooling is done.
`description`	Specifies a text description.
`url`	Specifies the URL for this connection pool. Although this is not a standard property, it is commonly used.

9. You can change the Pool Settings listed in the following table.

Table 2-2 Pool Settings

Setting	Default	Description
Steady Pool Size	`8`	Specifies the initial and minimum number of connections maintained in the pool.
Max Pool Size	`32`	Specifies the maximum number of connections that can be created to satisfy client requests.
Pool Resize Quantity	`2`	Specifies the number of connections to be destroyed if the existing number of connections is above the Steady Pool Size (subject to the Max Pool Size limit). This is enforced periodically at the Idle Timeout interval. An idle connection is one that has not been used for a period specified by Idle Timeout.
Idle Timeout (secs)	`300`	Specifies the minimum time that a connection can remain idle in the free pool. After this amount of time, the pool can close this connection.
Max Wait Time	`60000`	Specifies the amount of time, in milliseconds, that the caller is willing to wait to acquire a connection. If `0`, the caller is blocked indefinitely until a resource is available or an error occurs.

10. You can change the Connection Validation settings listed in the following table. All of these settings are optional.

Table 2-3 Connection Validation Settings

Setting	Default	Description
Connection Validation Required	Unchecked	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.
Validation Method	`auto-commit`	Legal values are as follows: • `auto-commit` (default), which uses `Connection.setAutoCommit()` • `meta-data`, which uses `Connection.getMetaData()` • `table`, which performs a query on the table specified in the Table Name setting

Table 2-3 Connection Validation Settings *(Continued)*

Setting	Default	Description
Table Name	none	Specifies the table name to be used to perform a query to validate a connection. This setting is mandatory if and only if the Validation Method is set to `table`.
Fail All Connections	Unchecked	If checked, closes all connections in the pool if a single validation check fails. Recovery of a minimum number of connections (specified by the Steady Pool Size setting) is attempted. This setting is mandatory if and only if Connection Validation Required is checked. If Connection Validation Required is unchecked, this setting is ignored.

11. You can change the Transaction Isolation settings listed in the following table. Both of these settings are optional.

Table 2-4 Transaction Isolation Settings

Setting	Default	Description
Transaction Isolation	default JDBC driver isolation level	Specifies the transaction isolation level on the pooled database connections. Allowed values are `read-uncommitted`, `read-committed`, `repeatable-read`, or `serializable`. Not all databases support all these values. For more information about these values, see "Using JDBC Transaction Isolation Levels" on page 33. Applications that change the isolation level on a pooled connection programmatically risk polluting the pool, which can lead to errors. See Guarantee Isolation Level for more details.
Guarantee Isolation Level	Checked	Applicable only when the Transaction Isolation level is explicitly set. If checked, every connection obtained from the pool is guaranteed to have the desired isolation level. This may impact performance on some JDBC drivers. You can uncheck this setting if you are certain that the hosted applications do not return connections with altered isolation levels.

12. Click the Finish button.

Using The Command Line Interface

To create a JDBC connection pool using the command line, use the `asadmin create-jdbc-connection-pool` command. The syntax is as follows, with defaults shown for optional parameters that have them:

`asadmin create-jdbc-connection-pool --user` *admin_user* `[--password` *admin_password*`] [--passwordfile` *password_file*`] [--host localhost] [--port 4848] [--secure | -s] [--instance` *instance_name*`] --datasourceclassname` *class_name* `[--restype javax.sql.DataSource] [--steadypoolsize=8] [--maxpoolsize=32] [--maxwait=60000] [--poolresize=2] [--idletimeout=300] [--isolationlevel` *isolation_level*`] [--isisolationguaranteed=true] [--isconnectvalidatereq=false] [--validationmethod=auto-commit] [--validationtable` *table_name*`] [--failconnection=false] [--description` *text*`] [--property (`*name=value*`)[:`*name=value*`]*] *connection_pool_id*

For more information about the parameters specific to `asadmin create-jdbc-connection-pool`, see "Using the Administration Interface" on page 21. For more information about the general `asadmin` parameters (`--user`, `--password`, `--passwordfile`, `--host`, `--port`, and `--secure`), see the *Sun ONE Application Server Administrator's Guide.*

For example:

```
asadmin create-jdbc-connection-pool --user joeuser --password secret
--datasourceclassname oracle.jdbc.pool.OracleDataSource
--failconnection=true --isconnectvalidatereq=true --property
url=jdbc\\:oracle\\:thin\\:@myhost\\:1521\\:V8i:user=staging_lookup
_app:password=staging_lookup_app OraclePoollookup
```

Note that the colon characters (`:`) within property values must be escaped with double backslashes (`\\`) on Solaris™ platforms as shown, because otherwise they are interpreted as property delimiters. On Windows platforms, colon characters (`:`) must be escaped with single backslashes (`\`). For details about using escape characters, see the *Sun ONE Application Server Administrator's Guide.*

To delete a JDBC connection pool, use the following command:

`asadmin delete-jdbc-connection-pool --user` *admin_user* `[--password` *admin_password*`] [--passwordfile` *password_file*`] [--host localhost] [--port 4848] [--secure | -s] [--instance` *instance_name*`]` *connection_pool_id*

For example:

```
asadmin delete-jdbc-connection-pool --user joeuser --password secret
OraclePoollookup
```

To list JDBC connection pools, use the following command:

```
asadmin list-jdbc-connection-pools --user admin_user [--password
admin_password] [--passwordfile password_file] [--host localhost] [--port
4848] [--secure | -s] [--instance instance_name]
```

For example:

```
asadmin list-jdbc-connection-pools --user joeuser --password secret
--instance server1
```

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:

- Using The Administration Interface
- Using The Command Line Interface

The "Using The Administration Interface" section describes each connection pool setting. The "Using The Command Line Interface" section merely lists syntax and default values.

For general information about JDBC resources, see the *Sun ONE Application Server Administrator's Guide*.

Using The Administration Interface

To create a JDBC resource using the Administration interface, perform these tasks:

1. Open the JDBC component under your server instance.
2. Click JDBC Resources.
3. Click the New button.
4. Enter the following information:
    - JNDI Name (required) - Enter the JNDI name that application components must use to access the JDBC resource. For more information, see "Looking Up a JDBC Resource" on page 31.
    - Pool Name (required) - Select from the list the name (or ID) of the connection pool used by this JDBC resource. For more information, see "Creating a Connection Pool" on page 21.
    - Description (optional) - You can enter a text description of the JDBC resource.

5. Check the Data Source Enabled box to enable the JDBC resource.

   If a JDBC resource is disabled, no application component can connect to it, but its configuration remains in the server instance.

6. Click the OK button.

7. Go to the server instance page.

8. Click the General tab.

9. Click the Apply Changes button.

Using The Command Line Interface

To create a JDBC resource using the command line, use the `asadmin create-jdbc-resource` command. The syntax is as follows, with defaults shown for optional parameters that have them:

`asadmin create-jdbc-resource --user` *admin_user* `[--password` *admin_password*`] [--passwordfile` *password_file*`] [--host localhost] [--port 4848] [--secure | -s] [--instance` *instance_name*`] --connectionpoolid` *connection_pool_id* `[--enabled=true] [--description` *text*`] [--property (`*name=value*`)[:`*name=value*`]*]` *jndi_name*

For more information about the parameters specific to `asadmin create-jdbc-resource`, see "Using The Administration Interface" on page 26. For more information about the general `asadmin` parameters (`--user`, `--password`, `--passwordfile`, `--host`, `--port`, and `--secure`), see the *Sun ONE Application Server Administrator's Guide.*

For example:

```
asadmin create-jdbc-resource --user joeuser --password secret
--connectionpoolid OraclePoollookup OracleDSlookup
```

To delete a JDBC resource, use the following command:

`asadmin delete-jdbc-resource --user` *admin_user* `[--password` *admin_password*`] [--passwordfile` *password_file*`] [--host localhost] [--port 4848] [--secure | -s] [--instance` *instance_name*`]` *jndi_name*

For example:

```
asadmin delete-jdbc-resource --user joeuser --password secret
OracleDSlookup
```

To list JDBC resources, use the following command:

```
asadmin list-jdbc-resources --user admin_user [--password
admin_password] [--passwordfile password_file] [--host localhost] [--port
4848] [--secure | -s] [--instance instance_name]
```

For example:

```
asadmin list-jdbc-resources --user joeuser --password secret
--instance server1
```

After you create the JDBC resource, you must reconfigure the server instance using the following command:

```
asadmin reconfig --user user [--password password] [--passwordfile
password_file] [--host localhost] [--port 4848] [--secure |
-s][--discardmanualchanges=false | --keepmanualchanges=false]
instance_name
```

For example:

```
asadmin reconfig --user joeuser --password secret server1
```

Configurations for Specific JDBC Drivers

The following certified JDBC 2.0 drivers have passed the J2EE Compatibility Test Suite (CTS) when tested with Sun ONE Application Server:

- PointBase Type4 Driver
- Data Direct Connect JDBC3.0 / Type4 Driver for Oracle 9.x Databases

For details about how to integrate a JDBC driver and how to use the Administration interface or the command line interface to implement the configuration, see “General Steps for Creating a JDBC Resource” on page 20.

PointBase Type4 Driver

The PointBase 4.2 JDBC driver is included with the Sun ONE Application Server by default, except for the Solaris bundled installation, which does not include PointBase. Therefore, unless you have the Solaris bundled installation, you do not need to integrate this JDBC driver with the Sun ONE Application Server.

Configure the connection pool using the following settings:

- **Name:** You will use this name when you configure the JDBC resource later.
- **Database Vendor:** PointBase 4.2

- **Global Transaction Support, Datasource Classname:** See the following table.

Table 2-5 Datasource Classname Values for the PointBase Type4 Driver

Global Transaction Support	Datasource Classname Value
No (unchecked)	`com.pointbase.jdbc.jdbcDataSource`
Yes (checked)	`com.pointbase.xa.xaDataSource`

- **Properties:**
  - **`User`** - Set as appropriate.
  - **`Password`** - Set as appropriate.
  - **`databaseName`** - Specify the complete database URL.

Configure the JDBC resource using the following settings:

- **JNDI Name:** Beginning the JNDI name with `jdbc/` is recommended.
- **Pool Name:** Select the name of the connection pool you configured.
- **Data Source Enabled:** Check this box.

Data Direct Connect JDBC3.0/ Type4 Driver for Oracle 9.x Databases

NOTE This JDBC driver limits the size of BLOB datatypes to 4 GB.

Configure the connection pool using the following settings:

- **Name:** You will use this name when you configure the JDBC resource later.
- **Database Vendor:** Data Direct 3
- **Global Transaction Support, Datasource Classname:** See the following table.

Table 2-6 Datasource Classname Values for the Data Direct Driver for Oracle 9.x Databases

Global Transaction Support	Datasource Classname Value
No (unchecked)	`com.ddtek.jdbcx.oracle.OracleDataSource`

Table 2-6 Datasource Classname Values for the Data Direct Driver for Oracle 9.x Databases *(Continued)*

Global Transaction Support	Datasource Classname Value
Yes (checked)	`com.ddtek.jdbcx.oracle.OracleDataSource`

- **Properties:**
  - **`serverName`** - Specify the host name or IP address of the database server.
  - **`portNumber`** - Specify the port number of the database server.
  - **`User`** - Set as appropriate.
  - **`Password`** - Set as appropriate.
  - **`SID`** - Set as appropriate.
  - **`xa-driver-does-not-support-non-tx-operations`** - Set to the value `true`. Optional: only needed if Global Transaction Support is checked.

Configure the JDBC resource using the following settings:

- **JNDI Name:** Beginning the JNDI name with `jdbc/` is recommended.
- **Pool Name:** Select the name of the connection pool you configured.
- **Data Source Enabled:** Check this box.

NOTE This JDBC driver limits the size of BLOB datatypes to 4 GB.

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:

- Using Connections
- Using JDBC Transaction Isolation Levels

Using Connections

To use connections, you should be familiar with these topics:

- Looking Up a JDBC Resource

- Pooling Connections
- Sharing Connections
- Opening and Closing Connections

For general information about connections and JDBC URLs, see the *Sun ONE Application Server Administrator's Guide.*

Looking Up a JDBC Resource

The recommended Java Naming and Directory Interface™ (JNDI) subcontext for JDBC resources is `java:comp/env/jdbc`.

The JDBC 3.0 API specifies that all JDBC resources are registered in the `jdbc` naming subcontext of a JNDI namespace, or in one of its child subcontexts. The JNDI namespace is hierarchical, like a file system's directory structure, so it is easy to find and nest references. A JDBC resource is bound to a logical JNDI name. The name identifies a subcontext, `jdbc`, of the root context, and a logical name. In order to change the JDBC resource, you can just change its entry in the JNDI namespace without having to modify the application.

For more information about the JNDI API, see the JDBC 2.0 Standard Extension API and Chapter 4, "Using the Java Naming and Directory Interface™."

The following code example demonstrates how a JDBC resource is looked up and a connection created from it. As illustrated, the string that is looked up is the same as specified in the `res-ref-name` element in the deployment descriptor file.

```
InitialContext ctx = null;
String dsName1 = "java:comp/env/jdbc/HelloDbDs";
DataSource ds1 = null;
Connection conn1 = null;

try {
   ctx = new InitialContext();
   ds1 = (DataSource)ctx.lookup(dsName1);

   UserTransaction tx = ejbContext.getUserTransaction();

   tx.begin();

   Connection conn1 = ds1.getConnection();

   // use conn1 to do some database work -- note that
   // conn1.commit(), conn1.rollback(), and conn1.setAutoCommit()
   // cannot be used here
```

```
    tx.commit();
    conn1.close();
    conn1 = null;

} catch(Exception e) {
    e.printStackTrace(System.out);
}
finally {
    if (conn1 != null) {
        try {
            conn1.close();
        } catch (Exception e) {
            // ignore
        }
    }
}
```

Pooling Connections

Creating and destroying database connections are expensive operations. Connection pooling allows reuse of persistent connections. When an application closes a connection, the connection is returned to the pool.

For details about connection pool settings (maximum number of connections, connection timeout, and so on), see "Creating a Connection Pool" on page 21.

NOTE Connection pooling is an automatic feature of the Sun ONE Application Server. The API is not exposed.

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 are shared only if `res-sharing-scope` is set to `Shareable` in the J2EE deployment descriptor. To turn off connection sharing, set `res-sharing-scope` to `Unshareable`.

Opening and Closing Connections

When you open a JDBC connection using `DataSource.getConnection()`, the Sun ONE Application Server allocates connection resources. You can use the default user name and password defined for your connection pool or you can pass in other values. For details about setting the default user name and password, see "Creating a Connection Pool" on page 21.

When a connection is no longer needed, call `Connection.close()` to free the connection resources. Always reestablish connections before continuing database operations after you call `Connection.close()`.

TIP Using `Connection.close()` in a `finally` block is recommended. Depending on your database vendor, you may have to close statements as well. Connections are not automatically closed; an application must close its connections.

Use `Connection.isClosed()` to test whether the connection is closed. This method returns false if the connection is open, and returns true only after `Connection.close()` is explicitly called.

Some connection behavior depends on whether it is a local or global connection:

- You can manage the transaction context for *local connections* using the `setAutoCommit()`, `commit()`, and `rollback()` methods.
- Transaction management methods such as `setAutoCommit()`, `commit()`, and `rollback()` are not allowed for *global connections*.

Using JDBC Transaction Isolation Levels

For general information about transactions, see Chapter 3, "Using the Transaction Service," and the *Sun ONE Application Server Administrator's Guide*.

Not all database vendors support all transaction isolation levels available in the JDBC API. The Sun ONE Application Server permits specifying any isolation level your database supports, but throws an exception against values your database does not support. The following table defines transaction isolation levels.

Table 2-7 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.

Specify or examine the transaction isolation level for a connection using the `Connection.setTransactionIsolation()` and `Connection.getTransactionIsolation()` methods, respectively. 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 21.

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:

```
java.sql.DatabaseMetaData db;
if (db.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.

Using the JDBC API in Application Layers

The JDBC API is part of the Sun ONE Application Server runtime environment. This means the JDBC API is always available any time you use Java to program an application. In a typical multi-tiered server application, you use the JDBC API to access an EIS database from a client, from the presentation layer, in servlets, and in EJB components.

However, in practice it makes sense—for security and portability reasons—to restrict database accesses to the middle layers of a multi-tiered server application. In the Sun ONE Application Server programming model, this means placing all JDBC calls in servlets or preferably EJB components.

Using the JDBC API in EJB Components

There are two reasons to place JDBC calls in EJB components:

- Placing all JDBC calls inside EJB components makes your application more modular and more portable. Because you use EJB components as building blocks for many applications with little or no changes, you can use an EJB component to maintain a common interface to your EIS database.
- EJB components provide built-in mechanisms for transaction control. Placing JDBC calls in well-designed EJB components frees you from programming explicit transaction control using the JDBC API or `java.transaction.UserTransaction`, which provides low-level transaction support under the JDBC API.

NOTE For container-managed transactions, use a globally available JDBC resource to create a global connection so that the EJB transaction manager controls the transaction.

For more information about transactions in EJB components, see “Transaction Management” on page 40 and the *Sun ONE Application Server Developer’s Guide to Enterprise JavaBeans Technology*.

Using the JDBC API in Servlets

Servlets are at the heart of a Sun ONE Application Server application. They stand between a client interface, such as an HTML or JSP™ page (a page created with the JavaServer Pages™ technology), and the EJB components that do the bulk of an application's work.

The Sun ONE Application Server applications use JDBC calls embedded in EJB components for most database accesses. This is the preferred method for database accesses using the Sun ONE Application Server because it enables you to take advantage of the transaction control built into EJB components and their containers. Servlets, however, can also provide database access through the JDBC API.

In some situations, accessing a database directly from a servlet can offer a speed advantage over accessing a database from EJB components. There is less call overhead, if an application is spread across servers so that EJB components are accessible only through the Java Remote Method Interface (RMI).

If access to a database is from a servlet, use the JDBC 2.0 `RowSet` interface to interact with the database. A row set is a Java object that encapsulates a set of rows that have been retrieved from a database or other tabular data source, such as a spreadsheet. The `RowSet` interface provides JavaBean properties that allow a `RowSet` instance to be configured to connect to a database and retrieve a set of rows.

Sample Applications

JDBC sample applications are in the following directory:

install_dir`/samples/jdbc`

Chapter 3

Using the Transaction Service

The J2EE™ platform provides several abstractions that simplify development of dependable transaction processing for applications. This chapter discusses J2EE transactions and transaction support in the Sun™ ONE Application Server.

This chapter contains the following sections:

- Introducing Transactions
- Configuring the Transaction Service
- Looking Up a Transaction
- Transaction Logging
- Sample Applications

Introducing Transactions

The purpose of a transaction is to ensure that data is updated in an all-or-nothing fashion. For example, a message is either delivered or not delivered. A transaction has these four characteristics.

- Atomicity - A transaction can end in two ways: with a commit or a rollback. When a transaction is committed, modifications made to the data within the transaction boundaries are saved and made permanent. When a transaction is rolled back, all modifications made to the data are undone.
- Consistency - If a transaction fails, data integrity is preserved.
- Isolation - Phases in a transaction cannot be observed by other applications and threads until the transaction is committed or rolled back.
- Durability - Committed changes can survive system failures.

In addition, J2EE transaction semantics do not support nested transactions. You must commit or roll back a transaction before you can begin another one.

For more information about the Java™ Transaction API (JTA) and Java™ Transaction Service (JTS), see the *Sun ONE Application Server Administrator's Guide* and the following sites:

`http://java.sun.com/products/jta/`

`http://java.sun.com/products/jts/`

You may also want to read the chapter on transactions in the J2EE tutorial:

`http://java.sun.com/j2ee/tutorial/1_3-fcs/index.html`

This section covers the following topics:

- Transaction Resource Managers
- Transaction Scope
- Transaction Management
- Transaction Recovery

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. Sun ONE Application Server supports a variety of JDBC™ XA drivers, listed in "Configurations for Specific JDBC Drivers" on page 28. For information about JDBC transaction isolation levels, see "Using JDBC Transaction Isolation Levels" on page 33.
- Java™ Message Service (JMS) Providers - Use of transactions ensures that messages are reliably delivered. Sun ONE Application Server is integrated with Sun ONE Message Queue, a fully capable JMS provider. For more information about transactions and the JMS API, see "Using Transactions for Message Acknowledgement" on page 78.
- 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 J2EE CA connectors, see the *Application Server J2EE CA Service Provider Implementation Administrator's Guide*.

For details about how transaction resource managers, the transaction service, and applications interact, see the *Sun ONE Application Server Administrator's 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 J2EE application.

In Sun ONE 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 `res-type` attribute 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 or XA resource is used, the transaction is aborted.
- No transaction importing or exporting occurs.

If the transaction timeout is greater than zero:

- If the resource involved is non-XA, this resource is wrapped in a Sun ONE XA wrapper, and the global transaction infrastructure is used.
- If the resource involved is XA, it is like any other global transaction.

Transactions that involve more than one resource, or multiple participant processes, are *distributed* or *global* transactions.

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 may 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.

Transaction Management

Sun ONE Application Server supports both types of transaction management:

- Container-Managed Transactions
- Component-Managed Transactions

These sections provide brief summaries. For more information, see the *Sun ONE Application Server Administrator's Guide* and the *Sun ONE Application Server Developer's Guide to Enterprise JavaBeans Technology*.

Container-Managed Transactions

In an enterprise bean with container-managed (or declarative) transactions, the EJB™ container sets the boundaries of the transactions. You can use container-managed transactions with any type of enterprise bean: session, entity, or message-driven. Container-managed transactions simplify development because the enterprise bean code does not explicitly mark the transaction's boundaries. The code does not include statements that begin and end the transaction.

Typically, the container begins a transaction immediately before an enterprise bean method starts. It commits the transaction just before the method exits. Each method can be associated with a single transaction. Nested or multiple transactions are not allowed within a method.

Component-Managed Transactions

There may be times when component-managed (or programmatic) transaction management using the JDBC API or `javax.transaction.UserTransaction` is appropriate for an application. In these cases, you can program the transaction management in the application yourself.

In a component-managed transaction, the code in the session or message-driven bean explicitly marks the boundaries of the transaction. An entity bean cannot have component-managed transactions; it must use container-managed transactions instead. Although beans with container-managed transactions require less coding, they have one limitation: When a method is executing, it can be associated with either a single transaction or no transaction at all. If this limitation will make coding your bean difficult, you should consider using component-managed transactions.

To initiate and perform programmatic transactions, components reference the Sun ONE Application Server's default transaction coordinator as described in "Looking Up a Transaction" on page 44. EJB components can also reference the transaction using the `EJBContext.getUserTransaction()` method.

Transaction Recovery

Transaction recovery is an important aspect of distributed transactions. If a resource becomes unreachable, or if there are unrecoverable errors, the status of the distributed transaction can be in question. Automatic and manual recovery of stranded or incomplete transactions is an important feature in Sun ONE Application Server. To enable automatic transaction recovery, see "Configuring the Transaction Service" on page 41.

Configuring the Transaction Service

You can configure the transaction service in Sun ONE Application Server in the following ways:

- Using the Administration Interface
- Using the Command Line Interface

The "Using The Administration Interface" section describes each connection pool setting. The "Using The Command Line Interface" section merely lists syntax and default values.

This section covers basic configuration. For details about monitoring and other administration topics, see the *Sun ONE Application Server Administrator's Guide.*

Using the Administration Interface

To configure the transaction service using the Administration interface, perform the following tasks:

1. Open the Transaction Service component under your server instance.
2. Edit the following information if desired. All of these settings are optional.
    - Monitoring Enabled - Check this box to enable monitoring of the transaction service. By default, monitoring is disabled.

- Log Level - Controls the type of messages logged by the transaction service to the server log. You can select a specific level, or you can select the default level set in the Log Service. For details, see the description of the Log Service in the *Sun ONE Application Server Administrator's Guide*.
- Recover on Restart - Check this box if you want the server instance to attempt transaction recovery during startup. By default, recovery is not attempted.
- Transaction Timeout - Specifies the amount of time after which the transaction is aborted. If set to `0` (the default), the transaction never times out. This is the application level timeout. To set the `XAResource` timeout, use the `xaresource-txn-timeout` property, described in Step 3.
- Transaction Log Location - Sets the location of the transaction log directory. The directory in which the transaction logs are kept must be writable by whatever user account the server runs as. The default location is *instance_dir*`/logs`.
- Heuristic Decision - During recovery, if the outcome of a transaction cannot be determined from the logs, this property determines the outcome. The default is `rollback`. The other choice is `commit`.
- Keypoint Interval - Specifies the number of transactions between keypoint operations in the log. Keypoint operations reduce the size of the transaction log file by compressing it. A larger value for this attribute (for example, `4096`) results in a larger transaction log file, but fewer keypoint operations and potentially better performance. A smaller value (for example, `100`) results in smaller log files, but slightly reduced performance due to the greater frequency of keypoint operations.

3. To add properties to the transaction service, perform the following tasks:

a. Click the Properties button.

b. Specify names and values for any properties you want to use. If you need another name-value row, use the Add button to add it. The following table lists the transaction service properties for Sun ONE Application Server.

Table 3-1 Transaction Service Properties

Property	Default	Description
`disable-distributed-transaction-logging`	`false`	If `true`, disables transaction logging, which may improve performance. If `false`, the transaction service writes transactional activity into transaction logs so that transactions can be recovered. If Recover on Restart is checked, this property is ignored. Use *only* if performance is more important than transaction recovery.
`xaresource-txn-timeout`	specific to the `XAResource` used	Changes the `XAResource` timeout. In some cases, the `XAResource` default timeout causes transactions to be aborted, so it is desirable to change it. To set the application level timeout, use the Transaction Timeout setting, described in Step 2.

c. Click the Save button.

4. Go to the server instance page.
5. Click the General tab.
6. Click the Apply Changes button.

Using the Command Line Interface

To configure the transaction service using the command line interface, use the `asadmin set` command. The syntax is as follows, with defaults shown for optional parameters that have them:

`asadmin set --user` *admin_user* `[--password` *admin_password*`] [--passwordfile` *password_file*`] [--host localhost] [--port 4848] [--secure | -s]` *attribute_name=value* `[`*attribute_name=value*`] *`

For more information about the general `asadmin` parameters (`--user`, `--password`, `--passwordfile`, `--host`, `--port`, and `--secure`), see the *Sun ONE Application Server Administrator's Guide.*

The *attribute_name* is a hierarchical name that looks like this:

instance`.transaction-service.`*ts_attribute_name*

The *instance* is the application server instance name. The *ts_attribute_name* is the transaction service attribute that needs to be configured. For example:

```
server1.transaction-service.transactionTimeout
```

To view the list of transaction service attribute names that can be configured using the `asadmin set` command, use the `asadmin get` command with a wildcard. The `asadmin get` command has the same syntax as the `asadmin set` command. For example:

```
asadmin get --user joeuser --password secret "server1.transaction-service.*"
```

A list of attribute names for configuring the transaction service of the `server1` application server instance is displayed. The transaction service attributes are as follows:

```
server1.transaction-service.logLevel
server1.transaction-service.automaticTransactionRecovery
server1.transaction-service.heuristicDecision
server1.transaction-service.transactionTimeout
server1.transaction-service.keypointInterval
server1.transaction-service.monitoringEnabled
server1.transaction-service.transactionLogFile
```

Here is an example of running the `asadmin set` command:

```
asadmin set --user joeuser --password secret
server1.transaction-service.transactionTimeout=0
```

The *attribute_name* for a transaction service property is a hierarchical name that looks like this:

instance`.transaction-service.property.`*ts_property_name*

The *instance* is the application server instance name. The *ts_property_name* is the transaction service property that needs to be configured. Here is an example of running the `asadmin set` command to set a transaction service property:

```
asadmin set --user joeuser --password secret
server1.transaction-service.property.xaresource-txn-timeout=30
```

Looking Up a Transaction

Application components obtain reference to the container's default transaction coordinator by doing a Java Naming and Directory Interface™ (JNDI) lookup of `java:comp/UserTransaction`. The returned object implements the `javax.transaction.UserTransaction` interface and can be used in the

application to begin, commit, rollback, and query the status of programmatic transactions. For more information about the JNDI API, see the JDBC 2.0 Standard Extension API and Chapter 4, "Using the Java Naming and Directory Interface™." The transaction lookup in the application code looks like this:

```
InitialContext ic = new InitialContext();
String txName = "java:comp/UserTransaction";
UserTransaction tx = (javax.transaction.UserTransaction)ic.lookup(txName);
tx.begin();
// transacted commands, such as JDBC calls
tx.commit();
```

Transaction Logging

The transaction service writes transactional activity into transaction logs so that transactions can be recovered. You can control transaction logging in these ways:

- Set the location of the transaction log files using the Transaction Log Location setting in the Administration interface or the `transactionLogFile` attribute in the command line interface.
- Turn off transaction logging by setting the `disable-distributed-transaction-logging` property to `true`. Do this *only* if performance is more important than transaction recovery.

For details, see "Configuring the Transaction Service" on page 41.

You can set the level of detail for transaction-related messages in the server log file using the Log Level setting in the Administration interface or the `LogLevel` attribute in the command line interface.

Sample Applications

Sample applications that use transactions are in the following directory:

install_dir`/samples/transactions`

Chapter 4

Using the Java Naming and Directory Interface™

A *naming service* maintains a set of bindings, which relate names to objects. The J2EE™ 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
- Configuring Resources
- Mapping References
- Sample Applications

Accessing the Naming Context

Sun™ ONE Application Server provides a naming environment, or *context*, which is compliant with standard J2EE 1.3 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 J2EE 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:

- Using the InitialContext to Look Up a Named Object
- Naming Environment for J2EE Application Components
- COSNaming Provider for Application Clients
- Naming Environment for Lifecycle Modules

Using the InitialContext to Look Up a Named Object

To look up a resource, first you instantiate an `InitialContext`, then you use the `InitialContext.lookup()` method to look up the resource by its JNDI name. The following example catches `NameNotFoundException` and `NamingException` exceptions that can occur during a lookup:

```
try {
    InitialContext ic = new InitialContext();
    String snName = "java:comp/env/mail/MyMailSession";
    Session session = (javax.mail.Session)ic.lookup(snName);
}
catch (NameNotFoundException e) {
    out("\nJNDI binding was not found");
}
catch (NamingException e) {
    out("\nJNDI binding error");
}
```

Naming Environment for J2EE Application Components

The namespace for objects looked up in a J2EE environment is organized into different subcontexts, with the standard prefix `java:comp/env`.

The following table describes recommended JNDI subcontexts for connection factories in the Sun ONE Application Server.

Table 4-1 Recommended JNDI Subcontexts for Connection Factories

Resource Manager	Connection Factory Type	JNDI Subcontext
JDBC™	`javax.sql.DataSource`	`java:comp/env/jdbc`
Transaction Service	`javax.transaction.UserTransaction`	`java:comp/UserTransaction`
JMS	`javax.jms.TopicConnectionFactory`	`java:comp/env/jms`
	`javax.jms.QueueConnectionFactory`	
JavaMail™	`javax.mail.Session`	`java:comp/env/mail`
URL	`java.net.URL`	`java:comp/env/url`
Connector	`javax.resource.cci.ConnectionFactory`	`java:comp/env/eis`

COSNaming Provider for Application Clients

To support a global JNDI namespace that is accessible to IIOP application clients, Sun ONE Application Server includes a J2EE-based CosNaming provider, which supports the binding of CORBA references (remote EJB references). The `InitialContext` returned to the IIOP clients is a CosNaming provider. A Sun ONE Application Server instance registers the entity beans for the IIOP clients to lookup and bind to.

Objects stored in the CosNaming and local JNDI environments 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 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. For details about lifecycle modules, see the *Sun ONE Application Server Developer's Guide*.

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

J2EE application components can access a variety of resources using the JNDI API, such as:

- JDBC Resources
- User Transaction Handles
- JMS Resources
- JavaMail Sessions
- Persistence Manager Factories
- URL Connection Factories
- J2EE Connector Architecture Connection Factories

In addition, Sun ONE Application Server exposes the following resources in the naming environment. For these resources, full administration details are provided in the following sections:

- External JNDI Resources
- Custom Resources

NOTE Each resource within a server instance must have a unique name. However, two resources in different server instances or different domains may have the same name.

JDBC Resources

For details on how to configure JDBC resources, see “General Steps for Creating a JDBC Resource” on page 20. The recommended JNDI subcontext for JDBC resources is `java:comp/env/jdbc`. For more information, see “Looking Up a JDBC Resource” on page 31.

User Transaction Handles

Application components obtain reference to the container’s default transaction coordinator by doing a JNDI lookup of `java:comp/UserTransaction`. The returned object implements the `javax.transaction.UserTransaction` interface and can be used in the application to begin, commit, rollback, and query the status of transactions. For additional information about transactions, see Chapter 3, “Using the Transaction Service.”

JMS Resources

For details on how to configure JMS resources, see “Administration of the JMS Service” on page 64. The recommended JNDI subcontext for JMS resources is `java:comp/env/jms`. For more information, see “Looking Up Connection Factories” on page 74 and “Looking Up Destinations” on page 75.

JavaMail Sessions

For details on how to configure JavaMail sessions, see “Creating a JavaMail Session” on page 84. The recommended JNDI subcontext for JavaMail sessions is `java:comp/env/mail`. For more information, see “Looking Up a JavaMail Session” on page 87.

Persistence Manager Factories

A persistence manager factory resource for container-managed persistence (CMP) implements the following interface:

```
com.sun.jdo.spi.persistence.support.sqlstore.impl.PersistenceManagerFactory
```

The recommended JNDI subcontext for persistence manager factories is `java:comp/env/jdo`.

For details about how to create and use a persistence manager factory resource, see the *Sun ONE Application Server Administrator's Guide* and the *Sun ONE Application Server Developer's Guide to Enterprise JavaBeans Technology*.

URL Connection Factories

A URL connection factory implements the `java.net.URL` interface. The recommended JNDI subcontext for URL connection factories is `java:comp/env/url`.

URL connection factories do not require any resource to be configured in the Sun ONE Application Server itself. The `jndi-name` element in the relevant deployment descriptor specifies the target URL. For example, the following entry in the `web.xml` file:

```
<resource-ref>
   <res-ref-name>myURL</res-ref-name>
   <res-type>java.net.URL</res-type>
   <res-auth>Container</res-auth>
</resource-ref>
```

maps to the following entry in the `sun-web.xml` file:

```
<resource-ref>
   <res-ref-name>myURL</res-ref-name>
   <jndi-name>http://www.sun.com/index.html</jndi-name>
</resource-ref>
```

J2EE Connector Architecture Connection Factories

A J2EE™ Connector Architecture (CA) connection factory implements the `javax.resource.cci.ConnectionFactory` interface.

The recommended JNDI subcontext for J2EE CA connection factory resources is `java:comp/env/eis`.

For details about how to create and use a J2EE CA connection factory, see the *Sun ONE Application Server J2EE CA Service Provider Implementation Administrator's Guide*.

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:

- Using the Administration Interface
- Using the Command Line Interface

The "Using The Administration Interface" section describes each connection pool setting. The "Using The Command Line Interface" section merely lists syntax and default values.

Using the Administration Interface

To create an external JNDI resource using the Administration interface, perform the following tasks:

1. Open the JNDI component under your server instance.
2. Click External Resources.
3. Click the New button.
4. Enter the following information:
    - JNDI Name (required) - Enter the JNDI name for the resource.
    - Resource Type (required) - Enter the fully qualified type of the resource.
    - JNDI Lookup (required) - Enter the JNDI value to look up in the external repository. For example, for a bean class, your JNDI Lookup might be `cn=mybean`.
    - Factory Class (required) - Enter the fully qualified name of the factory class.
    - Description (optional) - You can enter a text description of the external JNDI resource.
5. Check the External Resource Enabled box to enable the external JNDI resource.

    If an external JNDI resource is disabled, no application component can connect to it, but its configuration remains in the server instance.
6. Click the OK button.

7. To add properties to an external JNDI resource, perform the following tasks:
   a. Go back to the External Resources page.
   b. Click the external JNDI resource you just created.
   c. Click the Properties button.
   d. Specify names and values for any properties you want to use. If you need another name-value row, use the Add button to add it.
   e. Click the OK button.
   f. Click the Save button.
8. Go to the server instance page.
9. Click the General tab.
10. Click the Apply Changes button.

Using the Command Line Interface

To create an external JNDI resource using the command line, use the `asadmin create-jndi-resource` command. The syntax is as follows, with defaults shown for optional parameters that have them:

`asadmin create-jndi-resource --user` *admin_user* `[--password` *admin_password*`] [--passwordfile` *password_file*`] [--host localhost] [--port 4848] [--secure | -s] [--instance` *instance_name*`] --jndilookupname` *lookup_name* `--resourcetype` *resource_type* `--factoryclass` *class_name* `[--enabled=true] [--description` *text*`] [--property (`*name*`=`*value*`)[:`*name*`=`*value*`]*] ` *jndi_name*

For more information about the parameters specific to `asadmin create-jndi-resource`, see "Using the Administration Interface" on page 53. For more information about the general `asadmin` parameters (`--user`, `--password`, `--passwordfile`, `--host`, `--port`, and `--secure`), see the *Sun ONE Application Server Administrator's Guide*.

For example:

```
asadmin create-jndi-resource --user joeuser --password secret
--jndilookupname cn=myBean --resourcetype test.myBean --factoryclass
com.sun.jndi.ldap.LdapCtxFactory test/myBean
```

To delete an external JNDI resource, use the following command:

`asadmin delete-jndi-resource --user` *admin_user* `[--password` *admin_password*`] [--passwordfile` *password_file*`] [--host localhost] [--port 4848] [--secure | -s] [--instance` *instance_name*`]` *jndi_name*

For example:

```
asadmin delete-jndi-resource --user joeuser --password secret
test/myBean
```

To list external JNDI resources, use the following command:

`asadmin list-jndi-resources --user` *admin_user* `[--password` *admin_password*`] [--passwordfile` *password_file*`] [--host localhost] [--port 4848] [--secure | -s] --instance` *instance_name*

For example:

```
asadmin list-jndi-resources --user joeuser --password secret
--instance server1
```

After you create the external JNDI resource, you must reconfigure the server instance using the following command:

`asadmin reconfig --user` *user* `[--password` *password*`] [--passwordfile` *password_file*`] [--host localhost] [--port 4848] [--secure | -s][--discardmanualchanges=false | --keepmanualchanges=false]` *instance_name*

For example:

```
asadmin reconfig --user joeuser --password secret server1
```

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:

- Using the Administration Interface
- Using the Command Line Interface

The “Using The Administration Interface” section describes each connection pool setting. The “Using The Command Line Interface” section merely lists syntax and default values.

Using the Administration Interface

To create a custom resource using the Administration interface, perform the following tasks:

1. Open the JNDI component under your server instance.
2. Click Custom Resources.
3. Click the New button.
4. Enter the following information:
   - JNDI Name (required) - Enter the JNDI name for the resource.
   - Resource Type (required) - Enter the fully qualified type of the resource.
   - Factory Class (required) - Enter the fully qualified name of the factory class.
   - Description (optional) - You can enter a text description of the custom resource.
5. Check the Custom Resource Enabled box to enable the custom resource.

   If a custom resource is disabled, no application component can connect to it, but its configuration remains in the server instance.
6. Click the OK button.
7. To add properties to a custom resource, perform the following tasks:
   a. Go back to the Custom Resources page.
   b. Click the custom resource you just created.
   c. Click the Properties button.
   d. Specify names and values for any properties you want to use. If you need another name-value row, use the Add button to add it.
   e. Click the OK button.
   f. Click the Save button.
8. Go to the server instance page.
9. Click the General tab.
10. Click the Apply Changes button.

Using the Command Line Interface

To create a custom resource using the command line, use the `asadmin create-custom-resource` command. The syntax is as follows, with defaults shown for optional parameters that have them:

```
asadmin create-custom-resource --user admin_user [--password
admin_password] [--passwordfile password_file] [--host localhost] [--port
4848] [--secure | -s] [--instance instance_name] --resourcetype
resource_type --factoryclass class_name [--enabled=true] [--description
text] [--property (name=value)[:name=value]*] jndi_name
```

For more information about the parameters specific to `asadmin create-custom-resource`, see "Using the Administration Interface" on page 53. For more information about the general `asadmin` parameters (`--user`, `--password`, `--passwordfile`, `--host`, `--port`, and `--secure`), see the *Sun ONE Application Server Administrator's Guide.*

For example:

```
asadmin create-custom-resource --user joeuser --password secret
--resourcetype test.MyBean --factoryclass test.MyBeanFactory
test/myBean
```

To delete a custom resource, use the following command:

```
asadmin delete-custom-resource --user admin_user [--password
admin_password] [--passwordfile password_file] [--host localhost] [--port
4848] [--secure | -s] [--instance instance_name] jndi_name
```

For example:

```
asadmin delete-custom-resource --user joeuser --password secret
test/myBean
```

To list custom resources, use the following command:

```
asadmin list-custom-resources --user admin_user [--password
admin_password] [--passwordfile password_file] [--host localhost] [--port
4848] [--secure | -s] --instance instance_name
```

For example:

```
asadmin list-custom-resources --user joeuser --password secret
--instance server1
```

After you create the custom resource, you must reconfigure the server instance using the following command:

```
asadmin reconfig --user user [--password password] [--passwordfile
password_file] [--host localhost] [--port 4848] [--secure |
-s][--discardmanualchanges=false | --keepmanualchanges=false]
instance_name
```

For example:

```
asadmin reconfig --user joeuser --password secret server1
```

Mapping References

The following XML elements map JNDI names configured in the Sun ONE Application Server to resource references in application client, EJB, and web application components:

- `resource-env-ref` - Maps the `resource-env-ref` element in the corresponding J2EE XML file to the absolute JNDI name configured in Sun ONE Application Server.
- `resource-ref` - Maps the `resource-ref` element in the corresponding J2EE XML file to the absolute JNDI name configured in Sun ONE Application Server.
- `ejb-ref` - Maps the `ejb-ref` element in the corresponding J2EE XML file to the absolute JNDI name configured in Sun ONE Application Server.

  JNDI names for EJB components must be unique. For example, appending the application name and the module name to the EJB name would be 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-app.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 the manuals listed in the following table.

Table 4-2 Manuals that Document the Sun ONE Application Server Deployment Descriptors

Deployment Descriptor	Where to Find More Information
`sun-web.xml`	*Sun ONE Application Server Developer's Guide to Web Applications*
`sun-ejb-jar.xml`	*Sun ONE Application Server Developer's Guide to Enterprise JavaBeans Technology*
`sun-application-client.xml`	*Sun ONE Application Server Developer's Guide to Clients*

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-ref` element in the `sun-web-app.xml` deployment descriptor file maps the JNDI name of a resource reference to the the `resource-ref` element in the `web-app.xml` J2EE deployment descriptor file.

The resource lookup in the application code looks like this:

```
InitialContext ic = new InitialContext();
String dsName = "java:comp/env/jdbc/HelloDbDs";
DataSource ds = (javax.sql.DataSource)ic.lookup(dsName);
Connection connection = ds.getConnection();
```

The resource being queried is listed in the `res-ref-name` element of the `web.xml` file as follows:

```
<resource-ref>
   <description>DataSource Reference</description>
   <res-ref-name>jdbc/HelloDbDs</res-ref-name>
   <res-type>javax.sql.DataSource</res-type>
   <res-auth>Container</res-auth>
</resource-ref>
```

The `resource-ref` section in a Sun ONE specific deployment descriptor, for example `sun-web.xml`, maps the `res-ref-name` (the name being queried in the application code) to the JNDI name of the JDBC resource. The JNDI name is the same as the name of the JDBC resource as defined in the resource file when the resource is created.

```
<resource-ref>
   <res-ref-name>jdbc/HelloDbDs</res-ref-name>
   <jndi-name>jdbc/HelloDbDataSource</jndi-name>
</resource-ref>
```

The JNDI name in the Sun ONE specific deployment descriptor must match the JNDI name you assigned to the resource when you created and configured it.

Sample Applications

JNDI sample applications are in the following directory:

install_dir`/samples/jndi`

Chapter 5

Using the Java™ Message Service

This chapter describes how to use the Java™ Message Service (JMS) API. The Sun™ ONE Application Server has a fully integrated JMS provider: the Sun™ Open Net Environment (Sun ONE) Message Queue software.

This chapter contains the following sections:

- Introducing the JMS API
- Administration of the JMS Service
- Creating Applications That Use the JMS API
- Delivering SOAP Messages Using the JMS API
- Sample Applications

Introducing the JMS API

The Sun ONE Application Server provides support for applications that use the JMS API. The JMS API is a set of programming interfaces that provide a common way for Java applications to create, send, receive, and read messages in a distributed environment.

In particular, the JMS API is the standards-based way that J2EE™ applications perform asynchronous messaging. Accordingly, J2EE components (web or EJB™ components) can use the JMS API to send messages that can be consumed asynchronously by a specialized EJB, called a message-driven bean.

For more detailed information about JMS concepts and JMS support in Sun ONE Application Server, see the *Sun ONE Application Server Administrator's Guide*.

The rest of this section includes these topics:

- JMS Provider

- JMS Clients
- JMS Messaging Models and Interfaces

JMS Provider

Sun ONE 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. Sun ONE Application Server uses the Sun ONE Message Queue software as its native JMS provider. The Sun ONE Message Queue software is tightly integrated into Sun ONE Application Server, providing transparent JMS messaging support. This support (known within Sun ONE Application Server as the *JMS Service*) requires only minimal administration.

For more information about the Sun ONE Message Queue, refer to the following documentation:

`http://docs.sun.com/db/prod/s1.s1msgqu#hic`

For general information about the JMS API, see the JMS web page at:

`http://java.sun.com/products/jms/index.html`

NOTE The Sun ONE Message Queue software supports the JMS 1.1 API. However, Sun ONE Application Server supports the JMS 1.0.2 API in its application components (application client container included)

When the JMS API is used inside web components and EJB components, certain restrictions are placed on its use, which are outlined in the J2EE specification.

JMS Clients

JMS clients (components or applications) exchange messages by way of the JMS provider. Message producers send messages to the JMS provider, from which message consumers receive them.

A JMS client can be any type of J2EE application component: a web application, an EJB component, an Application Client Container client, and so on. To set up a JMS client, see “Creating Applications That Use the JMS API” on page 73.

A specialized JMS client called a *message-driven bean* is one of a family of EJB components. Unlike other EJB components (session beans and entity beans) message-driven beans are invoked asynchronously. For more information about message-driven beans, see the EJB 2.0 Specification (`http://java.sun.com/products/ejb/docs.html`) and the *Sun ONE Application Server Developer's Guide to Enterprise JavaBeans Technology*.

JMS Messaging Models and Interfaces

The JMS API supports two messaging models:

- **Point-to-point**: allows two applications to communicate by sending and receiving messages through a `Destination` called a `Queue`.
- **Publish-subscribe**: allows several messaging applications to communicate through a `Destination` called a `Topic`. Messages are sent by publishing to a `Topic`. Messages are received by *subscribers*.

Regardless of the messaging model, the link between applications and the JMS provider is the `Connection` object. Applications get their `Connection` objects from `ConnectionFactory` objects.

To maximize the portability of an application between JMS providers, provider-specific messaging features are encapsulated in *administered objects*. A JMS administered object implements one of four JMS interfaces, two for each messaging model.

The following table lists the JMS administered object interfaces. The first column lists the JMS parent interface, the second column lists the corresponding interfaces in the point-to-point domain, and the third column lists the corresponding interfaces in the publish-subscribe domain.

Table 5-1 JMS Interfaces

JMS Parent	Point-to-Point	Publish-Subscribe
`ConnectionFactory`	`QueueConnectionFactory`	`TopicConnectionFactory`
`Destination`	`Queue`	`Topic`

JMS providers supply:

- Classes that implement the `Queue`, `Topic`, `QueueConnectionFactory`, and `TopicConnectionFactory` interfaces.

- Tools to create and configure the administered object class instances according to the deployment requirements. Administrators use these tools to set provider-specific parameters. These tools can also store the administered objects in a Java Naming and Directory Interface™ (JNDI) repository.

This programming model lets you write JMS programs that are provider-independent. Applications look up administered objects using the JNDI API.

Administration of the JMS Service

To configure the JMS Service and prepare JMS resources for use in applications deployed to the Sun ONE Application Server, you must perform these tasks:

- Configuring the JMS Service
- Checking Whether the JMS Provider Is Running
- Creating Physical Destinations
- Creating JMS Resources: Destinations and Connection Factories

For information about other JMS administration tasks, see the *Sun ONE Application Server Administrator's Guide* and the Sun ONE Message Queue documentation at:

`http://docs.sun.com/db/prod/s1.s1msgqu#hic`

Configuring the JMS Service

You can edit or check the JMS Service configuration in the following ways:

- Using the Administration Interface
- Using the Command Line Interface

NOTE Configuration of the JMS Service should be done only when the Sun ONE Application Server instance is stopped.

The "Using The Administration Interface" section describes each connection pool setting. The "Using The Command Line Interface" section merely lists syntax and default values.

Using the Administration Interface

To edit the JMS Service configuration using the Administration interface, perform the following tasks:

1. Open the JMS component under your server instance.
2. Click Service.
3. You can edit the following information. Defaults are displayed.
   - Log Level - Controls the type of messages logged by the JMS Service to the server log. You can select a specific level, or you can select the default level set in the Log Service. For details, see the description of the Log Service in the *Sun ONE Application Server Administrator's Guide*.
   - Port - Specifies the port number used by the JMS provider. The default is `7676`.
   - Admin Username - Specifies the administrator user name for the JMS provider. The default is `admin`.
   - Admin Password - Specifies the administrator password for the JMS provider. The default is `admin`.
   - Start Timeout (secs) - Specifies the amount of time the server instance waits at startup for the corresponding JMS instance to respond. If there is no response, startup is aborted. If set to `0`, the server instance waits indefinitely. The default is `30` seconds.
   - Start Arguments - Specifies the string of arguments supplied for startup of the corresponding JMS instance. By default, there are no arguments.
   - Start Enabled - If checked (the default), the Sun ONE Application Server instance is responsible for starting up and shutting down the JMS provider. If unchecked, the Sun ONE Application Server instance does not start up nor shut down the JMS provider (either because the JMS provider is not used or because it is managed independently of the Sun ONE Application Server).
4. Click the Properties button to activate and specify values for any properties your application requires.
   a. To add a property, type its name and value in the Name and Value fields.
   b. To activate a property, check its box.
   c. If you need to add more Name and Value fields, click the Add button.

   The following table lists the standard JMS Service properties.

Table 5-2 JMS Service Properties

Property	Default	Description
`instance-name`	*domain_instance*	Specifies the full Sun ONE Message Queue broker instance name, which is a concatenation of the domain and server instance names. For example: `domain1_server1`.
`instance-name-suffix`	none	Specifies a suffix to add to the full Sun ONE Message Queue broker instance name. The suffix is separated from the instance name by an underscore character (_). For example, if the instance name is `domain1_server1`, appending the suffix `xyz` changes the instance name to `domain1_server1_xyz`.
`append-version`	`false`	If `true`, appends the major and minor version numbers, preceded by underscore characters (_), to the full Sun ONE Message Queue broker instance name. For example, if the instance name is `domain1_server1`, appending the version numbers changes the instance name to `domain1_server1_7_0`.

5. Click the OK button to return to the main JMS Service page.
6. Click the Save button.

Using the Command Line Interface

To configure the JMS service using the command line interface, use the `asadmin set` command. The syntax is as follows, with defaults shown for optional parameters that have them:

`asadmin set --user` *admin_user* `[--password` *admin_password*`] [--passwordfile` *password_file*`] [--host localhost] [--port 4848] [--secure | -s]` *attribute_name=value* `[`*attribute_name=value*`] *`

For more information about the general `asadmin` parameters (`--user`, `--password`, `--passwordfile`, `--host`, `--port`, and `--secure`), see the *Sun ONE Application Server Administrator's Guide.*

The *attribute_name* is a hierarchical name that looks like this:

instance`.jms-service.`*jms_attribute_name*

The *instance* is the application server instance name. The *jms_attribute_name* is the JMS service attribute that needs to be configured. For example:

`server1.jms-service.port`

To view the list of JMS service attribute names that can be configured using the `asadmin set` command, use the `asadmin get` command with a wildcard. The `asadmin get` command has the same syntax as the `asadmin set` command. For example:

```
asadmin get --user joeuser --password secret "server1.jms-service.*"
```

A list of attribute names for configuring the JMS service of the `server1` application server instance is displayed as follows:

```
server1.jms-service.logLevel
server1.jms-service.startArgs
server1.jms-service.adminPassword
server1.jms-service.port
server1.jms-service.enabled
server1.jms-service.adminUserName
server1.jms-service.initTimeoutInSeconds
```

Here is an example of running the `asadmin set` command:

```
asadmin set --user joeuser --password secret
server1.jms-service.enabled=false
```

The *attribute_name* for a JMS property is a hierarchical name that looks like this:

instance`.jms-service.property.`*jms_property_name*

The *instance* is the application server instance name. The *jms_property_name* is the JMS service property that needs to be configured. Here is an example of running the `asadmin set` command to set a JMS property:

```
asadmin set --user joeuser --password secret
server1.jms-service.property.instance-name-suffix=xyz
```

Checking Whether the JMS Provider Is Running

You can use the `asadmin jms-ping` command to check whether a Sun ONE Message Queue instance is running. The syntax is as follows, with defaults shown for optional parameters that have them:

`asadmin jms-ping --user` *admin_user* `[--password` *admin_password*`]`
`[--passwordfile` *password_file*`] [--host localhost] [--port 4848]`
`[--secure | -s]` *instance_name*

For example:

```
asadmin jms-ping --user joeuser --password secret server1
```

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.

This section describes how to create a physical destination. To create a destination resource, see "Creating JMS Resources: Destinations and Connection Factories" on page 69.

You can create a JMS physical destination in the following ways:

- Using the Administration Interface
- Using the Command Line Interface

The "Using The Administration Interface" section describes each connection pool setting. The "Using The Command Line Interface" section merely lists syntax and default values.

Using the Administration Interface

To create a JMS physical destination using the Administration interface, perform the following tasks:

1. Open the JMS component under your server instance.
2. Click Service, then click Physical Destinations.
3. Click the New button.
4. Enter the following information:
    - Destination Name (required) - Specify the name of the physical destination.
    - Type (required) - Select queue or topic from the list.
5. Click the OK button.

Using the Command Line Interface

To create a JMS physical destination using the command line, use the `asadmin create-jmsdest` command. The syntax is as follows, with defaults shown for optional parameters that have them:

```
asadmin create-jmsdest --user admin_user [--password admin_password]
[--passwordfile password_file] [--host localhost] [--port 4848]
[--secure | -s] [--instance instance_name] --desttype dest_type
[--property (name=value)[:name=value]*] dest_name
```

For more information about the parameters specific to `asadmin create-jmsdest`, see "Using the Administration Interface" on page 68. For more information about the general `asadmin` parameters (`--user`, `--password`, `--passwordfile`, `--host`, `--port`, and `--secure`), see the *Sun ONE Application Server Administrator's Guide*.

For example:

```
asadmin create-jmsdest --user joeuser --password secret --desttype
topic MyDest
```

To delete a JMS physical destination, use the following command:

```
asadmin delete-jmsdest --user admin_user [--password admin_password]
[--passwordfile password_file] [--host localhost] [--port 4848]
[--secure | -s] [--instance instance_name] --desttype dest_type dest_name
```

For example:

```
asadmin delete-jmsdest --user joeuser --password secret --desttype
topic MyDest
```

To list JMS physical destinations, use the following command:

```
asadmin list-jmsdest --user admin_user [--password admin_password]
[--passwordfile password_file] [--host localhost] [--port 4848]
[--secure | -s] [--desttype dest_type] instance_name
```

For example:

```
asadmin list-jmsdest --user joeuser --password secret --desttype
topic server1
```

Creating JMS Resources: Destinations and Connection Factories

You can create two kinds of JMS resources in Sun ONE Application Server:

- **Connection Factories:** administered objects that implement the `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:

- Using the Administration Interface
- Using the Command Line Interface

The "Using The Administration Interface" section describes each connection pool setting. The "Using The Command Line Interface" section merely lists syntax and default values.

Using the Administration Interface

To create a JMS resource using the Administration interface, perform the following tasks:

1. Open the JMS component under your server instance.
2. Click Connection Factories to create a connection factory, or click Destination Resources to create a queue or topic.
3. Click the New button.
4. Enter the following information:
   - JNDI Name (required) - Enter the JNDI name that application components must use to access the JMS resource. For more information, see "Looking Up Connection Factories" on page 74 and "Looking Up Destinations" on page 75.
   - Type (required) - Select the type of the JMS resource.
     - If you are on the Connection Factories page, the types are:

       ```
       javax.jms.TopicConnectionFactory
       javax.jms.QueueConnectionFactory
       ```

     - If you are on the Destination Resources page, the types are:

       ```
       javax.jms.Topic
       javax.jms.Queue
       ```

   - Description (optional) - You can enter a text description of the JMS resource.
5. Check the Resource Enabled box to enable the JMS resource.

   If a JMS resource is disabled, no application component can connect to it, but its configuration remains in the server instance.
6. Click the OK button.

7. To add properties to a JMS resource, perform the following tasks:
   a. Go back to the Connection Factories or Destination Resources page.
   b. Click the JMS resource you just created.
   c. Click the Properties button.
   d. Specify names and values for any properties you want to use. If you need another name-value row, use the Add button to add it. The following table lists the standard JMS resource properties.

Table 5-3 JMS Resource Properties

Property	Default	Description
`imqDestinationName`	none	Specifies the JMS physical destination name associated with this JMS resource. You must specify this property for JMS resources of the Type `javax.jms.Topic` or `javax.jms.Queue`. The *Sun ONE Message Queue Administrator's Guide* shows a default value for this property, but this does not apply in the Sun ONE Application Server environment.
`imqBrokerHostName`	the same host name as the Sun ONE Application Server instance (`localhost`)	Specifies the host name where the JMS service (Sun ONE Message Queue broker) is running. For JMS resources of the Type`javax.jms.TopicConnectionFactory` or `javax.jms.QueueConnectionFactory`.
`imqBrokerHostPort`	the JMS Service's Port setting	Specifies the port where the JMS service (Sun ONE Message Queue broker) is running. For JMS resources of the Type`javax.jms.TopicConnectionFactory` or `javax.jms.QueueConnectionFactory`.

Table 5-3 JMS Resource Properties *(Continued)*

Property	Default	Description
`imqConfiguredClientID`	none	Specifies the JMS Client Identifier to be associated with a `Connection` created using the `createQueueConnection` and `createTopicConnection` methods of the `QueueConnectionFactory` and `TopicConnectionFactory` classes, respectively. For JMS resources of the Type`javax.jms.TopicConnectionFactory` or `javax.jms.QueueConnectionFactory`. Durable subscription names are unique and only valid within the scope of a client identifier. To create or reactivate a durable subscriber, the connection must have a valid client identifier. The JMS specification ensures that client identifiers are unique and that a given client identifier is allowed to be used by only one active connection at a time.

e. Click the OK button.

f. Click the Save button.

8. Go to the server instance page.

9. Click the General tab.

10. Click the Apply Changes button.

Using the Command Line Interface

To create a JMS resource using the command line, use the `asadmin create-jms-resource` command. The syntax is as follows, with defaults shown for optional parameters that have them:

`asadmin create-jms-resource --user` *admin_user* `[--password` *admin_password*`] [--passwordfile` *password_file*`] [--host localhost] [--port 4848] [--secure | -s] [--instance` *instance_name*`] --resourcetype` *resource_type* `[--enabled=true] [--description` *text*`] [--property (`*name*`=`*value*`)[:`*name*`=`*value*`]*]` *jndi_name*

For more information about the parameters specific to `asadmin create-jms-resource`, see "Using the Administration Interface" on page 70. For more information about the general `asadmin` parameters (`--user`, `--password`, `--passwordfile`, `--host`, `--port`, and `--secure`), see the *Sun ONE Application Server Administrator's Guide.*

For example:

```
asadmin create-jms-resource --user joeuser --password secret
--resourcetype javax.jms.Topic --property
imqDestinationName=testTopic MyTopic
```

To delete a JMS resource, use the following command:

`asadmin delete-jms-resource --user` *admin_user* `[--password` *admin_password*`] [--passwordfile` *password_file*`] [--host localhost] [--port 4848] [--secure | -s] [--instance` *instance_name*`]` *jndi_name*

For example:

```
asadmin delete-jms-resource --user joeuser --password secret MyTopic
```

To list JMS resources, use the following command:

`asadmin list-jms-resources --user` *admin_user* `[--password` *admin_password*`] [--passwordfile` *password_file*`] [--host localhost] [--port 4848] [--secure | -s] [--resourcetype` *resource_type*`] [--instance` *instance_name*`]`

For example:

```
asadmin list-jms-resources --user joeuser --password secret
--resourcetype Topic --instance server1
```

After you create the JMS resource, you must reconfigure the server instance using the following command:

`asadmin reconfig --user` *user* `[--password` *password*`] [--passwordfile` *password_file*`] [--host localhost] [--port 4848] [--secure | -s][--discardmanualchanges=false | --keepmanualchanges=false]` *instance_name*

For example:

```
asadmin reconfig --user joeuser --password secret server1
```

Creating Applications That Use the JMS API

This section discusses how to use the JMS API in applications:

- Basic Steps for Developing a JMS Client

- Processing JMS Messages
- JMS Cleanup

Basic Steps for Developing a JMS Client

Developing a JMS client involves these tasks:

- Importing the JMS Package
- Looking Up Connection Factories
- Creating Connections
- Creating Sessions
- Looking Up Destinations
- Creating Message Producers
- Creating Message Consumers
- Starting the Connection

Importing the JMS Package

The `javax.jms` and `javax.naming` packages define all the JMS interfaces necessary to develop a JMS client. Import these packages as follows:

```
import javax.jms.*;
import javax.naming.*;
```

Looking Up Connection Factories

The recommended JNDI subcontext for JMS connection factories is `java:comp/env/jms`. The resource lookup in the application code looks like this for point-to-point messaging:

```
InitialContext ic = new InitialContext();
QueueConnectionFactory QCFactory = (QueueConnectionFactory)
ic.lookup("java:comp/env/jms/MyQCF");
```

For publish-subscribe messaging, the only difference is that `TopicConnectionFactory` is used instead.

For more information about the JNDI API, see Chapter 4, “Using the Java Naming and Directory Interface™.”

Creating Connections

After you have looked up a connection factory, use it to create connections to the JMS provider. The following code creates a connection for point-to-point messaging:

```
QueueConnection connection = QCFactory.createQueueConnection();
```

The following code creates a connection for publish-subscribe messaging:

```
TopicConnection tConnection = TCFactory.createTopicConnection();
```

NOTE A servlet can create Java threads, but this is not recommended.

Creating Sessions

Sessions are lightweight JMS objects which provide a context for producing and consuming messages. Sessions are used to create message producers and message consumers, as well as to build messages themselves. The following code creates a session for point-to-point messaging:

```
QueueSession session = connection.createQueueSession(false,
QueueSession.AUTO_ACKNOWLEDGE);
```

The following code creates a session for publish-subscribe messaging:

```
TopicSession tSession = tConnection.createTopicSession(false,
TopicSession.AUTO_ACKNOWLEDGE);
```

For more information about acknowledgement modes, of which `AUTO_ACKNOWLEDGE` is one, see “Using Acknowledgements” on page 78.

Looking Up Destinations

You look up queues and topics by their JNDI names directly. Unlike connection factories, queues and topics do not use resource references in the deployment descriptor files.

The following code looks up a queue:

```
Queue queue = (Queue) ic.lookup("java:comp/env/jms/sampleQ");
```

The following code looks up a topic:

```
Topic topic = (Topic) ic.lookup("java:comp/env/jms/sampleT");
```

To create a queue or topic, see “Creating JMS Resources: Destinations and Connection Factories” on page 69.

Creating Message Producers

Use the session and destination to create a message producer. The following code creates a message producer for point-to-point messaging:

```
QueueSender qSender = session.createSender(queue);
```

The following code creates a message producer for publish-subscribe messaging:

```
TopicPublisher tPublisher = tSession.createPublisher(topic);
```

Creating Message Consumers

You use the session and destination to create a message consumer as well. The following code creates a message consumer for point-to-point messaging:

```
QueueReceiver qReceiver = session.createReceiver(queue);
```

The following code creates a message consumer for publish-subscribe messaging:

```
TopicSubscriber tSubscriber = tSession.createSubscriber(topic);
```

For both point-to-point and publish-subscribe message consumers, you can optionally register a message listener with your message consumer to enable asynchronous messaging. First, write a class that implements the `MessageListener` interface (which contains the `onMessage()` method), then call the `setMessageListener()` method of the message consumer:

```
tSubscriber.setMessageListener(this);
```

It is assumed that the class containing the example code above implements the `MessageListener` interface, which is why `this` is used in the `setMessageListener()` method.

Starting the Connection

Use the following code to start a connection for point-to-point messaging:

```
connection.start();
```

Use the following code to start a connection for publish-subscribe messaging:

```
tConnection.start();
```

Processing JMS Messages

When you have created message producers and consumers, you are ready to perform these tasks:

- Sending Messages

- Receiving Messages
- Acknowledging Received Messages

Sending Messages

Point-to-point messaging sends one or more messages to a target queue as shown in the following code:

```
TextMessage msgSent = session.createTextMessage();
String msg = "Cold weather today";
msgSent.setText( to + ":" + from + " ["+new Date()+"]: " + msg);
qSender.send(msgSent);
```

Publish-subscribe messaging publishes messages to a given topic as shown in the following code:

```
TextMessage msgPub = tSession.createTextMessage();
msgPub.setText("temperature: 35 degrees");
tPublisher.publish(msgPub);
```

Receiving Messages

JMS messages can be received (or consumed) synchronously or asynchronously, whether they are used with point-to-point or publish-subscribe message consumers.

JMS messages can be received synchronously in any of these ways:

- Call the `receive` method with no arguments or an argument of `0`, so that the method blocks indefinitely until a message arrives:

  ```
  TextMessage msgReceived = (TextMessage) qReceiver.receive();
  ```

- Call the `receive` method with a timeout argument (in milliseconds) greater than `0`:

  ```
  TextMessage msgReceived = (TextMessage) qReceiver.receive(2000);
  ```

- Call the `receiveNoWait` method to receive a message only if one is available:

  ```
  TextMessage msgReceived = (TextMessage)
  qReceiver.receiveNoWait();
  ```

To receive JMS messages asynchronously, you register a `MessageListener` with the consumer as described in “Creating Message Consumers” on page 76. The client consumes a message when a session thread invokes the `onMessage()` method of the `MessageListener` object.

After you have received a message, you can get its contents:

```
out("\nMessage received: " + msgReceived.getText());
```

Acknowledging Received Messages

You can guarantee successful message delivery using either of these mechanisms supported by a JMS session:

- Using Acknowledgements
- Using Transactions for Message Acknowledgement

Using Acknowledgements

You can use one of these acknowledgement modes when you instantiate a session as described in “Creating Sessions” on page 75:

- `AUTO_ACKNOWLEDGE` - The session automatically acknowledges a client’s receipt of a message either when the session has successfully returned from a call to `receive` or when the message listener the session has called to process the message successfully returns.
- `DUPS_OK_ACKNOWLEDGE` - The session lazily acknowledges the delivery of messages. This is likely to result in the delivery of some duplicate messages if the JMS provider fails, so it should only be used by consumers that can tolerate duplicate messages. Use of this mode can reduce session overhead by minimizing the work the session does to prevent duplicates.
- `CLIENT_ACKNOWLEDGE` - The JMS client acknowledges a consumed message by calling the message’s `acknowledge` method. Acknowledging a consumed message acknowledges all messages that the session has consumed. When client acknowledgment mode is used, a client may build up a large number of unacknowledged messages while attempting to process them.

Using Transactions for Message Acknowledgement

You can send and receive messages within local or distributed transactions to ensure message delivery. The JMS API provides methods for initiating, committing, or rolling back a local transaction.

Distributed transactions use the Sun ONE Application Server’s transaction service. The following table lists the XA classes supported by the JMS provider. The first column lists the JMS parent classes, the second column lists the corresponding classes in the point-to-point domain, and the third column lists the corresponding classes in the publish-subscribe domain.

Table 5-4 XA Classes

JMS Parent	Point-to-Point	Publish-Subscribe
XAConnectionFactory	XAQueueConnectionFactory	`XATopicConnectionFactory`
XAConnection	`XAQueueConnection`	`XATopicConnection`
XASession	`XAQueueSession`	`XATopicSession`

You can use the `XASession.getTransacted()` method to find out whether the current JMS distributed session is within a transaction.

For details about the transaction service, see Chapter 3, "Using the Transaction Service."

JMS Cleanup

To free system resources, make sure your application closes message producers, message consumers, sessions, and connections in that order. For example:

```
qSender.close()
qReceiver.close()
session.close()
connection.close()
```

Calling `close()` statements inside a `finally` block is strongly recommended.

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 *Sun ONE Application Server Developer's Guide to Web Services* and the Apache SOAP web site:

`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:

- Sending SOAP Messages Using the JMS API
- Receiving SOAP Messages Using the JMS API

Sending SOAP Messages Using the JMS API

You use the `MessageTransformer` utility to convert a SOAP message into a JMS message. You then send the JMS message containing the SOAP payload as you would a normal JMS message.

1. Import the library `com.sun.messaging.xml.MessageTransformer`. This is the utility whose methods you use to convert SOAP messages to JMS messages and the reverse.

   ```
   import com.sun.messaging.xml.MessageTransformer;
   ```

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). For more information on constructing a SOAP message, see the *Sun ONE Application Server Developer's Guide to Web Services* and the *Sun ONE Message Queue Developer's Guide*.

   ```
   *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 SunOne 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();
   ```

Receiving SOAP Messages Using the JMS API

You receive the JMS message containing the SOAP payload as you would a normal JMS message. You then use the `MessageTransformer` utility to convert the JMS message back into a SOAP message.

1. Import the library `com.sun.messaging.xml.MessageTransformer`. This is the utility whose methods you use to convert SOAP messages to JMS messages and the reverse.

   ```
   import com.sun.messaging.xml.MessageTransformer;
   ```

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. For more information about SOAP messages, see the *Sun ONE Application Server Developer's Guide to Web Services*.

Sample Applications

JMS sample applications are in the following directory:

install_dir`/samples/jms`

Message-driven bean sample applications are in the following directory:

install_dir`/samples/ejb/mdb`

Chapter 6

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
- Creating a JavaMail Session
- JavaMail Session Properties
- Looking Up a JavaMail Session
- Sending Messages Using JavaMail
- Reading Messages Using JavaMail
- Sample Applications

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 the *Sun ONE Application Server Administrator's 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:

- Using the Administration Interface
- Using the Command Line Interface

The "Using The Administration Interface" section describes each connection pool setting. The "Using The Command Line Interface" section merely lists syntax and default values.

Using the Administration Interface

To create a JavaMail session using the Administration interface, perform the following tasks:

1. Open the Java Mail Sessions component under your server instance.
2. Click the New button.
3. Enter the following information:
    - JNDI Name (required) - Enter the JNDI name that application components must use to access the JavaMail session. For more information, see "Looking Up a JavaMail Session" on page 87.
    - Mail Host (required) - The mail server host name.
    - Default User (required) - The mail server user name.
    - Default Return Address (required) - The e-mail address the mail server uses to indicate the message sender.

- Description (optional) - You can enter a text description of the JavaMail session.

4. Check the Java Mail Session Enabled box to enable the JavaMail session.

   If a JavaMail session is disabled, no application component can connect to it, but its configuration remains in the server instance.

5. You can also edit the following Advanced settings:

   - Store Protocol - Specifies the storage protocol service, which connects to a mail server, retrieves messages, and saves messages in folder(s). Example values are `imap` (the default) and `pop3`.
   - Store Protocol Class - Specifies the service provider implementation class for storage. The default is `com.sun.mail.imap.IMAPStore`.
   - Transport Protocol - Specifies the transport protocol service, which sends messages. The default is `smtp`.
   - Transport Protocol Class - Specifies the service provider implementation class for transport. The default is `com.sun.mail.smtp.SMTPTransport`.
   - Debug Enabled - Enables debugging for this JavaMail session.

6. Click the OK button.
7. To add properties to a JavaMail session, perform the following tasks:

   a. Go back to the Java Mail Sessions page.

   b. Click the JavaMail session you just created.

   c. Click the Properties button.

   d. Specify names and values for any properties you want to use. If you need another name-value row, use the Add button to add it. For details about JavaMail session properties, see “JavaMail Session Properties” on page 87.

   e. Click the Save button.

8. Go to the server instance page.
9. Click the General tab.
10. Click the Apply Changes button.

Using the Command Line Interface

To create a JavaMail session using the command line, use the `asadmin create-javamail-resource` command. The syntax is as follows, with defaults shown for optional parameters that have them:

```
asadmin create-javamail-resource --user admin_user [--password
admin_password] [--passwordfile password_file] [--host localhost] [--port
4848] [--secure | -s] [--instance instance_name] --mailhost mail_host
--mailuser mail_user --fromaddress address [--storeprotocol=imap]
[--storeprotocolclass=com.sun.mail.imap.IMAPStore]
[--transprotocol=smtp]
[--transprotocolclass=com.sun.mail.smtp.SMTPTransport]
[--debug=false] [--enabled=true] [--description text] [--property
(name=value)[:name=value]*] jndi_name
```

For more information about the parameters specific to `asadmin create-javamail-resource`, see "Using the Administration Interface" on page 84. For more information about the general `asadmin` parameters (`--user`, `--password`, `--passwordfile`, `--host`, `--port`, and `--secure`), see the *Sun ONE Application Server Administrator's Guide*.

For example:

```
asadmin create-javamail-resource --user joeuser --password secret
--mailhost MailServer --mailuser MailUser --fromaddress
user@mailserver.com MailSession
```

To delete a JavaMail session, use the following command:

```
asadmin delete-javamail-resource --user admin_user [--password
admin_password] [--passwordfile password_file] [--host localhost] [--port
4848] [--secure | -s] [--instance instance_name] jndi_name
```

For example:

```
asadmin delete-javamail-resource --user joeuser --password secret
MailSession
```

To list JavaMail sessions, use the following command:

```
asadmin list-javamail-resources --user admin_user [--password
admin_password] [--passwordfile password_file] [--host localhost] [--port
4848] [--secure | -s] [--instance instance_name]
```

For example:

```
asadmin list-javamail-resources --user joeuser --password secret
--instance server1
```

After you create the JavaMail session, you must reconfigure the server instance using the following command:

`asadmin reconfig --user` *user* `[--password` *password*`] [--passwordfile` *password_file*`] [--host localhost] [--port 4848] [--secure | -s][--discardmanualchanges=false | --keepmanualchanges=false]` *instance_name*

For example:

```
asadmin reconfig --user joeuser --password secret server1
```

JavaMail Session Properties

You can set properties for a JavaMail `Session` object. Every property name must start with a `mail-` prefix. Sun ONE 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.password` in a JavaMail Session object, first define the property as follows:

- Name - `mail-password`
- Value - `secret`

After you get the JavaMail `Session` object, you can get the `mail.password` property to retrieve the value `secret`, as follows:

```
String password = session.getProperty("mail.password");
```

Looking Up a JavaMail Session

The recommended 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 recommended. 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 4, "Using the Java Naming and Directory Interface™."

Sending Messages Using JavaMail

To send a message using JavaMail, perform the following tasks:

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, as described in "Looking Up a JavaMail Session" on page 87:

   ```
   InitialContext ic = new InitialContext();
   String snName = "java:comp/env/mail/MyMailSession";
   Session session = (Session)ic.lookup(snName);
   ```

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));
   MimeBodyPart mbp = new MimeBodyPart();
   mbp.setText(msgTxt);
   Multipart mp = new MimeMultipart();
   mp.addBodyPart(mbp);
   msg.setContent(mp);
   ```

5. Send the message:

```
Transport.send(msg);
```

Reading Messages Using JavaMail

To read a message using JavaMail, perform the following tasks:

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, as described in "Looking Up a JavaMail Session" on page 87:

```
InitialContext ic = new InitialContext();
String snName = "java:comp/env/mail/MyMailSession";
Session session = (javax.mail.Session)ic.lookup(snName);
```

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 default folder, then get the INBOX folder:

```
Folder folder = store.getDefaultFolder();
folder = folder.getFolder("INBOX");
```

6. It is efficient to read the `Message` objects (which represent messages on the server) into an array:

```
Message[] messages = folder.getMessages();
```

Sample Applications

JavaMail sample applications are in the following directory:

install_dir/samples/javamail

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

X