Developer's Guide

Sun™ ONE Application Server

Version 7

816-7149-10
September 2002

Contents

About This Guide

This guide describes how to create and run Java 2 Platform, Enterprise Edition (J2EE) applications that follow the new open Java standards model for Servlets, Enterprise JavaBeans (EJB components), and JavaServer Pages (JSPs) on the Sun™ Open Net Environment (Sun ONE) Application Server 7. In addition to describing programming concepts and tasks, this guide offers sample code, implementation tips, reference material, and a glossary.

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 servlet, JSP, 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.

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*
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*

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 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 features such as JDBC, JNDI, JTS, JMS, JavaMail, resources, and connectors	*Developer's Guide to J2EE Features and Services*
Creating custom NSAPI plugins	*Developer's Guide to NSAPI*
Performing the following administration tasks: • 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	*Administrator's Guide*
Editing server configuration files	*Administrator's Configuration File Reference*

Sun ONE Application Server Documentation Roadmap *(Continued)*

For information about	See the following
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 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 Sun ONE Message Queue.	The Sun ONE Message Queue documentation at: http://docs.sun.com/?p=/coll/S1_MessageQueue_30

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, "Designing Applications"

  This chapter summarizes the Sun ONE Application Server application design process and offers effective development guidelines.

- Chapter 2, "Developing J2EE Applications"

  This chapter describes how to set up a development environment and provides basic steps for creating application components.

- Chapter 3, "Securing J2EE Applications"

  This chapter describes how to write secure J2EE applications, which contain components that perform user authentication and access authorization for servlets and EJB business logic.

- Chapter 4, “Assembling and Deploying J2EE Applications”

  This chapter describes the contents of Sun ONE Application Server modules and how these modules are assembled separately or together in an application.

- Chapter 5, “Debugging J2EE Applications”

  This chapter gives guidelines for debugging applications in Sun ONE Application Server 7.

- Chapter 6, “Developing Lifecycle Listeners”

  This chapter describes how to create and use a lifecycle module, which is automatically initiated at server startup and notified of server shutdown.

Finally, a *Glossary* and *Index* are provided.

Related Information

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

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 Servlets and JSPs:

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

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

Programming with EJB components:

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

Programming with JDBC:

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

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

Documentation Conventions

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

- General Conventions
- Conventions Referring to Directories

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 15.

  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 15 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

Designing Applications

This chapter summarizes the Sun ONE Application Server application design process and offers effective development guidelines.

This chapter contains the following sections:

- Application Requirements
- About the J2EE Programming Model
- Best Practices for Designing J2EE Applications

Application Requirements

When developing a Sun ONE Application Server application, start by identifying the application requirements. Typically, this means developing a widely deployable application that is fast and secure, and that can reliably handle additional requests as new users are added.

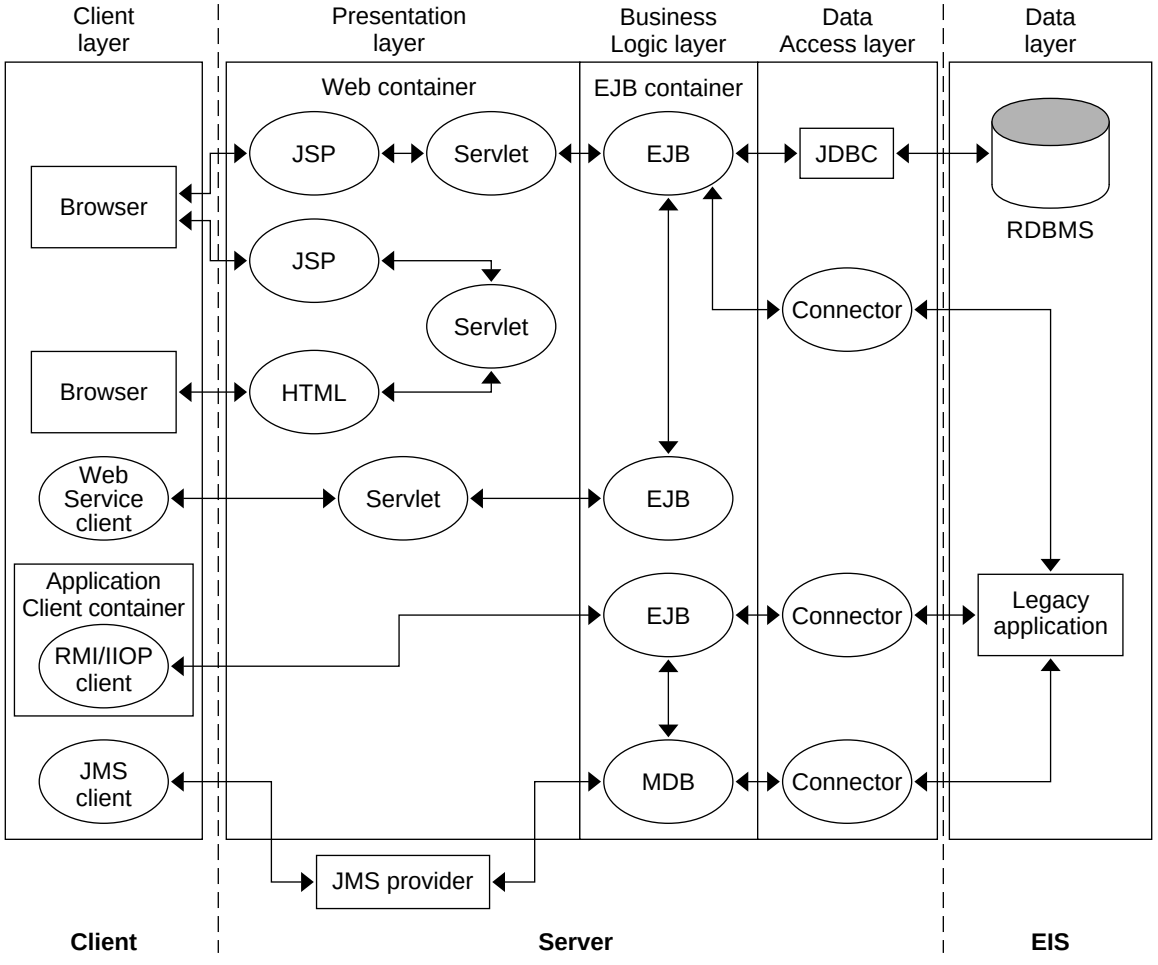
The Sun ONE Application Server meets these needs because it supports the J2EE APIs as well as a set of pre-existing high performance features. For example, for an online banking application, you can deliver:

- Security
- Rapid deployment of specific features; for example, account transfers, account reporting, online trades, special offers to qualified customers
- Management and administration of different types of end users; for example, individuals, corporations, or internal users
- Internal reporting
- Enterprise Information System (EIS) connectivity that provides access to information stored in legacy databases

About the J2EE Programming Model

In the following figure, client machines are running web browser, web service, RMI/IIOP, or JMS clients; J2EE server machines are running the Sun ONE Application Server; and EIS server machines are running databases and legacy applications. JSPs and servlets provide the interface to the client tier, EJB components reside in the business tier, and connectors provide the interface to legacy applications.

J2EE application layers



A distributed application model allows different individual application layers to focus on different functional elements, thereby improving performance.

These application layers are discussed in the following sections:

- The Client Layer
- The Presentation Layer
- The Business Logic Layer
- The Data Access Layer

The Client Layer

The *client layer* is where the user accesses the application. An application may require one of the following types of clients:

- Browser Clients
- Simple CORBA Clients
- ACC Clients
- Web Service Clients
- JMS Clients

For more information about components in the client layer, see the *Sun ONE Application Server Developer's Guide to Clients*.

Browser Clients

In many cases, the client is simply a browser.

Simple CORBA Clients

You can use any client that is compliant with the Common Object Request Broker Architecture (CORBA) to access EJB components deployed to Sun ONE Application Server. The client can be written in any CORBA-supported language, such as Java, C, C++, Visual Basic, and so on. CORBA clients are typically used when a stand-alone program or another application server acts as a client to services available on the Sun ONE Application Server.

Sun ONE Application Server supports access to EJB components using the IIOP protocol as specified in the Enterprise JavaBeans Specification, V2.0, and the Enterprise JavaBeans to CORBA Mapping specification.

Simple CORBA clients that do not use the Application Client Container (ACC) have the following limitations:

- JNDI is not supported. However, you can build name translations and do lookups using standard COSNaming binding.
- SSL over RMI/IIOP is not supported.
- Features that are configurable in the `sun-application-client.xml` and `sun-acc.xml` files are not available.

ACC Clients

Sun ONE Application Server supports Application Client Container (ACC) CORBA clients that are written in Java and that use RMI/IIOP to communicate with the server.

The Sun ONE Application Server provides system services that enable ACC client programs to execute. ACC clients use the Java Naming and Directory Interface (JNDI) to locate services such as EJB components, JDBC resources, and JavaMail. You can configure special features using the `application-client.xml` and `sun-application-client.xml` deployment descriptor files.

All CORBA clients described in this manual are also ACC clients unless otherwise specified.

Web Service Clients

Typically a business application sends a request to a web service at a given URL using the SOAP protocol over HTTP. The service receives the request, processes it, and returns a response. An often-cited example of a web service is that of a stock quote service, in which the request asks for the current price of a specified stock, and the response gives the stock price.

Sun ONE Application Server supports Apache SOAP version 2.2 and JAX RPC 1.1. Apache SOAP web services support is also built into Sun ONE Studio 4.

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

JMS Clients

Java applications that use Java Message Service (JMS) are called JMS clients. A JMS client can create, send, receive, and read messages. Clients that send messages are called *producers*, and those that receive messages are called *consumers*. For a detailed overview of JMS, see the *Sun ONE Message Queue Developer's Guide.*

The Presentation Layer

The *presentation layer* is where the user interface is dynamically generated. An application may require the following J2EE components in the presentation layer:

- Servlets
- JSPs
- Static Content

In addition, an application may require the following non-J2EE, HTTP server-based components in the presentation layer:

- SHTML
- CGI

For more information about components in the presentation layer, see the *Sun ONE Application Server Developer's Guide to Web Applications*.

Servlets

Servlets handle the application's presentation logic. Servlets are the page-to-page navigation dispatchers, and they also provide session management and simple input validation. Servlets tie business logic elements together.

A servlet developer must understand programming issues related to HTTP requests, security, internationalization, and web statelessness (such as sessions, cookies, and time-outs). For a Sun ONE Application Server application, servlets must be written in Java. Servlets are likely to call JSPs, EJB components, and JDBC objects. Therefore, a servlet developer works closely with the application element developers.

JSPs

JSPs handle most application display tasks, and they work in conjunction with servlets to define the application's presentation screens and page navigation. JSPs are likely to call EJB components and JDBC objects. The EJB components typically encapsulate business logic functionality. As such, they carry out calculations and other repetitively requested tasks. JDBC objects are used to connect to databases, make queries, and return query results.

Static Content

You can also provide static content, such as images and HTML pages. Properly designed HTML pages provide:

- Uniform appearance across different browsers
- Efficient HTML loading across slow modem connections
- Dynamically generated page appearances that are servlet or JSP dispatched

SHTML

SHTML (Server-parsed HTML) files are HTML files that contain tags that are executed on the server. In addition to supporting the standard server-side tags, or SSIs, Sun ONE Application Server 7 allows you to embed servlets and define your own server-side tags.

CGI

Common Gateway Interface (CGI) programs run on the server and generate a response to return to the requesting client. CGI programs can be written in various languages, including C, C++, Java, Perl, and as shell scripts. CGI programs are invoked through URL invocation. Sun ONE Application Server complies with the version 1.1 CGI specification.

The Business Logic Layer

The *business logic layer* typically contains deployed EJB components that encapsulate business rules and other business functions in:

- Session Beans
- Entity Beans
- Message-Driven Beans

For more information about components in the business logic layer, see the *Sun ONE Application Server Developer's Guide to Enterprise JavaBeans Technology*.

Session Beans

Session beans encapsulate the business processes and rules logic. For example, a session bean could calculate taxes for a billing invoice. When there are complex business rules that change frequently (for example, due to new business practices or new government regulations), an application typically uses more session beans than entity beans, and session beans may need continual revision.

Session beans are likely to call a full range of JDBC interfaces, as well as other EJB components. Applications perform better when session beans are stateless, although session beans can be stateful. A stateful session bean is needed when a user-specific state, such as a shopping cart, must be maintained on the server.

Entity Beans

Entity beans represent persistent objects, such as a database row. Entity beans are likely to call a full range of JDBC interfaces. However, entity beans typically do not call other EJB components. The entity bean developer's role is to design an object-oriented view of an organization's business data. Creating this object-oriented view often means mapping database tables into entity beans. For example, the developer might translate a customer table, invoice table, and order table into corresponding customer, invoice, and order objects.

An entity bean developer works with session bean and servlet developers to ensure that the application provides fast, scalable access to persistent business data.

There are two types of entity bean persistence:

- Container managed persistence (CMP) - The EJB container is responsible for maintaining the interactions between the business logic and the database.
- Bean managed persistence (BMP) - The developer is responsible for writing the code that controls interaction with the database.

Message-Driven Beans

Message-driven beans are persistent objects that are likely to call a full range of JDBC interfaces, much like entity beans. However, message-driven beans have no local or remote interfaces as do other EJB components, and they differ from entity beans in how they are accessed.

A message-driven bean is a message listener that can reliably consume messages from a queue or a durable subscription. The messages may be sent by any J2EE component—from an application client, another EJB component, or a Web component—or from an application or a system that does not use J2EE technology.

For example, an inventory entity bean may send a message to a stock ordering message-driven bean when the amount of an item is below a set lower limit.

The Data Access Layer

In the *data access layer*, JDBC (Java database connectivity) is used to connect to databases, make queries, and return query results, and custom connectors work with the Sun ONE Application Server to enable communication with legacy EIS systems, such as IBM's CICS.

Developers are likely to integrate access to the following systems using J2EE CA (connector architecture):

- Enterprise resource management systems
- Mainframe systems
- Third-party security systems

For more information about JDBC, see the *Sun ONE Application Server Developer's Guide to J2EE Features and Services*.

For more information about connectors, see the *Sun ONE J2EE CA Service Provider Implementation Administrator's Guide* and the corresponding release notes.

Best Practices for Designing J2EE Applications

This section lists guidelines to consider when designing and developing an Sun ONE Application Server application, and is merely a summary. For more details, you may want to consult *Core J2EE Patterns: Best Practices and Design Strategies* by Deepak Alur, John Crupi, and Dan Malks.

The guidelines are grouped into the following goals:

- Presenting Data with Servlets and JSPs
- Creating Reusable Application Code
- Modularizing Applications

Presenting Data with Servlets and JSPs

Servlets are often used for presentation logic and serve as central dispatchers of user input and data presentation. JSPs are used to dynamically generate the presentation layout. Both servlets and JSPs can be used to conditionally generate different pages.

If the page layout is its main feature and there is minimal processing involved to generate the page, it may be easier to use a JSP for the interaction.

For example, after an online bookstore application authenticates a user, it provides a boilerplate portal front page for the user to choose one of several tasks, including a book search, purchase selected items, and so on. Since this portal conducts very little processing, it can be implemented as a JSP.

Think of JSPs and servlets as opposite sides of the same coin. Each can perform all the tasks of the other, but each is designed to excel at one task at the expense of the other. The strength of servlets is in processing and adaptability. However, performing HTML output from them involves many cumbersome `println` statements. Conversely, JSPs excel at layout tasks because they are simply HTML files and can be edited with HTML editors, though performing complex computational or processing tasks with them can be awkward. You can use JSPs and servlets together to get the benefits of both.

For more information on servlets and JSPs, see the *Sun ONE Application Server Developer's Guide to Web Applications*.

Creating Reusable Application Code

Aside from using good object-oriented design principles, there are several things to consider when developing an application to maximize reusability, including the following tips:

- Use relative paths and URLs so links remain valid if the code tree moves.
- Minimize Java in JSPs; instead, put Java in servlets and helper classes. JSP designers can revise JSPs without being Java experts.
- Use property files or global classes to store hard-coded strings such as the data source names, tables, columns, JNDI objects, or other application properties.
- Use session beans, rather than servlets and JSPs, to store business rules that are domain-specific or likely to change often, such as input validation.
- Use entity beans for persistent objects; using entity beans allows management of multiple beans per user.

- For maximum flexibility, use Java interfaces rather than Java classes.
- Use J2EE CA to access legacy data.

Modularizing Applications

The major factors to keep in mind when designing your J2EE Applications are:

- Functional Isolation
- Reusable Code
- Prepackaged Components
- Shared Framework Classes
- Session and Security Issues

For more information about assembling modules and applications, see Chapter 4, "Assembling and Deploying J2EE Applications."

Functional Isolation

Each component should do one thing and one thing only. For example, in a payroll system, one EJB component should access the 401k accounts while a separate bean accesses the salary database. This functional isolation of tasks leads to the physical isolation of business logic into two separate beans. If separate development teams create these beans, each team should develop its own EJB JAR package.

Scenario 1

Assume that the user interface development team works with both of the bean development teams. In this case, the UI development team should assemble its servlets, JSPs, and static files into one WAR file. For example:

payroll system EAR file = payroll EJB jar
+ 401k EJB JAR
+ 1 common war from the UI team

This isolation of functionality within an EAR file does not mean that components cannot interact with each other. The beans (in separate EJB JAR files) can call business methods from each other.

Scenario 2

Assume that each bean development team has its own UI development team. If this is the case, then each web development team should assemble its servlets, JSPs, and static files into separate WAR files. For example:

payroll system EAR file = payroll EJB jar
+ 401k EJB JAR
+ 1 payroll UI team's war + 1 401k UI team's war

With this setup, the components in each WAR file can access components from the other WAR file.

Assembly Formulas

Some general formulas should be followed when assembling modules and applications.

The following table outlines assembly formulas. The left column lists the type of development group, the middle column lists the teams in the group, and the right column lists the modularizing scheme.

Assembly formulas

Type of Development Group	Teams in Group	Modularizing Scheme
Small workgroup	1 web team + 1 EJB team	1 EAR = 1 EJB + 1 WAR
Enterprise workgroup	2 EJB teams + 1 web team + 1 component	1 EAR = 2 EJB + 1 WAR + 1 individual component

Reusable Code

Reusable components are the primary reason for assembling and deploying individual modules rather than applications. If the code developed by one team of developers is a reusable component that may be accessed by several applications (different EAR files), then that code should be deployed as an individual module. For more information, see Chapter 4, "Assembling and Deploying J2EE Applications."

Prepackaged Components

If you do not want to create your application from scratch, you can use prepackaged components. Today's leading J2EE component vendors offer many prepackaged components that provide a whole host of services. Their goal is to provide up to 60% of the standard components needed for an application. With Sun ONE Application Server, you can easily assemble applications that make use of these readily available components.

Shared Framework Classes

Sometimes several applications need to access a single modular library. In such cases, including the library in each J2EE application is not a good idea for these reasons:

- **Library size:** Most framework libraries are large, so including them in an application increases the size of the assembled application.
- **Different versions:** Because a separate classloader loads each application, several copies of the framework classes exist during runtime.

For tips on how to set up a library so multiple applications can share it, see "Circumventing Classloader Isolation," on page 78.

Session and Security Issues

If session sharing is a requirement, all the components that need to access a session should be contained in the same application.

NOTE Session sharing across application boundaries is not supported in Sun ONE Application Server and is a violation of the J2EE specification.

If an HTTP session needs to be shared between two WAR files in an EAR file, the session should be marked "distributed" in the deployment descriptor.

You should not allow unauthorized runtime access to classes, EJB components, and other resources. A component should only contain classes that are permitted to access other resources included in the component. In addition, you should use the standard J2EE declarative security (see Chapter 3, "Securing J2EE Applications") for sensitive tasks.

Chapter 2

Developing J2EE Applications

This chapter gives guidelines for developing applications in Sun ONE Application Server 7. It includes the following sections:

- Setting Up a Development Environment
- Steps for Creating Components

Setting Up a Development Environment

Setting up an environment for creating, assembling, deploying, and debugging your code involves installing the mainstream version of Sun ONE Application Server and making use of development tools, which are covered in the following sections:

- Installing and Preparing the Server for Development
- Development Tools

Installing and Preparing the Server for Development

For bundled Solaris 9, Sun ONE Application Server installation is part of the operating system installation process. For more information, see the *Solaris 9 Installation Guide.*

For all other platforms, installing the mainstream (non-evaluation) version of Sun ONE Application Server is recommended.

For bundled Solaris 9 or the mainstream installation, the following components are installed by default:

- Sun ONE Application Server Core, including:
  - Sun ONE Message Queue
  - Administration interface
- JDK
- Sun ONE Studio 4
- Sample Applications

For more information, see the *Sun ONE Application Server Installation Guide*.

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

- Locate utility classes and libraries so they can be accessed by the proper classloaders. For more information, see “Classloaders,” on page 74.
- Enable dynamic reloading. For more information, see “Dynamic Reloading,” on page 94.
- Set up debugging. For more information, see Chapter 5, “Debugging J2EE Applications.”
- Configure the JVM. For more information, see the *Sun ONE Application Server Administrator’s Guide*.

Development Tools

The following general tools are provided with Sun ONE Application Server:

- The asadmin Command
- The Administration Interface

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

- Sun ONE Studio
- Apache Ant
- Migration Tools

The following third-party tools may also be useful:

- Profiling Tools
- Source Code Control Tools
- Other Tools Supported Through Sun ONE Studio

The asadmin Command

The `asadmin` command allows you to configure a local or remote server and perform both administrative and development tasks at the command line. For information about deployment using `asadmin`, see "The asadmin Command," on page 96. For general information about `asadmin`, see the *Sun ONE Application Server Administrator's Guide.*

The Administration Interface

The Administration interface allows you to configure the server and perform both administrative and development tasks using a web browser. For information about deployment using the Administration interface, see "The Administration Interface," on page 98. For general information about the Administration interface, see the *Sun ONE Application Server Administrator's Guide.*

Sun ONE Studio

Sun ONE Studio 4 is an IDE (integrated development environment) that allows you to create, assemble, deploy, and debug code in Sun ONE Application Server from a single, easy-to-use interface. Behind the scenes, a plugin integrates Sun ONE Studio with Sun ONE Application Server. For more information about using Sun ONE Studio, see the *Sun ONE Studio 4, Enterprise Edition Tutorial.*

Apache Ant

You can use the automated assembly features available through Ant, a Java-based build tool available through the Apache Software Foundation:

`http://jakarta.apache.org/ant/`

Ant is a java-based build tool that is extended using Java classes. Instead of using shell commands, the configuration files are XML-based, calling out a target tree where tasks get executed. Each task is run by an object that implements a particular task interface.

Apache Ant 1.4.1 is provided with Sun ONE Application Server (or with the operating system for bundled Solaris 9). Sun ONE Application Server also provides server-specific Ant tasks for deployment and administration with the sample applications. For more information about using Ant with Sun ONE Application Server, see "Apache Ant Assembly and Deployment Tool," on page 103.

Migration Tools

The following automated migration tools are downloadable from Sun:

- The Sun ONE Migration Tool for Application Servers reassembles J2EE applications and modules developed on:
  - iPlanet Application Server 6.*x*
  - iPlanet Web Server 6.*x*
  - IBM's Websphere Application Server 4.0
  - BEA Systems' WebLogic Server 6.1
- The Sun ONE Migration Toolbox helps you migrate applications developed on NetDynamics and Netscape Application Servers.

For more information, see *Sun ONE Application Server Migrating and Redeploying Server Applications*.

Profiling Tools

You can use several profilers with Sun ONE Application Server, including HPROF, Optimizeit™, Wily Introscope®, and JProbe™. For more information, see "Profiling," on page 140.

Source Code Control Tools

The following source code control tools are supported through Sun ONE Studio 4:

- Concurrent Versioning System (CVS) - built-in support in Community Edition
- PVCS - predefined configuration
- Visual Source Safe - predefined configuration
- VCS - can be integrated using API
- RCS - can be integrated using API
- SCCS - can be integrated using API
- Clearcase - can be integrated using API

For more information, see:

`http://wwws.sun.com/software/sundev/jde/features/ce-features.html`

Other Tools Supported Through Sun ONE Studio

For a list of other developer tools supported through Sun ONE Studio 4, see:

`http://forte.sun.com/ffj/partnerprograms/partnerlist.html`

Steps for Creating Components

Before creating J2EE applications and components, you should read “Best Practices for Designing J2EE Applications,” on page 24.

This section covers the basic steps for the following:

- Creating Web Applications
- Creating Enterprise JavaBeans
- Creating ACC Clients
- Creating Connectors
- Creating Complete Applications

Creating Web Applications

To create a web application:

1. Create a directory for all the web application’s files. This is the web application’s document root.
2. Create any needed HTML files, image files, and other static content. Place these files in the document root directory or a subdirectory where they can be accessed by other parts of the application.
3. Create any needed JSP files.
4. Create any needed servlets.
5. Compile the servlets. You can also precompile the JSPs.
6. Create the `WEB-INF` directory and the other structural requirements of a web application.

7. Create the deployment descriptor files, `web.xml` and optionally `sun-web.xml`, in the `WEB-INF` directory. It is a good idea to verify the structure of these files as described in "The Deployment Descriptor Verifier," on page 83.
8. Package the web application in a WAR file. If you are using directory deployment, this is optional.
9. Deploy the web application by itself or include it in a J2EE application.

For details about all these steps, see the *Sun ONE Application Server Developer's Guide to Web Application*s.

Creating Enterprise JavaBeans

To create an EJB component:

1. Create a directory for all the EJB component's files.
2. Decide on the type of EJB component you are creating:
   - Session
     - stateful
     - stateless
   - Entity
     - with bean-managed persistence
     - with container-managed persistence
   - Message-Driven
3. Write the code for the EJB component according to the EJB specification, including:
   - A local and/or remote home interface
   - A local and/or remote interface
   - An implementation class (for a message-driven bean, this is all you need)
4. Compile the interfaces and classes.
5. Create the `META-INF` directory and the other structural requirements of an EJB component.

6. Create the deployment descriptor files, `ejb-jar.xml` and `sun-ejb-jar.xml`, in the `META-INF` directory. If the EJB component is an entity bean with container-managed persistence, you must also create a `.dbschema` file and a `sun-cmp-mapping.xml` file. It is a good idea to verify the structure of these files as described in "The Deployment Descriptor Verifier," on page 83.
7. Package the EJB component in a JAR file. If you are using directory deployment, this is optional.
8. Deploy the EJB component by itself or include it in a J2EE application.

For details about all these steps, see the *Sun ONE Application Server Developer's Guide to Enterprise JavaBeans Technology*.

Creating ACC Clients

To create an ACC client:

1. Create a directory for all the client's files.
2. Create the code for the client's classes according to the Java 2 Platform Enterprise Edition Specification.
3. Compile the client's interfaces and classes.
4. Create the `META-INF` directory and the other structural requirements of an ACC client.
5. Create the deployment descriptor files, `application-client.xml` and `sun-application-client.xml`, in the `META-INF` directory. It is a good idea to verify the structure of these files as described in "The Deployment Descriptor Verifier," on page 83.
6. Package the client in a JAR file. If the client communicates with one or more EJB components and you are using directory deployment, this is optional.
7. If the client communicates with one or more EJB components, package the client and EJB components together in an application, then deploy the application.
8. Prepare the client machine:
    a. Create the ACC package JAR file.
    b. Copy the ACC package JAR file to the client machine and unjar it.
    c. Configure the `sun-acc.xml` and `asenv.conf` (`asenv.bat` on Windows) files.

d. Copy the client JAR to the client machine.

9. Execute the client.

For details about all these steps, see the *Sun ONE Application Server Developer's Guide to Clients*.

Creating Connectors

To create a connector:

1. Create a directory for all the connector's files.

2. Create the code for the resource adapter classes (`ConnectionFactory`, `Connection`, and so on) according to the J2EE Connector Architecture Specification.

3. Compile the connector's interfaces and classes.

4. Create one or more JAR files that contain all the connector's classes.

5. Add any native libraries needed by the connector to the directory structure.

6. Create the `META-INF` directory and the other structural requirements of a connector. Here is an example of the overall directory structure:

```
+ MyConnector/
|--- readme.html
|--- ra.jar
|--- client.jar
|--- win.dll
|--- solaris.so
'--+ META-INF/
   |--- MANIFEST.MF
   |--- ra.xml
   '--- sun-ra.xml
```

The `ra.jar` and `client.jar` files were created in Step 4. The `win.dll` and `solaris.so` files are native libraries that were added in Step 5.

7. Create the deployment descriptor files, `ra.xml` and `sun-ra.xml`, in the `META-INF` directory. Both are required. It is a good idea to verify the structure of these files as described in "The Deployment Descriptor Verifier," on page 83.

8. Create a RAR file from the directory structure described in Step 6. If you are using directory deployment, this is optional.
9. Deploy the connector by itself or include it in a J2EE application.

For details about all these steps, see the *Sun ONE J2EE CA Service Provider Implementation Administrator's Guide*.

NOTE You can use the Sun ONE Connector Builder to make building connectors easier. For more information, see the *Sun ONE Connector Builder Developer's Guide.*

Creating Complete Applications

To create a complete J2EE application:

1. Decide on the components (web applications, EJB components, and connectors) that the application will comprise.
2. Create a directory for all the application's files, and create a the `META-INF` directory under it.
3. Create the components of the application and copy them into the application directory. Each component must be in an open subdirectory named by changing `.jar`, `.war`, or `.rar` to `_jar`, `_war`, or `_rar`, respectively.
4. Make sure the components call each other properly.
5. Create the deployment descriptor files, `application.xml` and optionally `sun-application.xml`, in the `META-INF` directory under the application directory. It is a good idea to verify the structure of these files as described in "The Deployment Descriptor Verifier," on page 83.
6. Package the application in an EAR file. If you are using directory deployment, this is optional.
7. Deploy the application.

For details about all these steps, see Chapter 4, "Assembling and Deploying J2EE Applications."

Chapter 3

Securing J2EE Applications

This chapter describes how to write secure J2EE applications, which contain components that perform user authentication and access authorization for servlets and EJB business logic.

For information about administrative security for the server, see the *Sun ONE Application Server Administrator's Guide to Security*.

This chapter contains the following sections:

- Sun ONE Application Server Security Goals
- Sun ONE Application Server Specific Security Features
- Sun ONE Application Server Security Model
- Security Responsibilities Overview
- Common Security Terminology
- Container Security
- Guide to Security Information
- Realm Configuration
- The server.policy File
- Programmatic Login

Sun ONE Application Server Security Goals

In an enterprise computing environment there are many security risks. The Sun ONE Application Server's goal is to provide highly secure, interoperable, and distributed component computing based on the J2EE security model. The security goals for the Sun ONE Application Server include:

- Full compliance with the J2EE security model (for more information, see the J2EE specification, v1.3 Chapter 3 Security)
- Full compliance with the EJB v2.0 security model (for more information, see the Enterprise JavaBean specification v2.0 Chapter 15 Security Management). This includes EJB role-based authorization.
- Full compliance with the Java Servlet v2.3 security model (for more information, see the Java Servlet specification, v2.3 Chapter 11 Security). This includes servlet role-based authorization.
- Support for single sign-on across all Sun ONE Application Server applications within a single security domain.
- Security support for ACC Clients.
- Support for several underlying authentication realms, such as simple file and LDAP. Certificate authentication is also supported for SSL client authentication. For Solaris, OS platform authentication is supported in addition to these.
- Support for declarative security via Sun ONE Application Server specific XML-based role mapping.

Sun ONE Application Server Specific Security Features

The Sun ONE Application Server supports the J2EE v1.3 security model, as well as the following features which are specific to the Sun ONE Application Server:

- Single sign-on across all Sun ONE Application Server applications within a single security domain
- Programmatic login

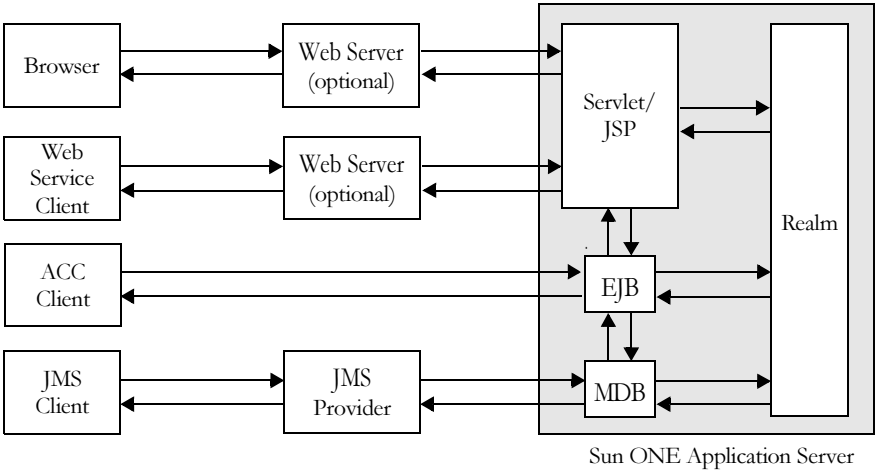
Sun ONE Application Server Security Model

Secure applications require a client to be authenticated as a valid application user and have authorization to access servlets, JSPs, and EJB business logic. Sun ONE Application Server supports security for web, ACC, web service, and JMS clients.

Applications with secure web and EJB containers may enforce the following security processes for clients:

- authenticate the caller
- authorize the caller for access to the EJB business methods

The following diagram shows the Sun ONE Application Server security model.



Web Application and URL Authorizations

Secure web applications may have authentication and authorization properties. The web container supports three types of authentication: basic, certificate, and form-based. When a browser requests the main application URL, the web container collects the user authentication information (for example, username and password) and passes it to the security service for authentication.

Sun ONE Application Server consults the security policies (derived from the deployment descriptors) associated with the web resource to determine the security roles used to permit resource access. The web container tests the user credentials against each role to determine if it can map the user to the role. For more information, see the *Sun ONE Application Server Developer's Guide to Web Applications*.

Invocation of Enterprise Bean Methods

Once the browser client has been authenticated and authorized by the web container and the servlet or JSP performs a method call to the EJB component, the user's credentials (gathered during the authentication process) are propagated to the EJB container. A secure EJB container has a deployment descriptor with authorization properties, which are used to enforce access control on the bean method. The EJB container uses role information received from the EJB JAR deployment descriptors to decide whether it can map the caller to the role and allow access to the bean method. For more information, see the *Sun ONE Application Server Developer's Guide to Enterprise JavaBeans Technology*.

ACC Client Invocation of Enterprise Bean Methods

For ACC clients, a secure EJB container consults its security policies (obtained from the deployment descriptors) to determine if the caller has the authority to access the bean method. This process is the same for both web and ACC clients. For more information, see the *Sun ONE Application Server Developer's Guide to Clients*.

Security Responsibilities Overview

A J2EE platform's primary goal is to isolate the developer from the security mechanism details and facilitate a secure application deployment in diverse environments. This goal is addressed by providing mechanisms for the application security specification requirements declaratively and outside the application.

Application Developer

The application developer is responsible for the following:

- Specifying application roles.
- Defining role-based access restrictions for the application components (Servlets / JSPs and EJB components).
- If programmatic security is used, verifying the user roles and authorizing access to features based on these roles. (Programmatic security management is discouraged since it hard codes the security login in the application instead of allowing the containers to manage it.)

Application Assembler

The application assembler or application component provider must identify all security dependencies embedded in a component including:

- All role names used by the components that call `isCallerInRole` or `isUserInRole`.
- References to all external resources accessed by the components.
- References to all intercomponent calls made by the component.

Application Deployer

The application deployer takes all component security views provided by the assembler and uses them to secure a particular enterprise environment in the application, including:

- Assigning users or groups (or both) to security roles.
- Refines the privileges required to access component methods to suit the requirements of the specific deployment scenario.

Common Security Terminology

The most common security processes are authentication, authorization, realm assignment, and role mapping. The following sections define this terminology.

Authentication

Authentication verifies the user. For example, the user may enter a username and password in a web browser, and if those credentials match the permanent profile stored in the active realm, the user is authenticated. The user is associated with a security identity for the remainder of the session.

For more information about authentication in the Sun ONE Application Server, see the *Sun ONE Application Server Developer's Guide to Web Applications* and the *Sun ONE Application Server Developer's Guide to Clients.*

Authorization

Authorization permits a user to perform the desired operations, after being authenticated. For example, a human resources application may authorize managers to view personal employee information for all employees, but allow employees to only view their own personal information.

For more information about authorization in the Sun ONE Application Server, see the *Sun ONE Application Server Developer's Guide to Web Applications* and the *Sun ONE Application Server Developer's Guide to Enterprise JavaBeans Technology*.

Realms

A *realm*, also called a *security policy domain* or *security domain* in the J2EE specification, is a scope over which a common security policy is defined and enforced by the security administrator of the security service. Supported realms in Sun ONE Application Server are `file`, `ldap`, `certificate`, and `solaris`. For information about how to configure a realm, see "Realm Configuration," on page 48.

Role Mapping

A client may be defined in terms of a security role. For example, a company might use its employee database to generate both a company wide phone book application and to generate payroll information. Obviously, while all employees might have access to phone numbers and email addresses, only some employees would have access to the salary information. Employees with the right to view or change salaries might be defined as having a special security role.

A role is different from a user group in that a role defines a function in an application, while a group is a set of users who are related in some way. For example, members of the groups *astronauts*, *scientists*, and (occasionally) *politicians* all fit into the role of *SpaceShuttlePassenger*.

The EJB security model describes roles (as distinguished from user groups) as being described by an application developer and independent of any particular domain. Groups are specific to a deployment domain. It is up to the deployer to map roles into one or more groups for each application or module.

In the Sun ONE Application Server, roles correspond to users or groups (or both) configured in the active realm.

Container Security

The component containers are responsible for providing J2EE application security. There are two security forms provided by the container:

- Programmatic security
- Declarative security

Programmatic Security

Programmatic security is when an EJB component or servlet uses method calls to the security API, as specified by the J2EE security model, to make business logic decisions based on the caller or remote user's security role. Programmatic security should only be used when declarative security alone is insufficient to meet the application's security model.

The J2EE specification, v1.3 defines programmatic security as consisting of two methods of the EJB `EJBContext` interface and two methods of the servlet `HttpServletRequest` interface. The Sun ONE Application Server supports these interfaces as specified in the specification.

For more information on programmatic security, see the following:

- Section 3.3.6, Programmatic Security, in the J2EE Specification,v1.3
- “Programmatic Login,” on page 60

Declarative Security

Declarative security means that the security mechanism for an application is declared and handled externally to the application. Deployment descriptors describe the J2EE application’s security structure, including security roles, access control, and authentication requirements.

The Sun ONE Application Server supports the DTDs specified by J2EE v1.3 and has additional security elements included in its own deployment descriptors. Declarative security is the application deployer’s responsibility. For more information, see Chapter 4, “Assembling and Deploying J2EE Applications.”

There are three levels of declarative security, as follows:

- Application Level Security
- Web Component Level Security
- EJB Level Security

Application Level Security

The application XML deployment descriptor (`sun-application.xml`) contains authorization descriptors for all user roles for accessing the application’s servlets and EJB components. On the application level, all roles used by any application container must be listed in a `role-name` element in this file. The role names are scoped to the EJB XML deployment descriptors (`ejb-jar.xml` and `sun-ejb-jar.xml` files) and to the servlet XML deployment descriptors (`web.xml` and `sun-web.xml` files). The `sun-application.xml` file must also contain matching `security-role-mapping` elements for each `role-name` used by the application.

Web Component Level Security

A secure web container authenticates users and authorizes access to a servlet or JSP by using the security policy laid out in the servlet XML deployment descriptors (`web.xml` and `sun-web.xml` files). Once the user has been authenticated and authorized, the servlet passes on user credentials to an EJB component to establish a secure association with the bean.

EJB Level Security

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

Guide to Security Information

Each information type below is shown with a short description, the location where the information resides, how to create the information, how to access the information, and where to look for further information.

- User Information
- Security Roles

User Information

User name, password, and so on.

Location:

The location of the user information depends on the realm being used:

- For the `file` realm, the users and groups are listed in the key file, which is located in the *instance_dir*`/config` directory.
- For the `ldap` realm, the users and groups are stored in an external LDAP directory.
- For the `certificate` realm, the user identities are obtained from cryptographically verified client certificates.
- For the `solaris` realm, the users and groups are stored in the underlying Solaris user database, as determined by the system's PAM configuration.

For more information about these realms, see "Realm Configuration," on page 48.

How to Create:

How to create users and define groups is specific to the realm being used. The Sun ONE Application Server does not provide administration capabilities for external realms such as Solaris / PAM or LDAP. Consult your Solaris or LDAP server documentation for details.

Security Roles

Role that defines an application function, made up of a number of users, groups, or both. The relationship between users and groups is determined by the specific realm implementation being used.

Location:
Roles are defined in the J2EE application deployment descriptors.

How to Create:
Use the Sun ONE Application Server Administration interface or the Sun ONE Studio 4 development tools for application assembly and deployment.

How To Access:
Use `isCallerInRole()` to test for a user's role membership. For example, in the following code, if `securedMethod()` can be accessed by the `Manager` role, the call to `sctx.isCallerInRole("Manager")` returns `true`.

```
public class SecTestEJB implements SessionBean
{
   private SessionContext sctx = null;

   public void setSessionContext(SessionContext sc)
   {
      sctx = sc;
   }

   public void securedMethod( )
   {
      System.out.println(  sctx.isCallerInRole( "Manager" ) );
   }
}
```

Realm Configuration

This section covers the following topics:

- How to Configure Realms
- Supported Realms

How to Configure Realms

You can configure realms in one of these ways:

- Using the Administration Interface
- Using the asadmin Command
- Editing the server.xml File

Using the Administration Interface

To configure a realm using the Administration interface:

1. Open the Security component under your server instance.
2. Open the Realms component under the Security component.
3. Go to the Realms page.
4. Click on the check boxes of the realms you wish to activate. You can also use the New and Delete buttons to add and remove realms. To edit a realm, click on it.
5. If you are adding or editing a realm, enter the realm's name, classname, properties, and users (`file` realm only), then select the Save button.
6. Go to the Security page.
7. Select a default realm, then select the Save button.
8. Go to the server instance page and select the Apply Changes button.
9. Restart the server.

Using the asadmin Command

You can use the `asadmin` command to configure realms on local servers.

asadmin create-auth-realm

The `asadmin create-auth-realm` command configures a realm. The syntax is as follows:

`asadmin create-auth-realm --user` *admin_user* `[--password` *admin_password*`]` `[--passwordfile` *password_file*`] [--host` *hostname*`] [--port` *adminport*`]` `[--secure | -s] [--instance` *instance_name*`] --classname` *realm_class* `[--property (`*name*`=`*value*`)[:`*name*`=`*value*`]*]` *realm_name*

For example, the following command configures the `certificate` realm:

```
asadmin create-auth-realm --user jadams --password secret --instance
server1 --classname
com.iplanet.ias.security.auth.realm.certificate.CertificateRealm
certificate
```

asadmin delete-auth-realm

The `asadmin delete-auth-realm` command deactivates a realm. The syntax is as follows:

`asadmin delete-auth-realm --user` *admin_user* `[--password` *admin_password*`]` `[--passwordfile` *password_file*`]` `[--host` *hostname*`]` `[--port` *adminport*`]` `[--secure | -s]` `[--instance` *instance_name*`]` *realm_name*

For example, the following command deactivates the `certificate` realm:

```
asadmin delete-auth-realm --user jadams --password secret --instance
server1 certificate
```

asadmin list-auth-realms

The `asadmin list-auth-realms` command lists all realms configured for a server instance. The syntax is as follows:

`asadmin list-auth-realms --user` *admin_user* `[--password` *admin_password*`]` `[--passwordfile` *password_file*`]` `[--host` *hostname*`]` `[--port` *adminport*`]` `[--secure | -s]` *instance_name*

For example, the following command lists all realms in the `server1` instance:

```
asadmin list-auth-realms --user jadams --password secret server1
```

Editing the server.xml File

Behind the scenes, the default realm is set in the `security-service` element in the `server.xml` file. The `security-service` configuration looks like this:

```
<security-service default-realm="file" anonymous-role="ANYONE"
       audit-enabled="false">
   <auth-realm name="file"
          classname="com.iplanet.ias.security.auth.realm.file.FileRealm">
      <property name="file" value="instance_dir/config/keyfile"/>
      <property name="jaas-context" value="fileRealm"/>
   </auth-realm>
   ...
</security-service>
```

The `default-realm` attribute points to the realm the server is using. It must point to one of the configured `auth-realm` names. The default is the `file` realm.

The audit flag determines whether auditing information is logged. If set to `true`, the server logs audit messages for all authentication and authorization events.

If you change the realm configuration, you must restart the server for the change to take effect.

For more information about the `server.xml` file, see the *Sun ONE Application Server Administrator's Configuration File Reference*.

Supported Realms

The following realms are supported in Sun ONE Application Server:

- `file`
- `ldap`
- `certificate`
- `solaris`
- Creating a Custom Realm

file

The `file` realm is the default realm when you first install the Sun ONE Application Server. It has the following configuration characteristics:

- Name - `file`
- Classname - `com.iplanet.ias.security.auth.realm.file.FileRealm`

Required properties are as follows:

- `file` - The name of the file that stores user information. By default this file is *instance_dir*`/config/keyfile`.
- `jaas-context` - The value must be `fileRealm`.

The user information file is initially empty, so you must add users before you can use the `file` realm. You can configure users in one of these ways:

- Using the Administration Interface
- Using the asadmin Command

Using the Administration Interface

To configure a user in the `file` realm using the Administration interface:

1. Open the Security component under your server instance.
2. Open the Realms component under the Security component.
3. Go to the file page.
4. Click on the Manage Users button.
5. To add a new user, click on the New button. To modify information for a user, click on the user's name in the list. In either case, enter the following information:
   - User ID (required if new) - The name of the user.
   - Password (required) - The user's password.
   - Retype Password (required) - The user's password again, for verification.
   - Group List (optional) - A comma-separated list of the groups the user belongs to.
6. Click on the OK button.
7. Go to the server instance page and select the Apply Changes button.
8. Restart the server.

Using the asadmin Command

The `asadmin create-file-user` command creates one user in the `file` realm. Its syntax is as follows, with defaults shown for optional parameters that have them:

`asadmin create-file-user --user` *admin_user* `[--password` *admin_password*`]` `[--passwordfile` *password_file*`] [--host localhost] [--port 4848]` `[--secure | -s] [--instance` *instance_name*`] [--userpassword` *user_password*`]` `[--groups` *user_group*`[:`*user_group*`]*]` *user_name*

For example, the following command adds user `dsanchez` to the `file` realm and assigns `secret` as this user's password. Note that `jadams` and `topsecret` are the administrator's name and password, respectively.

```
asadmin create-file-user --user jadams --password topsecret
--userpassword secret dsanchez
```

The `asadmin delete-file-user` command removes one user from the `file` realm. Its syntax is as follows, with defaults shown for optional parameters that have them:

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

For example, the following command removes user dsanchez from the file realm. Note that jadams and topsecret are the administrator's name and password, respectively.

```
asadmin delete-file-user --user jadams --password topsecret dsanchez
```

The asadmin update-file-user command changes information for one user in the file realm. Its syntax is as follows, with defaults shown for optional parameters that have them:

```
asadmin update-file-user --user admin_user [--password admin_password]
[--passwordfile password_file] [--host localhost] [--port 4848]
[--secure | -s] [--instance instance_name] [--userpassword user_password]
[--groups user_group[:user_group]*] user_name
```

For example, the following command changes the password for user dsanchez to private. Note that jadams and topsecret are the administrator's name and password, respectively.

```
asadmin update-file-user --user jadams --password topsecret
--userpassword private dsanchez
```

The asadmin list-file-users command lists users in the file realm. Its syntax is as follows, with defaults shown for optional parameters that have them:

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

For example, the following command lists file realm users for the server1 instance. Note that jadams and topsecret are the administrator's name and password, respectively.

```
asadmin list-file-users --user jadams --password topsecret server1
```

The asadmin list-file-groups command lists groups in the file realm. Its syntax is as follows, with defaults shown for optional parameters that have them:

```
asadmin list-file-groups --user admin_user [--password admin_password]
[--passwordfile password_file] [--host localhost] [--port 4848]
[--secure | -s] [--name user_name] instance_name
```

For example, the following command lists file realm groups for user dsanchez. Note that jadams and topsecret are the administrator's name and password, respectively.

```
asadmin list-file-users --user jadams --password topsecret --name
dsanchez server1
```

ldap

The `ldap` realm allows you to use an LDAP database for user security information. It has the following configuration characteristics:

- Name - `ldap`
- Classname - `com.iplanet.ias.security.auth.realm.ldap.LDAPRealm`

Required properties are as follows:

- `directory` - The LDAP URL to your server.
- `base-dn` - The base DN for the location of user data. This base DN can be at any level above the user data, since a tree scope search is performed. The smaller the search tree, the better the performance.
- `jaas-context` - The value must be `ldapRealm`.

You can add the following optional properties to tailor the LDAP realm behavior.

- `search-filter` - The search filter to use to find the user. The default is `uid=%s` (`%s` expands to the subject name).
- `group-base-dn` - The base DN for the location of groups data. By default it is same as the `base-dn`, but it can be tuned if necessary.
- `group-search-filter` - The search filter to find group memberships for the user. The default is `uniquemember=%d` (`%d` expands to the user element DN).
- `group-target` - The LDAP attribute name that contains group name entries. The default is `CN`.
- `search-bind-dn` - An optional DN used to authenticate to the directory for performing the `search-filter` lookup. Only required for directories that do not allow anonymous search.
- `search-bind-password` - The LDAP password for the DN given in `search-bind-dn`.

You must create the desired user(s) in your LDAP directory. You can do this from the Sun ONE Directory Server console in the Users & Groups main tab, or through any other administration tool which supports LDAP and your directory's schema.

The `principal-name` used in the deployment descriptors must correspond to your LDAP user information.

certificate

The `certificate` realm supports SSL authentication. The `certificate` realm sets up the user identity in the Sun ONE Application Server's security context and populates it with user data from the client certificate. The J2EE containers then handle authorization processing based on each user's DN from his or her certificate. It has the following configuration characteristics:

- Name - `certificate`
- Classname - `com.iplanet.ias.security.auth.realm.certificate.CertificateRealm`

You can add the following optional property to tailor the certificate realm behavior.

- `assign-groups` - If this property is set, its value is taken to be a comma-separated list of group names. All clients who present valid certificates are assigned membership to these groups for the purposes of authorization decisions in the web and EJB containers.

When you deploy an application, you must specify `CLIENT-CERT` as the authentication mechanism in the `web.xml` file as follows:

```
<login-config>
    <auth-method>CLIENT-CERT</auth-method>
</login-config>
```

You must obtain a client certificate and install it in your browser to complete the setup for client certificate authentication. For details on how to set up the server and client certificates, see the *Sun ONE Application Server Administrator's Guide to Security*.

You can configure the server instance for SSL authentication in these ways:

- Configure an `ssl` element in `server.xml`, then restart the server. For more information about the `server.xml` file, see the *Sun ONE Application Server Administrator's Configuration File Reference.*
- Use the Administration interface, as follows:
    a. Open the HTTP Server component under your server instance.
    b. Open the HTTP Listeners component under the HTTP Server component.
    c. Select the listener for which you want to configure SSL.
    d. Edit the SSL / TLS Settings for the listener.
    e. Select the Save button.

f. Go to the server instance page and select the Apply Changes button.

g. Restart the server.

For more information, see the *Sun ONE Application Server Administrator's Guide to Security*.

solaris

The `solaris` realm allows authentication using Solaris `username+password` data. This realm is only supported on Solaris 9. It has the following configuration characteristics:

- Name - `solaris`
- Classname - `com.iplanet.ias.security.auth.realm.file.SolarisRealm`

Required properties are as follows:

- `jaas-context` - The value must be `solarisRealm`.

NOTE The Solaris realm invokes the underlying PAM infrastructure for authenticating. If the configured PAM modules require root privileges, the instance must run as root to use this realm. For details, see the "Using Authentication Services (Tasks)" chapter in the *Solaris 9 Systems Administrators Guide: Security Services*.

Solaris supports an enhanced method for controlling access to resources called Role Based Access Control (RBAC). Sun ONE Application Server includes RBAC facilities for its administrative commands. For details, see the "Role-Based Access Control (Overview)" chapter in the *Solaris 9 Systems Administrators Guide: Security Services*.

Creating a Custom Realm

You can create a custom realm by providing a Java Authentication and Authorization Service (JAAS) login module and a realm implementation. Note that client-side JAAS login modules are not suitable for use with Sun ONE Application Server. For more information about JAAS, refer to the JAAS specification for Java 2 SDK, v 1.4, available here:

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

A sample application that uses a custom realm is available with the Sun ONE Application Server here:

install_dir`/samples/security/realms`

The server.policy File

Each Sun ONE Application Server instance has its own standard J2SE policy file, located in the *instance_dir*`/config` directory. The file is named `server.policy`.

Sun ONE Application Server 7 is a J2EE 1.3-compliant application server. As such, it follows the recommendations and requirements of the J2EE specification, including the presence of the security manager (the Java component that enforces the policy) and a limited permission set for J2EE application code.

This section covers the following topics:

- Default Permissions
- Changing Permissions for an Application
- Disabling the Security Manager

Default Permissions

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

Application permissions are granted in the default grant block. These permissions apply to all code not part of the internal server code listed previously. Sun ONE Application Server 7 does not distinguish between EJB and web module permissions. All code is granted the minimal set of Web component permissions (which is a superset of the EJB minimal set).

A few permissions above the minimal set are also granted in the default `server.policy` file. These are necessary due to various internal dependencies of the server implementation. J2EE application developers must not rely on these additional permissions.

One additional permission is granted specifically for using connectors. If connectors are not used in a particular server instance, removal of this permission is recommended, because it is not otherwise necessary.

Changing Permissions for an Application

The default policy for each instance limits the permissions of J2EE deployed applications to the minimal set of permissions required for these applications to operate correctly. If you develop applications that require more than this default set of permissions, you can edit the `server.policy` file to add the custom permissions that your applications need.

You should add the extra permissions only to the applications that require them, not to all applications deployed to a server instance. Do not add extra permissions to the default set (the grant block with no codebase, which applies to all code). Instead, add a new grant block with a codebase specific to the application requiring the extra permissions, and only add the minimally necessary permissions in that block.

NOTE Do not add `java.security.AllPermission` to the `server.policy` file for application code. Doing so completely defeats the purpose of the security manager, yet you still get the performance overhead associated with it. Disable the security manager instead, as described in "Disabling the Security Manager," on page 59.

As noted in the J2EE specification, an application should provide documentation of the additional permissions it needs. If an application requires extra permissions but does not document the set it needs, contact the application author for details.

As a last resort, you can iteratively determine the permission set an application needs by observing `AccessControlException` occurrences in the server log. If this is not sufficient, you can add the `-Djava.security.debug=all` JVM option to the server instance. For details, see the *Sun ONE Application Server Administrator's Guide* or the *Sun ONE Application Server Administrator's Configuration File Reference.*

You can use the J2SE standard `policytool` or any text editor to edit the `server.policy` file. For more information, see:

`http://java.sun.com/docs/books/tutorial/security1.2/tour2/index.html`

For detailed information about the permissions you can set in the `server.policy` file, see:

`http://java.sun.com/j2se/1.4/docs/guide/security/permissions.html`

The Javadoc for the `Permission` class is here:

`http://java.sun.com/j2se/1.4/docs/api/java/security/Permission.html`

Disabling the Security Manager

Developers of J2EE application components should not disable the security manager. As noted in the J2EE specification, J2EE application components should be capable of running with the default set, and otherwise a component needs to document its special requirements.

NOTE To ensure that a J2EE application is compliant with the J2EE policy set, it is critical that J2EE developers do not disable the security manager. Sun ONE Application Server 7 provides an option for disabling the security manager, but future releases may not. Applications that fail to comply with the J2EE policy set (and do not document divergences) may fail to run in future releases.

In a production environment, you may be able to safely disable the security manager if:

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

Disabling the security manager may improve performance significantly for some types of applications. To disable the security manager, remove or comment the following entry in the `server.xml` file:

```
<jvm-options>-Djava.security.policy=instance_dir/config/server.policy</jvm-options>
```

For more information about the `server.xml` file, see the *Sun ONE Application Server Administrator's Configuration File Reference.*

Programmatic Login

Programmatic login allows a deployed J2EE application to invoke a login method. If the login is successful, a `SecurityContext` is established as if the client had authenticated using any of the conventional J2EE mechanisms.

Programmatic login is useful for an application that has special needs which cannot be accommodated by any of the J2EE standard authentication mechanisms.

NOTE Programmatic login is specific to Sun ONE Application Server and not portable to other application servers.

This section contains the following topics:

- Precautions
- Granting Programmatic Login Permission
- The ProgrammaticLogin Class

Precautions

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

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

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

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

Lazy authentication is not supported for programmatic login. If an access check is reached and the deployed application has not properly authenticated via the programmatic login method, access is denied immediately and the application may fail if not properly coded to account for this occurrence.

Granting Programmatic Login Permission

The `ProgrammaticLoginPermission` permission is required to invoke the programmatic login mechanism for an application. This permission is not granted by default to deployed applications because this is not a standard J2EE mechanism.

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

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

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

For more information about the `server.policy` file, see "The server.policy File," on page 57.

The ProgrammaticLogin Class

The `com.sun.appserv.security.ProgrammaticLogin` class enables a user to perform login programmatically. This class has two `login` methods, one for servlets or JSPs and one for EJB components.

The login method for servlets or JSPs has the following signature:

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

The login method for EJB components has the following signature:

```
public Boolean login(String user, String password)
```

Both `login` methods:

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

Chapter 4

Assembling and Deploying J2EE Applications

This chapter describes Sun ONE Application Server modules and how these modules are assembled separately or together in an application. For design considerations that affect assembly, see “Modularizing Applications,” on page 26.

Sun ONE Application Server modules and applications include J2EE standard elements and Sun ONE Application Server specific elements. Only Sun ONE Application Server specific elements are described in detail in this chapter.

The following topics are presented in this chapter:

- Overview of Assembly and Deployment
- Assembling Modules and Applications
- Deploying Modules and Applications
- Apache Ant Assembly and Deployment Tool
- The Application Deployment Descriptor Files

Overview of Assembly and Deployment

Application assembly (also known as packaging) is the process of combining discrete components of an application into a single unit that can be deployed to a J2EE-compliant application server. A package can be classified either as a module or as a full-fledged application. This section covers the following topics:

- Modules
- Applications
- J2EE Standard Descriptors

- Sun ONE Application Server Descriptors
- Naming Standards
- JNDI Naming
- Directory Structure
- Runtime Environments
- Classloaders
- Sample Applications

Modules

A J2EE module is a collection of one or more J2EE components of the same container type (for example, web or EJB) with deployment descriptors of that type. One descriptor is J2EE standard, the other is Sun ONE Application Server specific. Types of J2EE modules are as follows:

- **Web Application Archive (WAR):** A web application is a collection of servlets, HTML pages, classes, and other resources that can be bundled and deployed to several J2EE application servers. A WAR file can consist of the following items: servlets, JSPs, JSP tag libraries, utility classes, static pages, client-side applets, beans, bean classes, and deployment descriptors (`web.xml` and optionally `sun-web.xml`).
- **EJB JAR File:** The EJB JAR file is the standard format for assembling enterprise beans. This file contains the bean classes (home, remote, local, and implementation), all of the utility classes, and the deployment descriptors (`ejb-jar.xml` and `sun-ejb-jar.xml`). If the EJB component is an entity bean with container managed persistence, a `.dbschema` file and a CMP mapping descriptor, `sun-cmp-mapping.xml`, must be included as well.
- **Application Client Container JAR File:** An ACC client is a Sun ONE Application Server specific type of J2EE client. An ACC client supports the standard J2EE Application Client specifications, and in addition, supports direct access to the Sun ONE Application Server. Its deployment descriptors are `application-client.xml` and `sun-application-client.xml`.
- **Resource RAR File:** RAR files apply to J2EE CA connectors. A connector module is like a device driver. It is a portable way of allowing EJB components to access a foreign enterprise system. Each Sun ONE Application Server connector has a J2EE XML file, `ra.xml`. A connector must also have a Sun ONE Application Server deployment descriptor, `sun-ra.xml`.

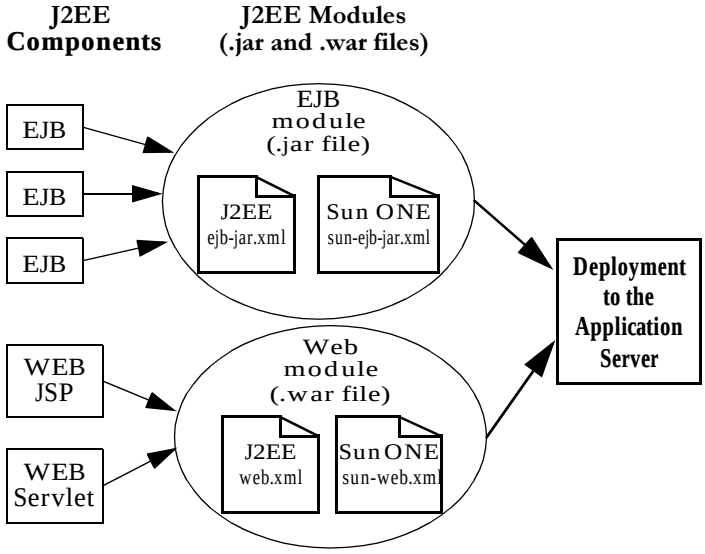
Package definitions must be used in the source code of all modules so the classloader can properly locate the classes after the modules have been deployed.

Because the information in a deployment descriptor is declarative, it can be changed without requiring modifications to source code. At run time, the J2EE server reads this information and acts accordingly.

Sun ONE Application Server also supports lifecycle modules. See Chapter 6, “Developing Lifecycle Listeners” for more information.

EJB JAR and Web modules can also be assembled as separate JAR or WAR files and deployed separately, outside of any application, as in the following figure.

Module Assembly and Deployment

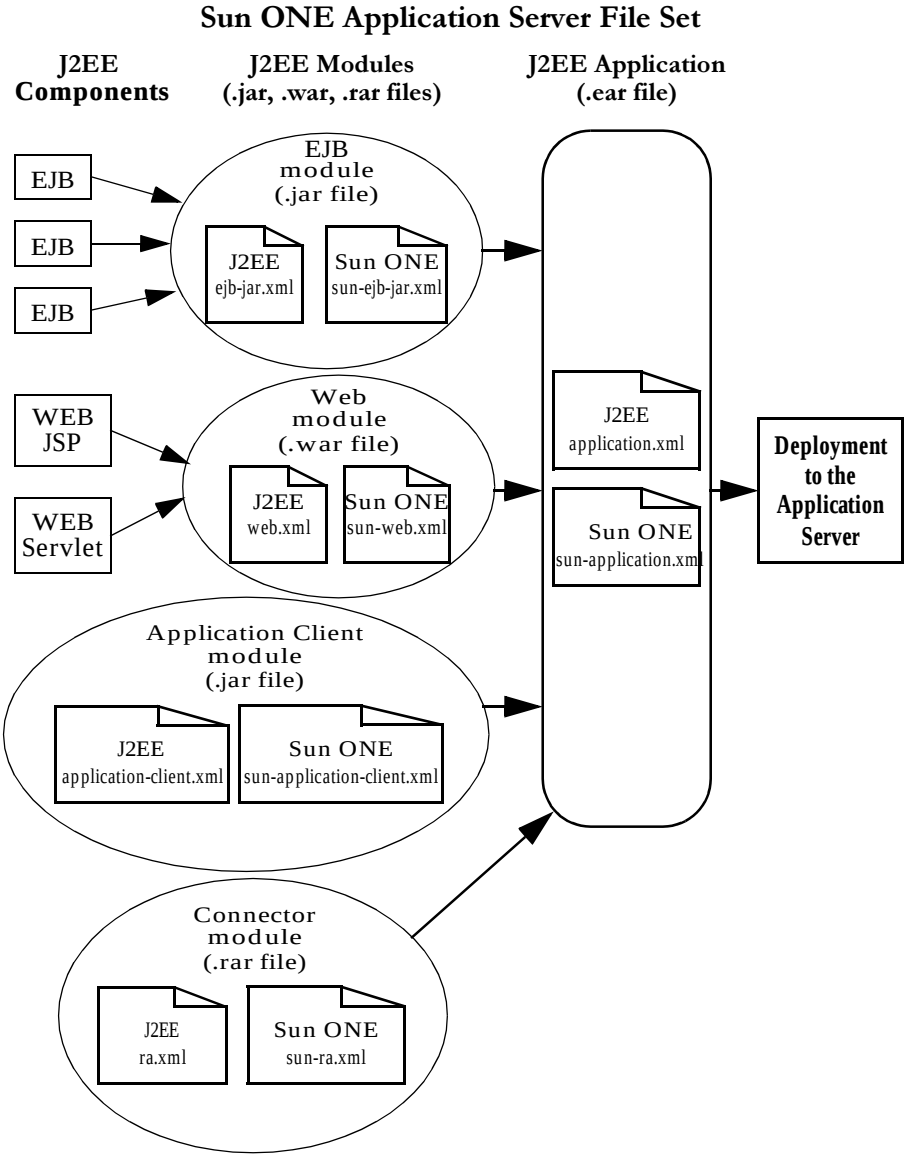


Applications

A J2EE application is a logical collection of one or more J2EE modules tied together by application deployment descriptors. Components can be assembled at either the module or the application level. Components can also be deployed at either the module or the application level.

The following diagram illustrates how components are assembled into modules and then assembled into a Sun ONE Application Server application EAR file ready for deployment.

Application Assembly and Deployment



Each module has a Sun ONE Application Server deployment descriptor and a J2EE deployment descriptor. The Sun ONE Application Server Administration interface uses the deployment descriptors to deploy the application components and to register the resources with the Sun ONE Application Server.

An application consists of one or more modules, an optional Sun ONE Application Server deployment descriptor, and a required J2EE application deployment descriptor. All items are assembled, using the Java ARchive (`.jar`) file format, into one file with an extension of `.ear`.

J2EE Standard Descriptors

The J2EE platform provides assembly and deployment facilities. These facilities use WAR, JAR, and EAR files as standard packages for components and applications, and XML-based deployment descriptors for customizing parameters.

J2EE standard deployment descriptors are described in the J2EE specification, v1.3. You can find the specification here:

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

To check the correctness of these deployment descriptors prior to deployment, see "The Deployment Descriptor Verifier," on page 83.

The following table shows where to find more information about J2EE standard deployment descriptors. The left column lists the deployment descriptors, and the right column lists where to find more information about those descriptors.

J2EE Standard Descriptors

Deployment Descriptor	Where to Find More Information
`application.xml`	Java 2 Platform Enterprise Edition Specification, v1.3, Chapter 8, "Application Assembly and Deployment - J2EE:application XML DTD"
`web.xml`	Java Servlet Specification, v2.3 Chapter 13, "Deployment Descriptor," and JavaServer Pages Specification, v1.2, Chapter 7, "JSP Pages as XML Documents," and Chapter 5, "Tag Extensions"
`ejb-jar.xml`	Enterprise JavaBeans Specification, v2.0, Chapter 16, "Deployment Descriptor"
`application-client.xml`	Java 2 Platform Enterprise Edition Specification, v1.3, Chapter 9, "Application Clients - J2EE:application-client XML DTD"
`ra.xml`	Java 2 Enterprise Edition, J2EE Connector Architecture Specification, v1.0, Chapter 10, "Packaging and Deployment."

Sun ONE Application Server Descriptors

Sun ONE Application Server uses additional deployment descriptors for configuring features specific to the Sun ONE Application Server. The `sun-application.xml` and `sun-web.xml` files are optional; all the others are required.

The DTD schema files for all the Sun ONE Application Server deployment descriptors are located in the *install_dir*`/lib/dtds` directory.

To check the correctness of these deployment descriptors prior to deployment, see "The Deployment Descriptor Verifier," on page 83.

The following table shows where to find more information about Sun ONE Application Server deployment descriptors. The left column lists the deployment descriptors, and the right column lists where to find more information about those descriptors.

Sun ONE Application Server Descriptors

Deployment Descriptor	Where to Find More Information
`sun-application.xml`	"The Application Deployment Descriptor Files," on page 128.
`sun-web.xml`	*Sun ONE Application Server Developer's Guide to Web Applications*
`sun-ejb-jar.xml` and `sun-cmp-mapping.xml`	*Sun ONE Application Server Developer's Guide to Enterprise JavaBeans Technology*
`sun-application-client.xml` and `sun-acc.xml`	*Sun ONE Application Server Developer's Guide to Clients*
`sun-ra.xml`	*Sun ONE J2EE CA Service Provider Implementation Administrator's Guide*

NOTE The Sun ONE Application Server deployment descriptors must have 600 level access privileges on UNIX systems.

Naming Standards

Names of applications and individually deployed EJB JAR, WAR, and connector RAR modules (as specified by the `name` attributes in the `server.xml` file) must be unique in a Sun ONE Application Server instance. If you do not explicitly specify a name, the default name is the first portion of the file name (without the `.war` or `.jar` extension). For details about `server.xml`, see the *Sun ONE Application Server Administrator's Configuration File Reference*.

Modules of different types can have the same name within an application, because when the application is deployed, the directories holding the individual modules are named with `_jar`, `_war` and `_rar` suffixes. Modules of the same type within an application must have unique names. In addition, for entity beans that use CMP, `.dbschema` file names must be unique within an application.

Make sure your package and file names do not contain spaces or characters that are illegal for your operating system.

JNDI Naming

When clients or web applications communicate with EJB components, or when web applications or EJB components require services provided by JDBC or other resources, a *naming service* is what allows these components to locate and talk to each other. A naming service maintains a set of bindings, which relate names to objects. The J2EE naming service is JNDI (the Java Naming and Directory Interface).

In Sun ONE Application Server, containers provide their components a naming environment, or context, which allows components to look up other distributed components and resources. A `Context` object provides the methods for binding names to objects, unbinding names from objects, renaming objects, and listing the bindings.

JNDI 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.

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 application `pkging.ear`.

NOTE To avoid collisions with names of other enterprise resources in JNDI, and to avoid portability problems, all names in a Sun ONE Application Server application should begin with the string `java:comp/env`.

The following table describes JNDI subcontexts for connection factories in Sun ONE Application Server. The left column lists Resource Manager types, the middle column lists Connection Factory types, and the right column lists JNDI subcontexts.

JNDI subcontexts for connection factories

Resource Manager Type	Connection Factory Types	JNDI Subcontext
JDBC	`javax.sql.DataSource`	`java:comp/env/jdbc`
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`

Directory Structure

When you deploy an application, the application is expanded to an open directory structure, and the directories holding the individual modules are named with `_jar`, `_war` and `_rar` suffixes. If you use the `asadmin deploydir` command to deploy a directory instead of an EAR file, your directory structure must follow this same convention.

Module and application directory structures follow the structure outlined in the J2EE specification. Here is an example directory structure of a simple application containing a web module, an EJB module, and a client module.

```
+ converter_1/
|--- converterClient.jar
|--+ META-INF/
|  |--- MANIFEST.MF
|  |--- application.xml
|  '--- sun-application.xml
|--+ war-ic_war/
|  |--- index.jsp
|  |--+ META-INF/
|  |  |--- MANIFEST.MF
|  '--+ WEB-INF/
|     |--- web.xml
|     '--- sun-web.xml
|--+ ejb-jar-ic_jar/
|  |--- Converter.class
|  |--- ConverterBean.class
|  |--- ConverterHome.class
|  '--+ META-INF/
|     |--- MANIFEST.MF
|     |--- ejb-jar.xml
|     '--- sun-ejb-jar.xml
'--+ app-client-ic_jar/
   |--- ConverterClient.class
   '--+ META-INF/
      |--- MANIFEST.MF
      |--- application-client.xml
      '--- sun-application-client.xml
```

Here is an example directory structure of an individually deployed connector module.

```
+ MyConnector/
|--- readme.html
|--- ra.jar
|--- client.jar
|--- win.dll
|--- solaris.so
'--+ META-INF/
   |--- MANIFEST.MF
   |--- ra.xml
   '--- sun-ra.xml
```

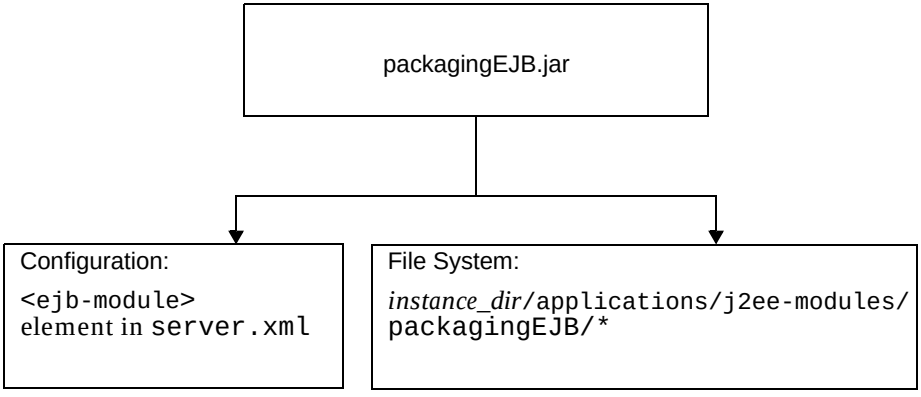
Runtime Environments

Whether you deploy an individual module or an application, deployment affects both the file system and the server configuration. See the following "Module runtime environment" and "Application runtime environment" figures.

Module Runtime Environment

The following figure illustrates the environment for individually deployed module-based deployment.

Module runtime environment



For file system entries, modules are extracted as follows:

instance_dir/applications/j2ee-modules/*module_name*
instance_dir/generated/ejb/j2ee-modules/*module_name*
instance_dir/generated/jsp/j2ee-modules/*module_name*

The applications directory contains the directory structures described in "Directory Structure," on page 71. The generated/ejb directory contains the stubs and ties that an ACC client needs to access the module; the generated/jsp directory contains compiled JSPs.

Lifecycle modules (see Chapter 6, "Developing Lifecycle Listeners") are extracted as follows:

instance_dir/applications/lifecycle-modules/*module_name*

Configuration entries are added in the server.xml file as follows:

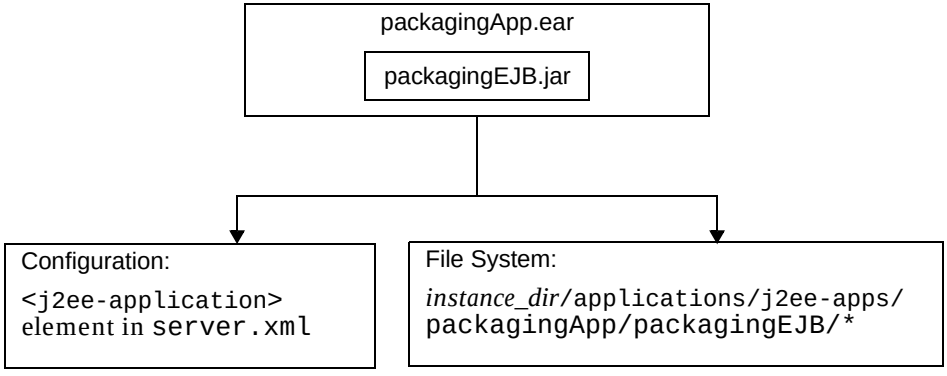
```
<server>
   <applications>
      <type-module>
         ...module configuration...
      </type-module>
   </applications>
</server>
```

The *type* of the module in server.xml can be lifecycle, ejb, web, or connector. For details about server.xml, see the *Sun ONE Application Server Administrator's Configuration File Reference.*

Application Runtime Environment

The following figure illustrates the environment for application-based deployment.

Application runtime environment



For file system entries, applications are extracted as follows:

instance_dir/applications/j2ee-apps/*app_name*
instance_dir/generated/ejb/j2ee-apps/*app_name*
instance_dir/generated/jsp/j2ee-apps/*app_name*

The applications directory contains the directory structures described in "Directory Structure," on page 71. The generated/ejb directory contains the stubs and ties that an ACC client needs to access the module; the generated/jsp directory contains compiled JSPs.

Configuration entries are added in the server.xml file as follows:

```
<server>
    <applications>
        <j2ee-application>
            ...application configuration...
        </j2ee-application>
    </applications>
</server>
```

For details about server.xml, see the *Sun ONE Application Server Administrator's Configuration File Reference*.

Classloaders

Understanding Sun ONE Application Server classloaders can help you determine where and how you can position supporting JAR and resource files for your modules and applications.

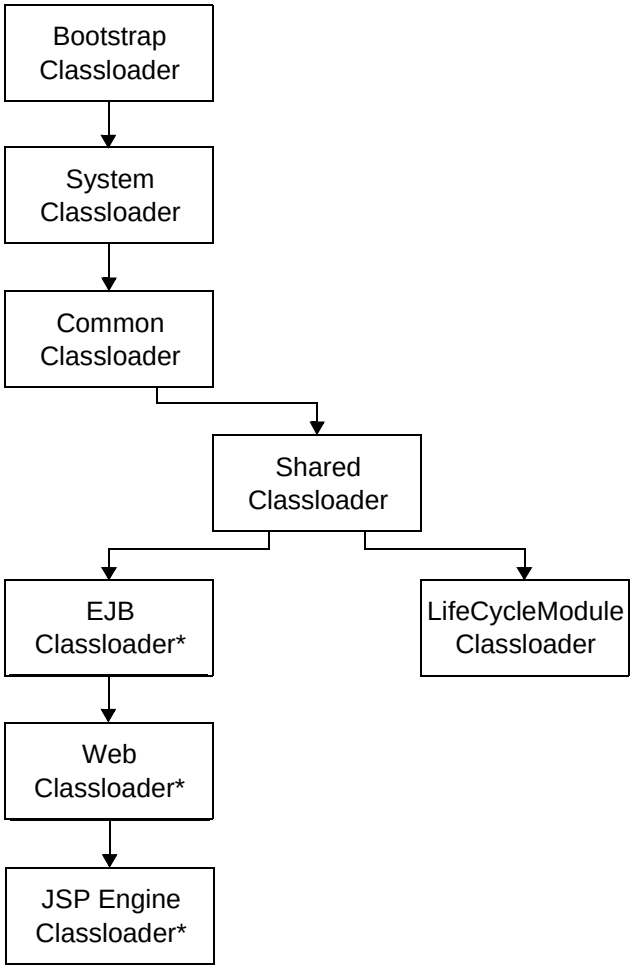
In a Java Virtual Machine (JVM), the classloaders dynamically load a specific java class file needed for resolving a dependency. For example, when an instance of java.util.Enumeration needs to be created, one of the classloaders loads the relevant class into the environment. This section includes the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

- The Classloader Hierarchy
- Classloader Universes
- Circumventing Classloader Isolation

The Classloader Hierarchy

Classloaders in the Sun ONE Application Server runtime follow a hierarchy that is illustrated here:

Classloader runtime hierarchy



*There are separate classloader instances for each application (one of these classloaders is in each application classloader universe).

Note that this is not a Java inheritance hierarchy, but a delegation hierarchy. In the delegation design, a classloader delegates classloading to its parent before attempting to load a class itself. A classloader parent can be either the System Classloader or another custom classloader. If the parent classloader can't load a class, the `findClass()` method is called on the classloader subclass. In effect, a classloader is responsible for loading only the classes not available to the parent.

The exception is the Web Classloader, which follows the delegation model in the Servlet specification. The Web Classloader looks in the local classloader before delegating to its parent. You can make the Web Classloader delegate to its parent first by setting `delegate="true"` in the `class-loader` element of the `sun-web.xml` file. For details, see the *Developer's Guide to Web Applications.*

NOTE The Web Classloader for a web component of a web service must delegate to its parent classloader, so you must set `delegate="true"` in the `class-loader` element of the `sun-web.xml` file in this case.

The following table describes Sun ONE Application Server classloaders. The left column lists the classloaders, and the right column lists descriptions of those classloaders and the files they examine.

Sun ONE Application Server classloaders

Classloader	Description
Bootstrap	The Bootstrap Classloader loads all the JDK classes.
System	The System Classloader loads most of the core Sun ONE Application Server classes. It is created based on the `classpath-prefix`, `server-classpath`, and `classpath-suffix` attributes of the `java-config` element in the `server.xml` file. The environment classpath is included if `env-classpath-ignored="false"` is set in the `java-config` element.
Common	The Common Classloader loads classes in the *instance_dir*`/lib/classes` directory, followed by JAR and ZIP files in the *instance_dir*`/lib` directory. No special classpath settings are required. The existence of these directories is optional; if they don't exist, the Common Classloader is not created.
Shared	The Shared Classloader is a single instance that loads classes (such as individually deployed connector modules) that are shared across all applications.
LifeCycle Module	The LifeCycleModule Classloader is the parent classloader for lifecycle modules. Each lifecycle module's classpath is used to construct its own classloader.

Sun ONE Application Server classloaders

Classloader	Description
EJB	The EJB Classloader loads the enabled EJB classes in a specific enabled EJB module or J2EE application. One instance of this classloader is present in each classloader universe. The EJB Classloader is created with a list of URLs that point to the locations of the classes it needs to load.
Web	The Web Classloader loads the servlets and other classes in a specific enabled web module or J2EE application. One instance of this classloader is present in each classloader universe. The Web Classloader is created with a list of URLs that point to the locations of the classes it needs to load.
JSP Engine	The JSP Engine Classloader loads compiled JSP classes of enabled JSPs. One instance of this classloader is present in each classloader universe. The JSP Engine Classloader is created with a list of URLs that point to the locations of the classes it needs to load.

Classloader Universes

Access to components within applications and modules installed on the server occurs within the context of isolated classloader universes, each of which has its own EJB, Web, and JSP Engine classloaders.

- **Application Universe:** Each J2EE application has its own classloader universe, which loads the classes in all the modules in the application.
- **Individually Deployed Module Universe:** Each individually deployed EJB JAR, web WAR, or lifecycle module has its own classloader universe, which loads the classes in the module.

NOTE In iPlanet Application Server 6.*x*, individually deployed modules shared the same classloader. In Sun ONE Application Server 7, each individually deployed module has its own classloader universe.

NOTE A resource such as a file that is accessed by a servlet, JSP, or EJB component must be in a directory pointed to by the classloader's classpath. For example, the web classloader's classpath includes these directories:

module_name`/WEB-INF/classes`
module_name`/WEB-INF/lib`

If a servlet accesses a resource, it must be in one of these directories or it will not be loaded.

Circumventing Classloader Isolation

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

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

- Using the System Classloader
- Using the Common Classloader
- Packaging the Client JAR for One Application in Another Application

Using the System Classloader

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

- Go to the server instance page in the Administration interface, select the JVM Settings tab, select the Path Settings option, edit the Classpath Suffix field, and select Save.
- Edit the `classpath-suffix` attribute of the `java-config` element in the `server.xml` file. For details about `server.xml`, see the *Sun ONE Application Server Administrator's Configuration File Reference*.

Using the System Classloader makes an application or module accessible to any other application or module across the server instance.

Using the Common Classloader

To use 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.

Using the Common Classloader makes an application or module accessible to any other application or module across the server instance.

Packaging the Client JAR for One Application in Another Application

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

Packaging the client JAR for one application in another application has trade-offs. After you enable the `-nolocalstubs` option, you can deploy multiple applications that contain client JARs of other applications without restarting the server. However, using the `-nolocalstubs` option may degrade server performance.

As an alternative, you can have the Common Classloader load client JAR of the dependent application as described in "Using the Common Classloader," on page 79. Server performance is better, but you must restart the server to make the dependent application accessible, and it is accessible across the server instance. This approach is recommended for a production environment.

To package the client JAR for one application in another application:

1. Add the `-nolocalstubs` option to the server instance's `rmic` options in one of these ways, then restart the server:

   - Go to the server instance page in the Administration interface, select the JVM Settings tab, select the General option, add `-nolocalstubs` to the rmic Options field, and select Save.
   - Add `-nolocalstubs` to the `rmic-options` attribute of the `java-config` element in `server.xml`. For details about `server.xml`, see the *Sun ONE Application Server Administrator's Configuration File Reference.*

   After you make this change, all subsequently deployed EJB components can only be accessed remotely.

NOTE Using the `-nolocalstubs` option may degrade server performance.

2. Deploy the dependent application.

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

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

     Each *filepath* is relative to the directory or JAR file containing the `MANIFEST.MF` file. For details, see the J2EE specification, section 8.1.1.2, "Dependencies."
   - For a calling web component, add the client JAR file under the `WEB-INF/lib` directory.
4. For most applications, packaging the client JAR file with the calling EJB component should suffice. You do not need to package the client JAR file with both the EJB and web components unless the web component is directly calling the EJB component in the dependent application. If you need to package the client JAR with both the EJB and web components, set `delegate="true"` in the `class-loader` element of the `sun-web.xml` file. This changes the Web Classloader so it follows the standard classloader delegation model and delegates to its parent before attempting to load a class itself.
5. Deploy the calling application.

NOTE The calling EJB or web component must use the JNDI name of the EJB component in the dependent application. Using an `ejb-ref` mapping won't work.

Sample Applications

Sample applications that you can examine and deploy are included in Sun ONE Application Server, in the *install_dir*`/samples` directory. The samples are organized in categories such as `ejb`, `jdbc`, `connectors`, `i18n`, and so on. Each sample category is further divided into subcategories. For example, under the `ejb` category are `stateless`, `stateful`, `security`, `mdb`, `bmp`, and `cmp` subcategories.

Most Sun ONE Application Server samples have the following directory structure:

- The `docs` directory contains instructions for how to use the sample.
- The `src` directory contains:
  - Source code

- The `build.xml` file, which defines `asant` targets for the sample (see "Apache Ant Assembly and Deployment Tool," on page 103)
- Deployment descriptors

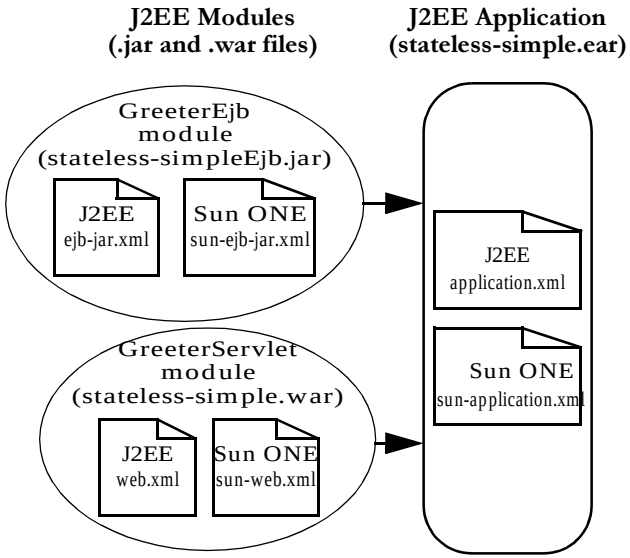
- The `build`, `assemble`, and `javadocs` directories are generated as a result of targets specified in the `build.xml` file.

The *install_dir*`/samples/common.xml` file defines properties common to all sample applications and implements targets needed to compile, assemble, deploy and undeploy sample applications. In most sample applications, the `build.xml` file includes `common.xml`.

NOTE Before using the samples under *install_dir*`/samples/webservices`, make sure to copy the Java XML Pack JAR files into the `jre/lib/endorsed` directory to override the JAR files bundled with the JDK.

The following figure shows the structure of the `helloworld` sample:

The `helloworld` sample



After you deploy the sample in Sun ONE Application Server, you can invoke it using the following URL:

`http://`*server*`:`*port*`/helloworld`

For a detailed description of the sample and how to deploy and run it, see the associated documentation at:

install_dir`/samples/ejb/stateless/simple/docs/index.html`

Assembling Modules and Applications

Assembling (or packaging) modules and applications in Sun ONE Application Server conforms to all of the customary J2EE-defined specifications. The only difference is that when you assemble in Sun ONE Application Server, you include Sun ONE Application Server-specific deployment descriptors (such as `sun-web.xml` and `sun-ejb-jar.xml`) that enhance the functionality of the application server.

This section covers the following topics:

- Tools for Assembly
- Assembling a WAR Module
- Assembling an EJB JAR Module
- Assembling a Lifecycle Module
- Assembling an Application
- Assembling an ACC Client
- Assembling a J2EE CA Resource Adapter

Tools for Assembly

The Sun ONE Application Server provides these methods for assembling a module or an application:

- Apache Ant
- Sun ONE Studio
- The Deployment Descriptor Verifier

Apache Ant

Ant can help you assemble and deploy modules and applications. For details, see "Apache Ant Assembly and Deployment Tool," on page 103.

Sun ONE Studio

You can use Sun ONE Studio 4 to assemble J2EE applications and modules. For more information about using Sun ONE Studio, see the *Sun ONE Studio 4, Enterprise Edition Tutorial.*

The Deployment Descriptor Verifier

The verifier tool validates both J2EE and Sun One Application Server specific deployment descriptors against their corresponding DTD files and gives errors and warnings if a module or application is not J2EE and Sun One Application Server compliant. You can verify deployment descriptors in EAR, WAR, RAR, and JAR files.

The verifier tool is not simply an XML syntax verifier. Rules and interdependencies between various elements in the deployment descriptors are verified. Where needed, user application classes are introspected to apply validation rules.

The verifier is also integrated into the `sun-appserv-deploy` Ant task.

TIP Using the verifier tool can help you avoid runtime errors that are difficult to debug.

This section covers the following topics:

- Command Line Syntax
- Ant Integration
- Sample Results Files

Command Line Syntax

The verifier tool's syntax is as follows:

`verifier` [*options*] *file*

The *file* can be an EAR, WAR, RAR, or JAR file.

The following table shows the *options* for the `verifier` tool. The left column lists options, and the right column lists descriptions of those options.

verifier options

Option	Description
`-v`	Turns on verbose debug mode.
`-d` *output_dir*	Writes test results to the *output_dir*, which must already exist. By default, the results files are created in the system-defined `tmp` directory.
`-ra`	Sets the output report level to display all results. This is the default in both verbose and non verbose modes.
`-rw`	Sets the output report level to display only warning and failure results.
`-rf`	Sets the output report level to display only failure results.

For example, the following command runs the verifier in verbose mode and writes all the results of static verification of the `ejb.jar` file to the output directory `ResultsDir`:

```
verifier -v -ra -d ResultsDir ejb.jar
```

The results files are `ejb.jar_verifier.txt` and `ejb.jar_verifier.xml`.

If the verifier runs successfully, a result code of `0` is returned. This does not mean that no verification errors occurred. A non-zero error code is returned if the verifier fails to run.

Ant Integration

You can integrate the verifier into an Ant build file as a target and use the Ant call feature to call the target each time an application or module is assembled. This is because the `main` method in `com.sun.enterprise.tools.verifier.Verifier` is callable from user Ant scripts. The `main` method accepts the arguments described in the “verifier options” table.

Example code for an Ant verify target is as follows:

```
<target name="verify">
   <echo message="Verification Process for ${testfile}"/>
   <java classname="com.sun.enterprise.tools.verifier.Verifier"
         fork="yes">
      <sysproperty key="com.sunone.enterprise.home"
         value="${appserv.home}"/>
      <sysproperty key="verifier.xsl"
         value="${appserv.home}/verifier/config" />
      <!-- uncomment the following for verbose output -->
      <!--<arg value="-v"/>-->
```

```
      <arg value="${assemble}/${ejbjar}" />
      <classpath path="${appserv.cpath}:${java.class.path}"/>
   </java>
</target>
```

Sample Results Files

Here is a sample results XML file:

```
<static-verification>
   <ejb>
      <failed>
         <test>
            <test-name>
tests.ejb.session.TransactionTypeNullForContainerTX
            </test-name>
            <test-assertion>
Session bean with bean managed transaction demarcation test
            </test-assertion>
            <test-description>
For [ TheGreeter ] Error: Session Beans [ TheGreeter ] with [ Bean ]
managed transaction demarcation should not have container
transactions defined.
            </test-description>
         </test>
      </failed>
   </ejb>
...
</static-verification>
```

Here is a sample results TXT file:

```
---------------------------
STATIC VERIFICATION RESULTS
---------------------------
---------------------------------
NUMBER OF FAILURES/WARNINGS/ERRORS
---------------------------------

# of Failures : 3
# of Warnings : 6
# of Errors : 0

-----------------------------
RESULTS FOR EJB-RELATED TESTS
-----------------------------
--------------
FAILED TESTS :
--------------
```

```
Test Name : tests.ejb.session.TransactionTypeNullForContainerTX
Test Assertion : Session bean with bean managed transaction
demarcation test
Test Description : For [ TheGreeter ]
Error: Session Beans [ TheGreeter ] with [ Bean ] managed
transaction demarcation should not have container transactions
defined.

...

---------------
PASSED TESTS :
---------------

Test Name : tests.ejb.session.ejbcreatemethod.EjbCreateMethodStatic
Test Assertion : Each session Bean must have at least one non-static
ejbCreate method test
Test Description : For [ TheGreeter ] For EJB Class [
samples.helloworld.ejb.GreeterEJB ] method [ ejbCreate ] [
samples.helloworld.ejb.GreeterEJB ] properly declares non-static
ejbCreate(...) method.

...

-----------
WARNINGS :
-----------

Test Name : tests.ejb.businessmethod.BusinessMethodException
Test Assertion : Enterprise bean business method throws
RemoteException test
Test Description :

Test Name : tests.ejb.ias.beanpool.IASEjbBeanPool
Test Assertion :
Test Description : WARNING [IAS-EJB ejb] : bean-pool should be
defined for Stateless Session and Message Driven Beans

...

---------------------
NOTAPPLICABLE TESTS :
---------------------

Test Name :
tests.ejb.entity.pkmultiplefield.PrimaryKeyClassFieldsCmp
```

```
Test Assertion : Ejb primary key class properly declares all class
fields within subset of the names of the container-managed fields
test.
Test Description : For [ TheGreeter ] class
com.sun.enterprise.tools.verifier.tests.ejb.entity.pkmultiplefield.
PrimaryKeyClassFieldsCmp expected Entity bean, but called with
Session.

Test Name : tests.ejb.entity.ejbcreatemethod.EjbCreateMethodReturn
Test Assertion : Each entity Bean may have zero or more ejbCreate
methods which return primary key type test
Test Description : For [ TheGreeter ] class
com.sun.enterprise.tools.verifier.tests.ejb.entity.ejbcreatemethod.
EjbCreateMethodReturn expected Entity bean, but called with Session
bean.

...

-----------------------------------
RESULTS FOR OTHER XML-RELATED TESTS
-----------------------------------
---------------
PASSED TESTS :
---------------

Test Name : tests.dd.ParseDD
Test Assertion : Test parses the deployment descriptor using a SAX
parser to avoid the dependency on the DOL
Test Description : PASSED [EJB] : [ remote ] and [ home ] tags
present.
PASSED [EJB]: session-type is Stateless.
PASSED [EJB]: trans-attribute is NotSupported.
PASSED [EJB]: transaction-type is Bean.

...
```

Assembling a WAR Module

To assemble a WAR module, follow these steps:

1. Create a working directory, and copy the contents of your web module into it. Make sure it has the structure described in the *Sun ONE Application Server Developer's Guide to Web Applications.*

2. Create two deployment descriptor files with these names: `web.xml` (required) and `sun-web.xml` (optional). For more information about these files, see the *Sun ONE Application Server Developer's Guide to Web Applications*.

TIP The first time, you can assemble the WAR module and create the deployment descriptors using Sun ONE Studio. The resulting WAR file can be extracted to yield the deployment descriptors.

3. Execute this command to create the WAR file:

```
jar -cvf module_name.war *
```

TIP The assembly process can be automated using the Ant tool. To learn more, see "Apache Ant Assembly and Deployment Tool," on page 103.

Assembling an EJB JAR Module

To assemble an EJB JAR module, follow these steps:

1. Create a working directory, and copy the contents of your module into it. Make sure it has the structure described in the *Sun ONE Application Server Developer's Guide to Enterprise JavaBeans Technology*.
2. Create two deployment descriptor files with these names: `ejb-jar.xml` and `sun-ejb-jar.xml` (both required). For more information about these files, see the *Sun ONE Application Server Developer's Guide to Enterprise JavaBeans Technology*. If the EJB component is an entity bean with container-managed persistence, you must also create a `.dbschema` file and a `sun-cmp-mapping.xml` file.

TIP The first time, you can assemble the EJB JAR module and create the deployment descriptors using Sun ONE Studio. The resulting EJB JAR file can be extracted to yield the deployment descriptors.

3. Execute this command to create the JAR file:

```
jar -cvf module_name.jar *
```

TIP The assembly process can be automated using the Ant tool. To learn more, see "Apache Ant Assembly and Deployment Tool," on page 103.

NOTE According to the J2EE specification, section 8.1.1.2, "Dependencies," you cannot package utility classes within an individually deployed EJB module. Instead, package the EJB module and utility JAR within an application using the JAR Extension Mechanism Architecture. For other alternatives, see "Circumventing Classloader Isolation," on page 78.

Assembling a Lifecycle Module

To assemble a lifecycle module, follow these steps:

1. Create a working directory, and copy the contents of your module into it.
2. Execute this command to create the JAR file:

   `jar -cvf` *`module_name`*`.jar *`

TIP The assembly process can be automated using the Ant tool. To learn more, see "Apache Ant Assembly and Deployment Tool," on page 103.

For general information about lifecycle modules, see Chapter 6, "Developing Lifecycle Listeners."

Assembling an Application

To assemble an application, follow these steps:

1. Create a working directory, and copy the contents of your application into it, including all modules. Make sure it has the structure described in the *Sun ONE Application Server Developer's Guide to Enterprise JavaBeans Technology*.

2. Create two deployment descriptor files with these names: `application.xml` (required) and `sun-application.xml` (optional). For more information about these files, see "Sample Application XML Files," on page 134.

TIP The first time, you can assemble the application and create the deployment descriptors using Sun ONE Studio. The resulting EAR file can be extracted to yield the deployment descriptors.

3. Execute this command to create the J2EE application EAR file:

```
jar -cvf app_name.ear *
```

TIP The assembly process can be automated using the Ant tool. To learn more, see "Apache Ant Assembly and Deployment Tool," on page 103.

Assembling an ACC Client

This section provides some brief pointers for assembling ACC clients, but you should first read the *Sun ONE Application Server Developer's Guide to Clients*.

To assemble an ACC client JAR module, follow these steps:

1. Create a working directory, and copy the contents of your module into it. Make sure it has the structure described in the *Sun ONE Application Server Developer's Guide to Clients*.
2. Create deployment descriptor files with these names: `application-client.xml` and `sun-application-client.xml` (both required). For more information about these files, see the *Sun ONE Application Server Developer's Guide to Clients*.

TIP The first time, you can assemble the client JAR module and create the deployment descriptors using Sun ONE Studio. The resulting client JAR file can be extracted to yield the deployment descriptors.

3. Execute this command to create the client JAR file:

```
jar -cvfm module_name.jar META-INF/MANIFEST.MF *
```

TIP The assembly process can be automated using the Ant tool. To learn more, see "Apache Ant Assembly and Deployment Tool," on page 103.

For a brief description of how to deploy an ACC client and prepare the client machine, see "Deploying an ACC Client," on page 102.

Assembling a J2EE CA Resource Adapter

This section provides some brief pointers for assembling J2EE CA resource adapters, but you should first read the *Sun ONE J2EE CA Service Provider Implementation Administrator's Guide*.

The following XML connector files are required for deploying a connector to the application server.

- `ra.xml`
- `sun-ra.xml` (including security map)

The `ra.xml` file is based on the J2EE CA specification and is packaged with the connector. The `sun-ra.xml` file contains Sun ONE Application Server specific information.

To assemble a connector RAR module, follow these steps:

1. Create a working directory, and copy the contents of your module into it. Make sure it has the structure described in the *Sun ONE J2EE CA Service Provider Implementation Administrator's Guide*.
2. Create two deployment descriptor files with these names: `ra.xml` and `sun-ra.xml`. For more information about these files, see the *Sun ONE J2EE CA Service Provider Implementation Administrator's Guide*.

TIP The first time, you can assemble the RAR module and create the deployment descriptors using Sun ONE Studio. The resulting RAR file can be extracted to yield the deployment descriptors.

3. Execute this command to create the RAR file:

   `jar -cvf` *`module_name`*`.rar *`

TIP The assembly process can be automated using the Ant tool. To learn more, see "Apache Ant Assembly and Deployment Tool," on page 103.

Deploying Modules and Applications

This section describes the different ways to deploy J2EE applications and modules to the Sun ONE Application Server. It covers the following topics:

- Deployment Names and Errors
- The Deployment Life Cycle
- Tools for Deployment
- Deployment by Module or Application
- Deploying a WAR Module
- Deploying an EJB JAR Module
- Deploying a Lifecycle Module
- Deploying an ACC Client
- Deploying a J2EE CA Resource Adapter
- Access to Shared Frameworks

Deployment Names and Errors

When you deploy an application or module, a unique name is generated in the Sun ONE Application Server deployment descriptors file. Redeploying an application changes this name. Do not change this name manually. During deployment, the server detects any name collisions and does not load an application or module having a non-unique name. Messages are sent to the server log when this happens. For more about naming, see "Naming Standards," on page 69.

If an error occurs during deployment, the application or module is not deployed. If a module within an application contains an error, the entire application is not deployed. This prevents a partial deployment that could leave the server in an inconsistent state.

The Deployment Life Cycle

After an application is initially deployed, it may be modified and reloaded, redeployed, disabled, re-enabled, and finally undeployed (removed from the server). This section covers the following topics related to the deployment life cycle:

- Dynamic Deployment
- Disabling a Deployed Application or Module
- Dynamic Reloading

Dynamic Deployment

You can deploy, redeploy, and undeploy an application or module without restarting the server. This is called dynamic deployment.

Although primarily for developers, dynamic deployment can be used in operational environments to bring new applications and modules online without requiring a server restart. Whenever a redeployment is done, the sessions at that transit time become invalid. The client must restart the session.

NOTE You can overwrite a previously deployed application by using the `--force` option of `asadmin deploy` or by checking the appropriate box in the Administration interface during deployment. However, you must remove a preconfigured resource before you can update it.

Redeploying an application changes its automatically generated name.

Disabling a Deployed Application or Module

You can disable a deployed application or module without removing it from the server. Disabling an application makes it inaccessible to clients.

To disable an application or module, you can do one of the following:

- Set `enabled="false"` for the application or module in the `server.xml` file. For details about `server.xml`, see the *Sun ONE Application Server Administrator's Configuration File Reference.*
- Use the Administration interface:

  a. Open the Applications component under your server instance.

b. Go to the page for the type of application or module. For example, for a web application, go to the Web Apps page.

c. Check the box for the application or module you wish to disable.

d. Select the Disable button. The status listed on this page for the application or module changes to Disabled.

Dynamic Reloading

If dynamic reloading is enabled, you do not have to redeploy an application or module when you change its code or deployment descriptors. All you have to do is copy the changed JSP or class files into the deployment directory for the application or module. The server checks for changes periodically and redeploys the application, automatically and dynamically, with the changes.

This is useful in a development environment, because it allows code changes to be tested quickly. Dynamic reloading is not recommended for a production environment, however, because it may degrade performance. In addition, whenever a reload is done, the sessions at that transit time become invalid. The client must restart the session.

To enable dynamic reloading, you can do one of the following:

- Use the Administration interface:

  a. Open the Applications component under your server instance.

  b. Go to the Applications page.

  c. Check the Reload Enabled box to enable dynamic reloading.

  d. Enter a number of seconds in the Reload Poll Interval field to set the interval at which applications and modules are checked for code changes and dynamically reloaded.

  e. Click on the Save button.

  f. Go to the server instance page and select the Apply Changes button.

  For details, see the *Sun ONE Application Server Administrator's Guide*.

- Edit the following attributes of the `server.xml` file's `applications` element:

  - `dynamic-reload-enabled="true"` enables dynamic reloading.
  - `dynamic-reload-poll-interval-in-seconds` sets the interval at which applications and modules are checked for code changes and dynamically reloaded.

For details about `server.xml`, see the *Sun ONE Application Server Administrator's Configuration File Reference*.

In addition, to load new servlet files, reload EJB related changes, or reload deployment descriptor changes, you must do the following:

1. Create an empty file named `.reload` at the root of the deployed application:

   instance_dir`/applications/j2ee-apps/`*app_name*`/.reload`

   or individually deployed module:

   instance_dir`/applications/j2ee-modules/`*module_name*`/.reload`

2. Explicitly update the `.reload` file's timestamp (`touch .reload` in UNIX) each time you make the above changes.

For JSPs, changes are reloaded automatically at a frequency set in the `reload-interval` property of the `jsp-config` element in the `sun-web.xml` file. To disable dynamic reloading of JSPs, set the `reload-interval` property to `-1`.

Tools for Deployment

This section discusses the various tools that can be used to deploy modules and applications. The deployment tools include:

- Apache Ant
- Sun ONE Studio
- The asadmin Command
- The Administration Interface

Apache Ant

Ant can help you assemble and deploy modules and applications. For details, see "Apache Ant Assembly and Deployment Tool," on page 103.

Sun ONE Studio

You can use Sun ONE Studio 4 to deploy J2EE applications and modules. For more information about using Sun ONE Studio, see the *Sun ONE Studio 4, Enterprise Edition Tutorial*.

NOTE In Sun ONE Studio, deploying a module or application is referred to as *executing* it. Execution also includes making sure the server is running and displaying the correct URL to activate the module or application.

The asadmin Command

You can use the `asadmin` command to deploy or undeploy applications and individually deployed modules on local servers. Concurrent deployment on multiple machines or instances is not supported. This section describes the `asadmin` command only briefly. For full details, see the *Sun ONE Application Server Administrator's Guide*.

To deploy a lifecycle module, see "Deploying a Lifecycle Module," on page 100.

asadmin deploy

The `asadmin deploy` command deploys a WAR, JAR, RAR, or EAR file. To deploy an application, specify `--type application` in the command. To deploy an individual module, specify `--type ejb`, `web`, `connector`, or `client`. The syntax is as follows, with defaults shown for optional parameters that have them:

```
asadmin deploy --user admin_user [--password admin_password]
[--passwordfile password_file] [--host localhost] [--port 4848]
[--secure | -s] [--virtualservers virtual_servers] [--type
application|ejb|web|connector] [--contextroot contextroot]
[--force=true] [--precompilejsp=false] [--verify=false] [--name
component_name] [--upload=true] [--retrieve local_dirpath] [--instance
instance_name] filepath
```

For example, the following command deploys an individual EJB module:

```
asadmin deploy --user jadams --password secret --host localhost
--port 4848 --type ejb --instance server1 packagingEJB.jar
```

If `upload` is set to `false`, the *filepath* must be an absolute path on the server machine.

asadmin deploydir

The `asadmin deploydir` command deploys an application or module in an open directory structure. The structure must be as specified in "Directory Structure," on page 71. The location of the *dirpath* under *instance_dir*`/applications/j2ee-apps` or *instance_dir*`/applications/j2ee-modules` determines whether it is an application or individually deployed module. The syntax is as follows, with defaults shown for optional parameters that have them:

`asadmin deploydir --user` *admin_user* `[--password` *admin_password*`] [--passwordfile` *password_file*`] [--host localhost] [--port 4848] [--secure | -s] [--virtualservers` *virtual_servers*`] [--type application|ejb|web|connector] [--contextroot` *contextroot*`] [--force=true] [--precompilejsp=false] [--verify=false] [--name` *component_name*`] [--instance` *instance_name*`]` *dirpath*

For example, the following command deploys an individual EJB module:

```
asadmin deploydir --user jadams --password secret --host localhost
--port 4848 --type ejb --instance server1 packagingEJB
```

If `upload` is set to `false`, the *filepath* must be an absolute path on the server machine.

NOTE On Windows, if you are deploying a directory on a mapped drive, you must be running Sun ONE Application Server as the same user to which the mapped drive is assigned, or Sun ONE Application Server won't see the directory.

---

asadmin undeploy

The `asadmin undeploy` command undeploys an application or module. To undeploy an application, specify `--type app` in the command. To undeploy a module, specify `--type ejb`, `web`, `connector`, or `client`. The syntax is as follows, with defaults shown for optional parameters that have them:

`asadmin undeploy --user` *admin_user* `[--password` *admin_password*`] [--passwordfile` *password_file*`] [--host localhost] [--port 4848] [--secure | -s] [--type application|ejb|web|connector] [--instance` *instance_name*`]` *component_name*

For example, the following command undeploys an individual EJB module:

```
asadmin undeploy --user jadams --password secret --host localhost
--port 4848 --type ejb --instance server1 packagingEJB
```

The Administration Interface

You can use the Administration interface to deploy modules and applications to both local and remote Sun ONE Application Server sites. To use this tool, follow these steps:

1. Open the Applications component under your server instance.
2. Go to the Enterprise Apps, Web Apps, Connector Modules, or EJB Modules page.
3. Click on the Deploy button. (You can also undeploy, enable, or disable an application or module from this page.)
4. Enter the full path to the module or application directory or archive file (or click on Browse to find it), then click on the OK button.
5. Enter the module or application name.
6. For a web module, enter the context root.
7. Assign the application or web module to one or more virtual servers by checking the boxes next to the virtual server names.
8. You can also redeploy the module or application if it already exists (called *forced* deployment) by checking the appropriate box. This is optional.
9. You can run the verifier to check your deployment descriptor files. This is optional. For details about the verifier, see “The Deployment Descriptor Verifier,” on page 83.
10. Other fields are displayed depending on the type of module. Check appropriate boxes and enter appropriate values. Required fields are marked with asterisks (*).
11. Click on the OK button.

To deploy a lifecycle module, see “Deploying a Lifecycle Module,” on page 100.

Deployment by Module or Application

You can deploy applications or individual modules that are independent of applications. The runtime and file system implications of application-based or individual module-based deployment are described in “Runtime Environments,” on page 72.

Individual module-based deployment is preferable when components need to be accessed by:

- Other modules
- J2EE Applications
- ACC clients (Module-based deployment allows shared access to a bean from an ACC client, a servlet, or an EJB component.)

Modules can be combined into an EAR file and then deployed as a single module. This is similar to deploying the modules of the EAR independently.

Deploying a WAR Module

You deploy a WAR module as described in "Tools for Deployment," on page 95.

You can precompile JSPs during deployment by checking the appropriate box in the Administration interface or by using the `--precompilejsp` option of the `asadmin deploy` or `asadmin deploydir` command. The `sun-appserv-deploy` and `sun-appserv-jspc` Ant tasks also allow you to precompile JSPs.

You can keep the generated source for JSPs by adding the `-keepgenerated` flag to the `jsp-config` element in `sun-web.xml`. If you include this property when you deploy the WAR module, the generated source is kept in *instance_dir*`/generated/jsp/j2ee-apps/`*app_name*`/`*module_name* if it is in an application or *instance_dir*`/generated/jsp/j2ee-modules/`*module_name* if it is in an individually deployed web module.

For more information about JSP precompilation and the `-keepgenerated` property, see the *Sun ONE Application Server Developer's Guide to Web Applications*.

Deploying an EJB JAR Module

You deploy an EJB JAR module as described in "Tools for Deployment," on page 95.

You can keep the generated source for stubs and ties by adding the `-keepgenerated` flag to the `rmic-options` attribute of the `java-config` element in `server.xml`. If you include this flag when you deploy the EJB JAR module, the generated source is kept in *instance_dir*`/generated/ejb/j2ee-apps/`*app_name*`/`*module_name* if it is in an

application or *instance_dir*/generated/ejb/j2ee-modules/*module_name* if it is in an individually deployed EJB JAR module. For more information about the -keepgenerated flag, see the *Sun ONE Application Server Administrator's Configuration File Reference*.

Deploying a Lifecycle Module

For general information about lifecycle modules, see Chapter 6, "Developing Lifecycle Listeners."

You can deploy a lifecycle module using the following tools:

- The asadmin Command
- The Administration Interface

The asadmin Command

To deploy a lifecycle module, use the asadmin create-lifecycle-module command. The syntax is as follows, with defaults shown for optional parameters that have them:

asadmin create-lifecycle-module --user *admin_user* [--password *admin_password*] [--passwordfile *password_file*] [--host localhost] [--port 4848] [--secure | -s] [--instance *instance_name*] --classname *classname* [--classpath *classpath*] [--loadorder *load_order_number*] [--failurefatal=false] [--enabled=true] [--description *text_description*] [--property (*name*=*value*)[:*name*=*value*]*] *modulename*

For example:

```
asadmin create-lifecycle-module --user jadams --password secret
--host localhost --port 4848 --instance server1 --classname
RMIServer MyRMIServer
```

To undeploy a lifecycle module, use the asadmin delete-lifecycle-module command. The syntax is as follows, with defaults shown for optional parameters that have them:

asadmin delete-lifecycle-module --user *admin_user* [--password *admin_password*] [--passwordfile *password_file*] [--host localhost] [--port 4848] [--secure | -s] [--instance *instance_name*] *module_name*

For example:

```
asadmin delete-lifecycle-module --user jadams --password secret
--host localhost --port 4848 --instance server1 MyRMIServer
```

To list the lifecycle modules that are deployed on a server instance, use the `asadmin list-lifecycle-modules` command. The syntax is as follows, with defaults shown for optional parameters that have them:

`asadmin list-lifecycle-modules --user` *admin_user* `[--password` *admin_password*`] [--passwordfile` *password_file*`] [--host localhost] [--port 4848]` *instance_name*

For example:

```
asadmin list-lifecycle-module --user jadams --password secret --host
localhost --port 4848 server1
```

The Administration Interface

You can also use the Administration interface to deploy a lifecycle module. Follow these steps:

1. Open the Applications component under your server instance.
2. Go to the Lifecycle Modules page.
3. Click on the Deploy button.
4. Enter the following information:
   - Name (required) - The name of the lifecycle module.
   - Class Name (required) - The fully qualified name of the lifecycle module's class file.
   - Classpath (optional) - The classpath for the lifecycle module. Specifies where the module is located. The default location is under the application root directory.
   - Load Order (optional) - Determines the order in which lifecycle modules are loaded at startup. Modules with smaller integer values are loaded sooner. Values can range from `101` to the operating system's `MAXINT`. Values from `1` to `100` are reserved.
   - Fatal Failure (optional) - Determines whether the server is shut down if the lifecycle module fails. The default is `false`.
   - Lifecycle Enabled (optional) - Determines whether the lifecycle module is enabled. The default is `true`.
5. Click on the OK button.

Deploying an ACC Client

Deployment is only necessary for clients that communicate with EJB components. To deploy an ACC client:

1. Assemble the necessary client files (as described in "Assembling an ACC Client," on page 90).
2. Assemble the EJB components to be accessed by the client.
3. Package the client and EJB components together in an application.
4. Deploy the application.
5. After deployment, a client JAR file is created in the following location:

   instance_dir/applications/j2ee-apps/*app_name*/*app_name*Client.jar

   The client JAR contains the ties and necessary classes for the ACC client. Copy this file to the client machine, and set the APPCPATH environment variable on the client to point to this JAR.

If you wish to execute the client on the Sun ONE Application Server machine to test it, you can use the appclient script in the *install_dir*/bin directory. If you are using the default server instance, the only required option is -client. For example:

```
appclient -client converterClient.jar
```

The -xml parameter, which specifies the location of the sun-acc.xml file, is also required if you are not using the default instance.

Before you can execute an ACC client on a different machine, you must prepare the client machine:

1. Use the package-appclient script in the *install_dir*/bin directory to create the ACC package JAR file. This JAR file is created in the *install_dir*/lib/appclient directory.
2. Copy the ACC package JAR file to the client machine and unjar it. This creates a directory structure under an appclient directory.
3. Configure the sun-acc.xml file, located in the appclient/appserv/lib/appclient directory.
4. Configure the asenv.conf (asenv.bat on Windows) file, located in the appclient/appserv/bin directory.
5. Copy the client JAR to the client machine.

You are now ready to execute the client. For more information, see the *Sun ONE Application Server Developer's Guide to Clients*.

Deploying a J2EE CA Resource Adapter

You deploy a connector module as described in “Tools for Deployment,” on page 95.

Access to Shared Frameworks

When J2EE applications and modules use shared framework classes (such as utility classes and libraries) the classes can be put in the path for the System Classloader or the Common Classloader rather than in an application or module. If you assemble a large, shared library into every module that uses it, the result is a huge file that takes too long to register with the server. In addition, several versions of the same class could exist in different classloaders, which is a waste of resources.

For more information, see “Circumventing Classloader Isolation,” on page 78.

Apache Ant Assembly and Deployment Tool

You can use the automated assembly features available through Ant, a Java-based build tool available through the Apache Software Foundation:

`http://jakarta.apache.org/ant/`

Ant is a java-based build tool that is extended using Java classes. Instead of using shell commands, you declare the assembly steps using an XML document. Each task is run by an object that implements a particular task interface.

Apache Ant 1.4.1 is provided with Sun ONE Application Server (or with the operating system for bundled Solaris 9). The sample applications provided with Sun ONE Application Server have Ant `build.xml` files; see “Sample Applications,” on page 80.

Make sure you have done these things before using Ant:

- Include *install_dir*`/bin` in the PATH environment variable (`/usr/sfw/bin` for bundled Solaris 9). The Ant script provided with Sun ONE Application Server, `asant`, is located in this directory. For details on how to use `asant`, see the sample applications documentation in the *install_dir*`/samples/docs/ant.html` file.
- If you are executing platform-specific applications, such as the `exec` or `cvs` task, the ANT_HOME environment variable must be set to the Ant installation directory.

- The ANT_HOME environment variable for bundled Solaris 9 must include the following:
  - `/usr/sfw/bin` - the Ant binaries (shell scripts)
  - `/usr/sfw/doc/ant` - HTML documentation
  - `/usr/sfw/lib/ant` - Java classes that implement Ant
- The ANT_HOME environment variable for all other platforms is *install_dir*`/lib`.

This section covers the following Ant-related topics:

- Ant Tasks for Sun ONE Application Server 7
- Reusable Subelements

For information about standard Ant tasks, see the Ant documentation:

`http://jakarta.apache.org/ant/manual/index.html`

Ant Tasks for Sun ONE Application Server 7

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

- sun-appserv-deploy
- sun-appserv-undeploy
- sun-appserv-instance
- sun-appserv-component
- sun-appserv-admin
- sun-appserv-jspc

sun-appserv-deploy

Deploys any of the following to a local or remote Sun ONE Application Server instance.

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

- Enterprise connector (RAR file)
- Application client

Subelements

The following table describes subelements for the `sun-appserv-deploy` task. These are objects upon which this task acts. The left column lists the subelement name, and the right column describes what the element specifies.

`sun-appserv-deploy` subelements

Element	Description
`server`	A Sun ONE Application Server instance.
`component`	A component to be deployed.
`fileset`	A set of component files that match specified parameters.

Attributes

The following table describes attributes for the `sun-appserv-deploy` task. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute specifies.

`sun-appserv-deploy` attributes

Attribute	Default	Description
`file`	none	(optional if a `component` or `fileset` subelement is present, otherwise required) The component to deploy. If this attribute refers to a file, it must be a valid archive. If this attribute refers to a directory, it must contain a valid archive in which all components have been exploded. If `upload` is set to `false`, this must be an absolute path on the server machine.
`name`	file name without extension	(optional) The display name for the component being deployed.

`sun-appserv-deploy` attributes

Attribute	Default	Description
`type`	determined from the file or directory name extension	(optional) The type of component being deployed. Valid types are `application`, `ejb`, `web`, and `connector`. If not specified, the file (or directory) extension is used to determine the component type: `.ear` for `applicaton`, `.jar` for `ejb`, `.war` for `web`, and `.rar` for `connector`. If it's not possible to determine the component type using the file extension, the default is `application`.
`force`	`true`	(optional) If `true`, the component is overwritten if it already exists on the server. If `false`, `sun-appserv-deploy` fails if the component exists.
`retrievestubs`	client stubs not saved	(optional) The directory where client stubs are saved. This attribute is inherited by nested `component` elements.
`precompilejsp`	`false`	(optional) If `true`, all JSPs found in an enterprise application (`.ear`) or web application (`.war`) are precompiled. This attribute is ignored for other component types. This attribute is inherited by nested `component` elements.
`verify`	`false`	(optional) If `true`, syntax and semantics for all deployment descriptors are automatically verified for correctness. This attribute is inherited by nested `component` elements.
`contextroot`	file name without extension	(optional) The context root for a web module (WAR file). This attribute is ignored if the component is not a WAR file.
`upload`	`true`	(optional) If `true`, the component is transferred to the server for deployment. If the component is being deployed on the local machine, set upload to `false` to reduce deployment time.
`virtualservers`	default virtual server only	(optional) A comma-separated list of virtual servers to be deployment targets. This attribute applies only to application (`.ear`) or web (`.war`) components and is ignored for other component types. This attribute is inherited by nested `server` elements.
`user`	`admin`	(optional) The username used when logging into the application server administration instance. This attribute is inherited by nested `server` elements.

`sun-appserv-deploy` attributes

Attribute	Default	Description
`password`	none	The password used when logging into the application server administration instance. This attribute is inherited by nested `server` elements.
`host`	`localhost`	(optional) Target server. When deploying to a remote server, use the fully qualified hostname. This attribute is inherited by nested `server` elements.
`port`	`4848`	(optional) The administration port on the target server. This attribute is inherited by nested `server` elements.
`instance`	name of default instance	(optional) Target application server instance. This attribute is inherited by nested `server` elements.
`sunonehome`	see description	(optional) The installation directory for the local Sun ONE Application Server 7 installation, which is used to find the administrative classes. If not specified, the command checks to see if the `sunone.home` parameter has been set. Otherwise, administrative classes must be in the system classpath.

Examples

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

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

Here is an equivalent script showing all the implied attributes:

```
<sun-appserv-deploy
   file="${assemble}/simpleapp.ear"
   name="simpleapp"
   type="application"
   force="true"
   precompilejsp="false"
   verify="false"
   upload="true"
   user="admin"
   password="${password}"
```

```
    host="localhost"
    port="4848"
    instance="${default-instance-name}"
    sunonehome="${sunone.home}" />
```

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

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

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

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

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

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

sun-appserv-undeploy

Undeploys any of the following from a local or remote Sun ONE Application Server instance.

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

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

Subelements

The following table describes subelements for the `sun-appserv-undeploy` task. These are objects upon which this task acts. The left column lists the subelement name, and the right column describes what the element specifies.

`sun-appserv-undeploy` subelements

Element	Description
`server`	A Sun ONE Application Server instance.
`component`	A component to be deployed.
`fileset`	A set of component files that match specified parameters.

Attributes

The following table describes attributes for the `sun-appserv-undeploy` task. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute specifies.

`sun-appserv-undeploy` attributes

Attribute	Default	Description
`name`	file name without extension	(optional if a `component` or `fileset` subelement is present or the `file` attribute is specified, otherwise required) The display name for the component being undeployed.
`file`	none	(optional) The component to undeploy. If this attribute refers to a file, it must be a valid archive. If this attribute refers to a directory, it must contain a valid archive in which all components have been exploded.

`sun-appserv-undeploy` attributes

Attribute	Default	Description
`type`	determined from the file or directory name extension	(optional) The type of component being undeployed. Valid types are `application`, `ejb`, `web`, and `connector`. If not specified, the file (or directory) extension is used to determine the component type: `.ear` for `applicaton`, `.jar` for `ejb`, `.war` for `web`, and `.rar` for `connector`. If it's not possible to determine the component type using the file extension, the default is `application`.
`user`	`admin`	(optional) The username used when logging into the application server administration instance. This attribute is inherited by nested `server` elements.
`password`	none	The password used when logging into the application server administration instance. This attribute is inherited by nested `server` elements.
`host`	`localhost`	(optional) Target server. When deploying to a remote server, use the fully qualified hostname. This attribute is inherited by nested `server` elements.
`port`	`4848`	(optional) The administration port on the target server. This attribute is inherited by nested `server` elements.
`instance`	name of default instance	(optional) Target application server instance. This attribute is inherited by nested `server` elements.
`sunonehome`	see description	(optional) The installation directory for the local Sun ONE Application Server 7 installation, which is used to find the administrative classes. If not specified, the command checks to see if the `sunone.home` parameter has been set. Otherwise, the administrative classes must be in the system classpath.

Examples

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

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

Here is an equivalent script showing all the implied attributes:

```
<sun-appserv-undeploy
    name="simpleapp"
    type="application"
    user="admin"
```

```
    password="${password}"
    host="localhost"
    port="4848"
    instance="${default-instance-name}"
    sunonehome="${sunone.home}" />
```

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

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

As with the deployment process, components can be undeployed from multiple servers in a single command. This example shows the same three components being removed from two different instances of Sun ONE Application Server 7. In this example, the passwords for both instances are the same.

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

sun-appserv-instance

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

Subelements

The following table describes subelements for the `sun-appserv-instance` task. These are objects upon which this task acts. The left column lists the subelement name, and the right column describes what the element specifies.

`sun-appserv-instance` subelements

Element	Description
`server`	A Sun ONE Application Server instance.

Attributes

The following table describes attributes for the `sun-appserv-instance` task. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute specifies.

`sun-appserv-instance` attributes

Attribute	Default	Description
`action`	none	The control command for the target application server. Valid values are `start`, `stop`, `restart`, `create`, and `delete`. A `restart` sends the `stop` command followed by the `start` command. The `restart` command is not supported on Windows.
`debug`	`false`	(optional) If `action` is set to `start` or `restart`, specifies whether the server starts in debug mode. This attribute is ignored for other values of `action`. If `true`, the instance generates additional debugging output throughout its lifetime. This attribute is inherited by nested `server` elements.
`instanceport`	none	(optional unless `action` is `create`) If a new instance is being created, this attribute specifies its port number. Otherwise, this attribute is ignored. This attribute is inherited by nested `server` elements.
`local`	`false`	(optional) If `true`, an instance on the local machine (in other words, `localhost`) is the target for the `action`, an administration server need not be running, and the `host`, `port`, `user`, and `password` attributes are ignored. If false, an administration server must be running and the `host`, `port`, `user`, and `password` attributes must be set appropriately. This attribute is inherited by nested `server` elements.
`domain`		(optional unless `local="true"` and there are multiple local domains) The target domain for a local `action`. If `local="false"` this attribute is ignored. This attribute is inherited by nested `server` elements.
`user`	`admin`	(optional) The username used when logging into the application server administration instance. This attribute is inherited by nested `server` elements.
`password`	none	(required unless `local` is set to `true`)The password used when logging into the application server administration instance. This attribute is inherited by nested `server` elements.

`sun-appserv-instance` attributes

Attribute	Default	Description
`host`	`localhost`	(optional) Target server. If it is a remote server, use the fully qualified hostname. This attribute is inherited by nested `server` elements.
`port`	4848	(optional) The administration port on the target server. This attribute is inherited by nested `server` elements.
`instance`	name of default instance	Target application server instance. This attribute is inherited by nested `server` elements.
`sunonehome`	see description	(optional) The installation directory for the local Sun ONE Application Server 7 installation, which is used to find the administrative classes. If not specified, the command checks to see if the `sunone.home` parameter has been set. Otherwise, the administrative classes must be in the system classpath.

Examples

This example starts the local Sun ONE Application Server 7 instance:

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

Here is an equivalent script showing all the implied attributes:

```
<sun-appserv-instance
   action="start"
   user="admin"
   password="${password}"
   host="localhost"
   port="4848"
   instance="${default-instance-name}"
   sunonehome="${sunone.home}" />
```

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

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

This example creates a new Sun ONE Application Server 7 instance:

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

Here is an equivalent script showing all the implied attributes:

```
<sun-appserv-instance
   action="create"
   instanceport="8080"
   user="admin"
   password="${password}"
   host="localhost"
   port="4848"
   instance="development"
   sunonehome="${sunone.home}" />
```

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

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

These instances can also be removed from their respective servers:

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

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

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

sun-appserv-component

Enables or disables the following J2EE component types that have been deployed to Sun ONE Application Server 7.

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

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

Subelements

The following table describes subelements for the `sun-appserv-component` task. These are objects upon which this task acts. The left column lists the subelement name, and the right column describes what the element specifies.

`sun-appserv-component` subelements

Element	Description
`server`	A Sun ONE Application Server instance.
`component`	A component to be deployed.
`fileset`	A set of component files that match specified parameters.

Attributes

The following table describes attributes for the `sun-appserv-component` task. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute specifies.

`sun-appserv-component` attributes

Attribute	Default	Description
`action`	none	The control command for the target application server. Valid values are `enable` and `disable`.

`sun-appserv-component` attributes

Attribute	Default	Description
`name`	file name without extension	(optional if a `component` or `fileset` subelement is present or the `file` attribute is specified, otherwise required) The display name for the component being enabled or disabled.
`file`	none	(optional) The component to enable or disable. If this attribute refers to a file, it must be a valid archive. If this attribute refers to a directory, it must contain a valid archive in which all components have been exploded.
`type`	determined from the file or directory name extension	(optional) The type of component being enabled or disabled. Valid types are `application`, `ejb`, `web`, and `connector`. If not specified, the file (or directory) extension is used to determine the component type: `.ear` for `applicaton`, `.jar` for `ejb`, `.war` for `web`, and `.rar` for `connector`. If it's not possible to determine the component type using the file extension, the default is `application`.
`user`	`admin`	(optional) The username used when logging into the application server administration instance. This attribute is inherited by nested `server` elements.
`password`	none	The password used when logging into the application server administration instance. This attribute is inherited by nested `server` elements.
`host`	`localhost`	(optional) Target server. When enabling or disabling a remote server, use the fully qualified hostname. This attribute is inherited by nested `server` elements.
`port`	4848	(optional) The administration port on the target server. This attribute is inherited by nested `server` elements.
`instance`	name of default instance	(optional) Target application server instance. This attribute is inherited by nested `server` elements.
`sunonehome`	see description	(optional) The installation directory for the local Sun ONE Application Server 7 installation, which is used to find the administrative classes. If not specified, the command checks to see if the `sunone.home` parameter has been set. Otherwise, the administrative classes must be in the system classpath.

Examples

Here is a simple example of disabling a component:

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

Here is a simple example of enabling a component:

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

Here is an equivalent script showing all the implied attributes:

```
<sun-appserv-component
   action="enable"
   name="simpleapp"
   type="application"
   user="admin"
   password="${password}"
   host="localhost"
   port="4848"
   instance="${default-instance-name}"
   sunonehome="${sunone.home}" />
```

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

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

Components can be enabled or disabled on multiple servers in a single task. This example shows the same three components being enabled on two different instances of Sun ONE Application Server 7. In this example, the passwords for both instances are the same.

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

sun-appserv-admin

Enables arbitrary administrative commands and scripts to be executed on the Sun ONE Application Server 7. This is useful for cases where a specific Ant task hasn't been developed or a set of related commands are in a single script.

Subelements

The following table describes subelements for the `sun-appserv-admin` task. These are objects upon which this task acts. The left column lists the subelement name, and the right column describes what the element specifies.

`sun-appserv-admin` subelements

Element	Description
`server`	A Sun ONE Application Server instance.

Attributes

The following table describes attributes for the `sun-appserv-admin` task. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute specifies.

`sun-appserv-admin` attributes

Attribute	Default	Description
`command`	none	(exactly one of these is required: `command`, `commandfile`, or `explicitcommand`) The command to execute. If the `user`, `password`, `host`, `port`, or `instance` attributes are also specified, they are automatically inserted into the command before execution. If any of these options are specified in the command string, the corresponding attribute values are ignored.
`commandfile`	none	(exactly one of these is required: `command`, `commandfile`, or `explicitcommand`) The command script to execute. If `commandfile` is used, the values of all other attributes are ignored. Be sure to end the script referenced by `commandfile` with the `exit` command; if you omit `exit`, the Ant task may appear to hang after the command script is called.

`sun-appserv-admin` attributes

Attribute	Default	Description
`explicitcommand`	none	(exactly one of these is required: `command`, `commandfile`, or `explicitcommand`) The exact command to execute. No command processing is done, and all other attributes are ignored.
`user`	`admin`	(optional) The username used when logging into the application server administration instance. This attribute is inherited by nested `server` elements.
`password`	none	(optional) The password used when logging into the application server administration instance. This attribute is inherited by nested `server` elements.
`host`	`localhost`	(optional) Target server. If it is a remote server, use the fully qualified hostname. This attribute is inherited by nested `server` elements.
`port`	`4848`	(optional) The administration port on the target server. This attribute is inherited by nested `server` elements.
`instance`	name of default instance	(optional) Target application server instance. This attribute is inherited by nested `server` elements.
`sunonehome`	see description	(optional) The installation directory for the local Sun ONE Application Server 7 installation, which is used to find the administrative classes. If not specified, the command checks to see if the `sunone.home` parameter has been set. Otherwise, the administrative classes must be in the system classpath.

sun-appserv-jspc

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

Subelements

none

Attributes

The following table describes attributes for the `sun-appserv-jspc` task. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute specifies.

`sun-appserv-jspc` attributes

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

`sun-appserv-jspc` attributes

Attribute	Default	Description
`sunonehome`	see description	(optional) The installation directory for the local Sun ONE Application Server 7 installation, which is used to find the administrative classes. If not specified, the command checks to see if the `sunone.home` parameter has been set. Otherwise, the administrative classes must be in the system classpath.

Example

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

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

Reusable Subelements

Reusable subelements of the Ant tasks for Sun ONE Application Server 7 are as follows. These are objects upon which the Ant tasks act.

- server
- component
- fileset

server

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

Subelements

none

Attributes

The following table describes attributes for the `server` element. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute specifies.

`server` attributes

Attribute	Default	Description
`user`	`admin`	(optional) The username used when logging into the application server administration instance.
`password`	none	(optional if specified in the parent task) The password used when logging into the application server administration instance.
`host`	`localhost`	(optional) Target server. When targeting a remote server, use the fully qualified hostname.
`port`	`4848`	(optional) The administration port on the target server.
`instance`	name of default instance	(optional) Target application server instance.
`domain`		(applies to `sun-appserv-instance` only, optional unless `local="true"` and there are multiple local domains) The target domain for a local `action`. If `local="false"` this attribute is ignored.
`instanceport`	none	(applies to `sun-appserv-instance` only, optional unless `action` is `create`) If a new instance is being created, this attribute specifies its port number. Otherwise, this attribute is ignored.

`server` attributes

Attribute	Default	Description
`debug`	`false`	(applies to `sun-appserv-instance` only, optional) If `action` is set to `start`, specifies whether the server starts in debug mode. This attribute is ignored for other values of `action`. If `true`, the instance generates additional debugging output throughout its lifetime.
`local`	`false`	(applies to `sun-appserv-instance` only, optional) If `true`, an instance on the local machine (in other words, `localhost`) is the target for the `action`, an administration server need not be running, and the `host`, `port`, `user`, and `password` attributes are ignored. If false, an administration server must be running and the `host`, `port`, `user`, and `password` attributes must be set appropriately.
`upload`	`true`	(applies to `sun-appserv-deploy` only, optional) If `true`, the component is transferred to the server for deployment. If the component is being deployed on the local machine, set upload to `false` to reduce deployment time.
`virtualservers`	default virtual server only	(applies to `sun-appserv-deploy` only, optional) A comma-separated list of virtual servers to be deployment targets. This attribute applies only to application (`.ear`) or web (`.war`) components and is ignored for other component types.

Examples

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

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

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

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

These instances can also be removed from their respective servers:

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

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

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

You can deploy multiple components to multiple servers (see the `component` nested element. This example deploys each component to two Sun ONE Application Server instances running on remote servers. Both servers use the same password.

```
<sun-appserv-deploy password="${password}" sunonehome="/opt/s1as7" >
   <server host="greg.sun.com"/>
   <server host="joe.sun.com"/>
   <component file="${assemble}/simpleapp.ear"/>
   <component file="${assemble}/simpleservlet.war"
      contextroot="test"/>
   <component file="${assemble}/simplebean.jar"/>
</sun-appserv-deploy>
```

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

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

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

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

component

Specifies a J2EE component. Allows a single task to act on multiple components. The `component` attributes override corresponding attributes in the parent task; therefore, the parent task attributes function as default values.

Subelements

none

Attributes

The following table describes attributes for the `component` element. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute specifies.

`component` attributes

Attribute	Default	Description
`file`	none	(optional if the parent task is `sun-appserv-undeploy` or `sun-appserv-component`) The target component. If this attribute refers to a file, it must be a valid archive. If this attribute refers to a directory, it must contain a valid archive in which all components have been exploded. If `upload` is set to `false`, this must be an absolute path on the server machine.
`name`	file name without extension	(optional) The display name for the component.

`component` attributes

Attribute	Default	Description
`type`	determined from the file or directory name extension	(optional) The type of component. Valid types are `application`, `ejb`, `web`, and `connector`. If not specified, the file (or directory) extension is used to determine the component type: `.ear` for `applicaton`, `.jar` for `ejb`, `.war` for `web`, and `.rar` for `connector`. If it's not possible to determine the component type using the file extension, the default is `application`.
`force`	`true`	(applies to `sun-appserv-deploy` only, optional) If `true`, the component is overwritten if it already exists on the server. If `false`, the containing element's operation fails if the component exists.
`precompilejsp`	`false`	(applies to `sun-appserv-deploy` only, optional) If `true`, all JSPs found in an enterprise application (`.ear`) or web application (`.war`) are precompiled. This attribute is ignored for other component types.
`retrievestubs`	client stubs not saved	(applies to `sun-appserv-deploy` only, optional) The directory where client stubs are saved.
`contextroot`	file name without extension	(applies to `sun-appserv-deploy` only, optional) The context root for a web module (WAR file). This attribute is ignored if the component is not a WAR file.
`verify`	`false`	(applies to `sun-appserv-deploy` only, optional) If `true`, syntax and semantics for all deployment descriptors is automatically verified for correctness.

Examples

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

```
<sun-appserv-deploy password="${password}" host="greg.sun.com"
       sunonehome="/opt/s1as7" >
    <component file="${assemble}/simpleapp.ear"/>
    <component file="${assemble}/simpleservlet.war"
       contextroot="test"/>
    <component file="${assemble}/simplebean.jar"/>
</sun-appserv-deploy>
```

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

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

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

```
<sun-appserv-deploy password="${password}" sunonehome="/opt/s1as7" >
   <server host="greg.sun.com"/>
   <server host="joe.sun.com"/>
   <component file="${assemble}/simpleapp.ear"/>
   <component file="${assemble}/simpleservlet.war"
      contextroot="test"/>
   <component file="${assemble}/simplebean.jar"/>
</sun-appserv-deploy>
```

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

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

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

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

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

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

fileset

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

`http://jakarta.apache.org/ant/manual/CoreTypes/fileset.html`

The Application Deployment Descriptor Files

Sun ONE Application Server applications include two deployment descriptor files:

- A J2EE standard file (`application.xml`), described in the Java Servlet Specification, v2.3, Chapter 13, “Deployment Descriptors.”
- An optional Sun ONE Application Server specific file (`sun-application.xml`), described in this section.

This section covers the following topics:

- The sun-application_1_3-0.dtd File
- Elements in the sun-application.xml File
- Sample Application XML Files

The sun-application_1_3-0.dtd File

The `sun-application_1_3-0.dtd` file defines the structure of the `sun-application.xml` file, including the elements it can contain and the subelements and attributes these elements can have. The `sun-application_1_3-0.dtd` file is located in the *install_dir*`/lib/dtds` directory.

NOTE Do not edit the `sun-application_1_3-0.dtd` file; its contents change only with new versions of Sun ONE Application Server.

For general information about DTD files and XML, see the XML specification at:

`http://www.w3.org/TR/REC-xml`

Each element defined in a DTD file (which may be present in the corresponding XML file) can contain the following:

- Subelements
- Data
- Attributes

Subelements

Elements can contain subelements. For example, the following code defines the `sun-application` element.

```
<!ELEMENT sun-application (web*, pass-by-reference?, unique-id?, security-role-mapping*)>
```

The `ELEMENT` tag specifies that a `sun-application` element can contain `web`, `pass-by-reference`, `unique-id`, and `security-role-mapping` elements.

The following table shows how optional suffix characters of subelements determine the requirement rules, or number of allowed occurrences, for the subelements. The left column lists the subelement ending character, and the right column lists the corresponding requirement rule.

requirement rules and subelement suffixes

Subelement Suffix	Requirement Rule
element`*`	Can contain *zero or more* of this subelement.
element`?`	Can contain *zero or one* of this subelement.
element`+`	Must contain *one or more* of this subelement.
element (no suffix)	Must contain *only one* of this subelement.

If an element cannot contain other elements, you see `EMPTY` or `(#PCDATA)` instead of a list of element names in parentheses.

Data

Some elements contain character data instead of subelements. These elements have definitions of the following format:

```
<!ELEMENT element-name (#PCDATA)>
```

For example:

```
<!ELEMENT role-name (#PCDATA)>
```

In the `sun-application.xml` file, white space is treated as part of the data in a data element. Therefore, there should be no extra white space before or after the data delimited by a data element. For example:

```
<role-name>manager</role-name>
```

Attributes

Elements that have `ATTLIST` tags contain attributes. None of the elements in the `sun-application.xml` file contain attributes.

Elements in the sun-application.xml File

This section describes the following elements in the `sun-application.xml` file:

- `sun-application`
- `web`
- `web-uri`
- `context-root`
- `pass-by-reference`
- `unique-id`
- `security-role-mapping`
- `role-name`
- `principal-name`
- `group-name`

sun-application

Defines Sun ONE Application Server specific configuration for an application. This is the root element; there can only be one `sun-application` element in a `sun-application.xml` file.

Subelements

The following table describes subelements for the `sun-application` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

`sun-application` subelements

Element	Required	Description
`web`	zero or more	Specifies the application's web tier configuration.
`pass-by-reference`	zero or one	Determines whether EJB modules use pass-by-value or pass-by-reference semantics.
`unique-id`	zero or one	Contains the unique ID for the application.
`security-role-mapping`	zero or more	Maps a role in the corresponding J2EE XML file to a user or group.

web

Specifies the application's web tier configuration.

Subelements

The following table describes subelements for the `web` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

`web` subelements

Element	Required	Description
`web-uri`	only one	Contains the web URI for the application.
`context-root`	only one	Contains the web context root for the application.

web-uri

Contains the web URI for the application. Must match the corresponding element in the `application.xml` file.

Subelements

none

context-root

Contains the web context root for the application. Overrides the corresponding element in the `application.xml` file.

Subelements

none

pass-by-reference

If `false` (the default if this element is not present), this application uses pass-by-value semantics, which the EJB specification requires. If `true`, this application uses pass-by-reference semantics. The setting of this element in the `sun-application.xml` file applies to all EJB modules in the application.

For an individually deployed EJB module, you can set the same element in the `sun-ejb-jar.xml` file. If you want to use pass-by-reference at both the bean and application level, the bean level takes precedence over the application level. For details, see the *Sun ONE Application Server Developer's Guide to Enterprise JavaBeans Technology*.

Subelements

none

unique-id

Contains the unique ID for the application. This value is automatically updated each time the application is deployed or redeployed. Do not edit this value.

Subelements

none

security-role-mapping

Maps roles to users and groups. At least one principal or group name is required, but you do not need to have one of each.

Subelements

The following table describes subelements for the `security-role-mapping` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

`security-role-mapping` subelements

Element	Required	Description
`role-name`	only one	Contains the `role-name` in the `security-role` element of the `application.xml` file.
`principal-name`	one or more if no `group-name`, otherwise zero or more	Contains the principal (user) name.
`group-name`	one or more if no `principal-name`, otherwise zero or more	Contains the group name.

role-name

Contains the `role-name` in the `security-role` element of the `application.xml` file.

Subelements

none

principal-name

Contains the principal (user) name.

Subelements

none

group-name

Contains the group name.

Subelements

none

Sample Application XML Files

This section includes the following:

- Sample application.xml File
- Sample sun-application.xml File

Sample application.xml File

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE application PUBLIC '-//Sun Microsystems, Inc.//DTD J2EE
Application 1.3//EN' 'http://java.sun.com/dtd/application_1_3.dtd'>

<application>

   <display-name>app_stateless-simple</display-name>

   <description>Application description</description>

   <module>

      <ejb>stateless-simpleEjb.jar</ejb>

   </module>

   <module>

      <web>

         <web-uri>stateless-simple.war</web-uri>

         <context-root>helloworld</context-root>

      </web>

   </module>

</application>
```

Sample sun-application.xml File

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE sun-application PUBLIC '-//Sun Microsystems, Inc.//DTD Sun
ONE Application Server 7.0 J2EE Application 1.3//EN'
'http://www.sun.com/software/sunone/appserver/dtds/sun-application_
1_3-0.dtd'>

<sun-application>

  <unique-id>67488732739338240</unique-id>

</sun-application>
```

Chapter 5

Debugging J2EE Applications

This chapter gives guidelines for debugging applications in Sun ONE Application Server 7. It includes the following sections:

- Enabling Debugging
- JPDA Options
- Using Sun ONE Studio for Debugging
- Debugging JSPs
- Generating a Stack Trace for Debugging
- Sun ONE Message Queue Debugging
- Logging
- Profiling

Debugging applications requires that you edit the `server.xml` file as described in this chapter. For more general information about this file, see the *Sun ONE Application Server Administrator's Configuration File Reference*.

Enabling Debugging

When you enable debugging, you enable both local and remote debugging.

You can enable debugging in one of these ways:

- Using the Administration Interface (recommended)
- Editing the server.xml File

Sun ONE Application Server debugging is based on the JPDA (Java Platform Debugger Architecture). For more information, see "JPDA Options," on page 137.

Using the Administration Interface

To enable debugging:

1. Go to the server instance page.
2. Select the General tab.
3. Check the Start in Debug Mode box.
4. Select the Apply Changes button.
5. Restart the server.
6. Select the JVM Settings tab and the General option.
7. Look in the Debug Options field for address=*port_number*, and write down this port number. You will need this port number when you attach a debugger.
8. If you wish to add JPDA options, follow these additional, optional steps:
   a. Add any desired JPDA debugging options in Debug Options. See "JPDA Options," on page 137.
   b. Select the Save button.
   c. Repeat Step 2 through Step 5 above.

Editing the server.xml File

To enable debugging, set the following attributes of the `java-config` element in the `server.xml` file:

- Set `debug-enabled="true"` to turn on debugging.
- Add any desired JPDA debugging options in the `debug-options` attribute. See "JPDA Options," on page 137.
- To specify the port to use when attaching the JVM to a debugger, specify `address=`*port_number* in the `debug-options` attribute.

For details about the `server.xml` file, see the *Sun ONE Application Server Administrator's Configuration File Reference*.

JPDA Options

The default JPDA options in Sun ONE Application Server are as follows:

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

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

To specify the port to use when attaching the JVM to a debugger, specify `address=`*port_number*.

You can include additional options. A list of JPDA debugging options is available here:

`http://java.sun.com/products/jpda/doc/conninv.html#Invocation`

Using Sun ONE Studio for Debugging

To use the Sun ONE Studio 4 debugger with Sun ONE Application Server:

1. Start Sun ONE Studio, and mount the directory that contains the application source code you want to debug.
2. Select the Runtime tab, then navigate to the Sun ONE Application Server instance that you want to start in debug mode (local or remote).
3. Right click on the server instance and select Status from the menu that appears. The Status window appears.
4. If the server instance is not running in debug mode, select Stop Server Instance, then select Start in Debug Mode.
5. When the server instance is running in debug mode, a *port_number* is displayed in the Status window and on the status line. Write down this port number.
6. Select the Debug menu and the Attach... option.
7. Change the Connector:text field to SocketAttach.
8. Type the host name of the Application Server in the Host text box.
9. Type the *port_number* in the Port text box, then select OK.

You should be able to debug your Java classes now using Sun ONE Studio.

For help on debugging applications with Sun ONE Studio, select Help, select Contents, then select Debugging Java Programs. You can also consult the *Sun ONE Studio 4, Enterprise Edition Tutorial*.

Debugging JSPs

When you use Sun ONE Studio 4 to debug JSPs, you can set breakpoints in either the JSP code or the generated servlet code, and you can switch between them and see the same breakpoints in both.

To set up debugging in Sun ONE Studio, see the previous section. For further details, see the *Sun ONE Studio 4, Enterprise Edition Tutorial*.

Generating a Stack Trace for Debugging

You can generate a Java stack trace for debugging as described here:

`http://developer.java.sun.com/developer/technicalArticles/Programming/Stacktrace/`

If the `-Xrs` flag is set (for reduced signal usage) in the `server.xml` file (under `<jvm-options>`), comment it out before generating the stack trace. If the `-Xrs` flag is used, the server may simply dump core and restart when you send the signal to generate the trace.

The stack trace goes to the system log file or to `stderr` based on the `log-service` attributes in `server.xml`.

For more about the `server.xml` file, see the *Sun ONE Application Server Administrator's Configuration File Reference*.

Sun ONE Message Queue Debugging

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

Logging

You can use the Sun ONE Application Server's log files to help debug your applications. For general information about logging, see the *Sun ONE Application Server Administrator's Guide*. For information about configuring logging in the `server.xml` file, see the *Sun ONE Application Server Administrator's Configuration File Reference*.

You can change logging settings in one of these ways:

- Using the Administration Interface
- Editing the server.xml File

Using the Administration Interface

To change logging settings:

1. Go to the server instance page.
2. Select the Logging tab and the General option.
3. If you wish to send exceptions to the client in addition to the log file, check the Echo to stderr box.
4. On Windows only, if you wish to enable the console, check the Create console box.
5. Select the Save button.
6. Restart the server.

Editing the server.xml File

To change logging settings, set the attributes of the `log-service` element in the `server.xml` file.

You can send exceptions to the client in addition to the log file. Set the following parameter in `server.xml`. If the client is a browser, exceptions are displayed in the browser.

```
<log-service ... echo-log-messages-to-stderr=true ... />
```

On Windows only, you can add the following line to the `server.xml` file to enable the console:

```
<log-service ... create-console=true ... />
```

For details about the `server.xml` file, see the *Sun ONE Application Server Administrator's Configuration File Reference*.

Profiling

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

- The HPROF Profiler
- The Optimizeit Profiler
- The Wily Introscope Profiler
- The JProbe Profiler

The HPROF Profiler

HPROF is a simple profiler agent shipped with the Java 2 SDK. It is a dynamically linked library that interacts with the JVMPI and writes out profiling information either to a file or to a socket in ASCII or binary format. This information can be further processed by a profiler front-end tool such as HAT.

HPROF can present CPU usage, heap allocation statistics, and monitor contention profiles. In addition, it can also report complete heap dumps and states of all the monitors and threads in the Java virtual machine. For more details on the HPROF profiler, see the JDK documentation at:

`http://java.sun.com/j2se/1.4/docs/guide/jvmpi/jvmpi.html#hprof`

Once HPROF is installed using the following instructions, its libraries are loaded into the server process.

To use HPROF profiling on UNIX, follow these steps:

1. Configure Sun ONE Application Server in one of these ways:
    - Go to the server instance page in the Administration interface, select the JVM Settings tab, select the Profiler option, and edit the following fields before selecting Save:

- Name: `hprof`
- Enabled: `true`
- Classpath: (leave blank)
- Native Library Path: (leave blank)
- JVM Option: For each of these options, type the option in the JVM Option field, select Add, then check its box in the JVM Options list:

  `-Xrunhprof:file=log.txt,`*options*

- Edit the `server.xml` file as appropriate:

```
<!-- hprof options -->
<profiler name="hprof" enabled="true">
   <jvm-options>
      -Xrunhprof:file=log.txt,options
   </jvm-options>
</profiler>
```

NOTE Do not use the `-Xrs` flag.

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

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

The `file` option is important because it determines where the stack dump is written in Step 6.

The syntax of HPROF options is as follows:

`-Xrunhprof[:help]|[:`*option=value*`,`*option2=value2*`, ...]`

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

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

Option Name and Value   Description             Default
---------------------   -----------             -------
heap=dump|sites|all     heap profiling          all
cpu=samples|old         CPU usage               off
format=a|b              ascii or binary output  a
file=<file>             write data to file      java.hprof
                                                (.txt for ascii)
net=<host>:<port>       send data over a socket write to file
depth=<size>            stack trace depth       4
```

```
cutoff=<value>          output cutoff point     0.0001
lineno=y|n              line number in traces?  y
thread=y|n              thread in traces?       n
doe=y|n                 dump on exit?           y
```

NOTE The `cpu` and `monitor` options don't work in JDK 1.4.

2. You must also change a line in the Sun ONE Application Server start script. The start script file is *instance_dir*`/startserv`. Change the following line:

   ```
   PRODUCT_BIN=appservd-wdog
   ```

   to this:

   ```
   PRODUCT_BIN=appservd
   ```

3. Start the server by running the start script. Since the server runs in the foreground (the change in step 2), the command prompt returns only after the server has been stopped.

4. In another window or terminal, find the process ID of the server process.

   ```
   % ps -ef | grep appservd
   ```

   This command lists two `appservd` processes. Look at the PPID (parent process ID) column and identify which of the two processes is the parent process and which is the child process. Note the PID (process ID) of the child process ID.

5. Send a SIGQUIT signal (signal 3) to the child process:

   `% kill -QUIT` *child_PID*

6. To stop the Application Server, run the stop script from another window.

   ```
   % ./stopserv
   ```

   This writes an HPROF stack dump to the file you specified using the `file` HPROF option in Step 1. For general information about using a stack dump, see "Generating a Stack Trace for Debugging," on page 138.

7. Undo the changes in steps 1 and 2 to return your Application Server to its original configuration.

The Optimizeit Profiler

You can purchase Optimizeit™ 4.2 from Intuitive Systems at:

`http://www.optimizeit.com/index.html`

Once Optimizeit is installed using the following instructions, its libraries are loaded into the server process.

To enable remote profiling with Optimizeit, do one of the following:

- Go to the server instance page in the Administration interface, select the JVM Settings tab, select the Profiler option, and edit the following fields before selecting Save:
  - Name: `optimizeit`
  - Enabled: `true`
  - Classpath: *Optimizeit_dir*`/lib/optit.jar`
  - Native Library Path: *Optimizeit_dir*`/lib`
  - JVM Option: For each of these options, type the option in the JVM Option field, select Add, then check its box in the JVM Options list:
    - `-DOPTITHOME=`*Optimizeit_dir*
    - `-Xrunoii`
    - `-Xbootclasspath/a:`*Optimizeit_dir*`/lib/oibcp.jar`
- Edit the `server.xml` file as appropriate:

```
<!-- Optimizeit options -->
<profiler name="optimizeit" classpath="Optimizeit_dir/lib/optit.jar"
       native-library-path="Optimizeit_dir/lib" enabled="true">
   <jvm-options>
       -DOPTIT_HOME=Optimizeit_dir -Xboundthreads -Xrunoii
       -Xbootclasspath/a:Optimizeit_dir/lib/oibcp.jar
   </jvm-options>
</profiler>
```

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

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

For more information about the `server.policy` file, see “The server.policy File,” on page 57.

When the server starts up with this configuration, you can attach the profiler. For further details, see the Optimizeit documentation.

NOTE If any of the configuration options are missing or incorrect, the profiler may experience problems that affect the performance of the Sun ONE Application Server.

The Wily Introscope Profiler

Information about Introscope® from Wily Technology is available at:

`http://www.wilytech.com/solutions_introscope.html`

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

To enable remote profiling with Introscope edit the `server.xml` file as appropriate:

```
<!-- Introscope options. For Win2K, use ; in classpath -->
<java-config ... bytecode-preprocessors" value="S1ASAutoProbe" ... >
   <profiler name="wily" enabled="true"
          classpath="Wily_dir/ProbeBuilder.jar:Wily_dir/Agent.jar" >
   </profiler>
</java-config>
```

When the server starts up with this configuration, you can attach the profiler. For further details, see the Introscope documentation.

NOTE If any of the configuration options are missing or incorrect, the profiler may experience problems that affect the performance of the Sun ONE Application Server.

The JProbe Profiler

Information about JProbe™ from Sitraka is available at:

`http://www.klgroup.com/software/jprobe/`

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

To enable remote profiling with JProbe:

1. Install JProbe 3.0.1.1. This version supports JDK 1.4. For details, see the JProbe documentation.
2. Configure Sun ONE Application Server in one of these ways:

   - Go to the server instance page in the Administration interface, select the JVM Settings tab, type a path to JDK 1.4.0 or 1.4.0_01 in the Java Home field, and select Save.

     Select the Profiler option, and edit the following fields before selecting Save and restarting the server:

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

       `-Xbootclasspath/p:`*JProbe_dir*`/profiler/jpagent.jar`

       `-Xrunjprobeagent`

       `-Xnoclassgc`

   - Edit the `server.xml` file as appropriate, then restart the server:

     ```
     <java-config java-home="JDK_path" ...>
         <profiler name="jprobe" enabled="true"
                native-library-path="JProbe_dir/profiler" >
            <jvm-options>
                -Xbootclasspath/p:JProbe_dir/profiler/jpagent.jar
                -Xrunjprobeagent -Xnoclassgc
            </jvm-options>
         </profiler>
     </java-config>
     ```

     The *JDK_path* must point to JDK 1.4.0 or 1.4.0_01.

NOTE JProbe does not work with JDK 1.4.0_02, which is bundled with Sun ONE Application Server.

NOTE If any of the configuration options are missing or incorrect, the profiler may experience problems that affect the performance of the Sun ONE Application Server.

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

3. Set the following environment variable:

   `JPROBE_ARGS_0=-jp_input=`*JPL_file_path*

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

4. Start the server instance.

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

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

   a. Select Server Side for the type of application.

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

      - Target Server - *other_server*
      - Server home Directory - *install_dir*
      - Server class File - `com.iplanet.ias.server.J2EERunner`
      - Working Directory - *install_dir*
      - Classpath - *install_dir*`/lib/appserv-rt.jar`
      - Source File Path - *source_code_dir* (in case you want to get the line level details)
      - Server class arguments - (optional)
      - Main Package - `com.iplanet.ias.server`

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

Chapter 6

Developing Lifecycle Listeners

Lifecycle listener modules provide a means of running short or long duration Java-based tasks within the application server environment, such as instantiation of singletons or RMI servers. These modules are automatically initiated at server startup and are notified at various phases of the server life cycle.

The following sections describe how to create and use a lifecycle module:

- Server Life Cycle Events
- The LifecycleListener Interface
- The LifecycleEvent Class
- The Server Lifecycle Event Context
- Assembling and Deploying a Lifecycle Module
- Considerations for Lifecycle Modules

Server Life Cycle Events

A lifecycle module listens for and performs its tasks in response to the following events in the server life cycle:

- During the `INIT_EVENT`, the server reads the configuration, initializes built-in subsystems (such as security and logging services), and creates the containers.
- During the `STARTUP_EVENT`, the server loads and initializes deployed applications.
- During the `READY_EVENT`, the server is ready to service requests.
- During the `SHUTDOWN_EVENT`, the server destroys loaded applications and stops.

- During the TERMINATION_EVENT, the server closes the containers, the built-in subsystems, and the server runtime environment.

These events are defined in the LifecycleEvent class.

The lifecycle modules that listen for these events implement the LifecycleListener interface and are configured in the server.xml file.

The LifecycleListener Interface

To create a lifecycle module is to configure a customized class that implements the com.sun.appserv.server.LifecycleListener interface. You can create and simultaneously execute multiple lifecycle modules.

The LifecycleListener interface defines this method:

- public void handleEvent(com.sun.appserv.server.LifecycleEvent event) throws ServerLifecycleException

  This method responds to a lifecycle event and throws a com.sun.appserv.server.ServerLifecycleException if an error occurs.

A sample implementation of the LifecycleListener interface is the LifecycleListenerImpl.java file, which you can use for testing lifecycle events:

```
package com.sun.appserv.server;

import java.util.Properties;

/**
 *  LifecycleListenerImpl is a dummy implementation for the LifecycleListener
 * interface.This implementaion stubs out various lifecycle interface methods.
 */

public class LifecycleListenerImpl implements LifecycleListener {

   /** receive a server lifecycle event
    *  @param event associated event
    *  @throws <code>ServerLifecycleException</code> for exceptional condition.
    *
    *  Configure this module as a lifecycle-module in server.xml:
    *
    *  <applications>
    *     <lifecycle-module name="test"
    *               class-name="com.sun.appserv.server.LifecycleListenerImpl"
    *               is-failure-fatal="false">
    *        <property name="foo" value="fooval"/>
```

```
 *     </lifecycle-module>
 *  </applications>
 *
 *  Set<code>is-failure-fatal</code>in server.xml to <code>true</code> for
 *  fatal conditions.
 */
public void handleEvent(LifecycleEvent event) throws ServerLifecycleException
{
   LifecycleEventContext context = event.getLifecycleEventContext();

   context.log("got event" + event.getEventType() + " event data: "
      + event.getData());

   Properties props;

   if (LifecycleEvent.INIT_EVENT == event.getEventType()) {
      context.log("LifecycleListener: INIT_EVENT");

      props = (Properties) event.getData();

      // handle INIT_EVENT
      return;
   }

   if (LifecycleEvent.STARTUP_EVENT == event.getEventType()) {
      context.log("LifecycleListener: STARTUP_EVENT");

      // handle STARTUP_EVENT
      return;
   }

   if (LifecycleEvent.READY_EVENT == event.getEventType()) {

      context.log("LifecycleListener: READY_EVENT");

      // handle READY_EVENT
      return;
   }

   if (LifecycleEvent.SHUTDOWN_EVENT== event.getEventType()) {
      context.log("LifecycleListener: SHUTDOWN_EVENT");

      // handle SHUTDOWN_EVENT
      return;
   }

   if (LifecycleEvent.TERMINATION_EVENT == event.getEventType()) {
```

```
            context.log("LifecycleListener: TERMINATE_EVENT");

            // handle TERMINATION_EVENT
            return;
        }
    }
}
```

The LifecycleEvent Class

The `com.sun.appserv.server.LifecycleEvent` class defines a server life cycle event. The following methods are associated with the event:

- `public java.lang.Object getData()`

  This method returns the data associated with the event.

- `public int getEventType()`

  This method returns the event type, which is `INIT_EVENT`, `STARTUP_EVENT`, `READY_EVENT`, `SHUTDOWN_EVENT`, or `TERMINATION_EVENT`.

- `public com.sun.appserv.server.LifecycleEventContext getLifecycleEventContext()`

  This method returns the lifecycle event context, described next.

A `LifecycleEvent` instance is passed to the `LifecycleListener.handleEvent` method.

The Server Lifecycle Event Context

The `com.sun.appserv.server.LifecycleEventContext` interface exposes runtime information about the server. The lifecycle event context is created when the `LifecycleEvent` class is instantiated at server initialization. The `LifecycleEventContext` interface defines these methods:

- `public java.lang.String[] getCmdLineArgs()`

  This method returns the server startup command-line arguments.

- `public java.lang.String getInstallRoot()`

  This method returns the server installation root directory.

- `public java.lang.String getInstanceName()`

  This method returns the server instance name.

- `public javax.naming.InitialContext getInitialContext()`

  This method returns the initial JNDI naming context. The naming environment for lifecycle modules is installed during the `STARTUP_EVENT`. A lifecycle module can look up any resource defined in the `server.xml` file by its `jndi-name` attribute after the `STARTUP_EVENT` is complete.

NOTE To avoid collisions with names of other enterprise resources in JNDI, and to avoid portability problems, all names in a Sun ONE Application Server lifecycle module should begin with the string `java:comp/env`.

If a lifecycle module needs to look up beans, it can do so in the `READY_EVENT`. It can use the `getInitialContext()` method to get the initial context to which all the resources are bound.

- `public void log(java.lang.String message)`

  This method writes the specified message to the server log file. The `message` parameter is a String specifying the text to be written to the log file.

- `public void log(java.lang.String message, java.lang.Throwable throwable)`

  This method writes an explanatory message and a stack trace for a given `Throwable` exception to the server log file. The `message` parameter is a String that describes the error or exception. The `throwable` parameter is the `Throwable` error or exception.

Assembling and Deploying a Lifecycle Module

You assemble a lifecycle module as described in “Assembling a Lifecycle Module,” on page 89. You deploy a lifecycle module as described in “Deploying a Lifecycle Module,” on page 100.

During lifecycle module deployment, a `lifecycle-module` element is created in the `server.xml` file. You can edit this file to change its configuration. The `property` subelement allows you to specify input parameters. For example:

```
<lifecycle-module   name="customStartup"
                    enabled="true"
                    class-name="com.acme.CustomStartup"
                    classpath="/apps/customStartup"
                    load-order="200"
                    is-failure-fatal="true">
   <description>custom startup module to do my tasks</description>
   <property name="rmiServer" value="acme1:7070" />
   <property name="timeout" value="30" />
</lifecycle-module>
```

Note that if `is-failure-fatal` is set to `true` (the default is `false`), lifecycle module failure prevents server initialization or startup, but not shutdown or termination.

For more information about the `server.xml` file, see the *Sun ONE Application Server Administrator's Configuration File Reference*.

After you deploy a lifecycle module, you must restart the server to activate it. The server instantiates it and registers it as a lifecycle event listener at server initialization.

Considerations for Lifecycle Modules

The resources allocated during initialization or startup should be freed during shutdown or termination. The lifecycle module classes are called synchronously from the main server thread, therefore it is important to ensure that these classes don't block the server. Lifecycle modules may create threads if appropriate, but these threads must be stopped in the shutdown and termination phases.

The LifeCycleModule Classloader is the parent classloader for lifecycle modules. Each lifecycle module's `classpath` in `server.xml` is used to construct its classloader. All the support classes needed by a lifecycle module must be available to the LifeCycleModule Classloader or its parent, the Shared Classloader. (The Shared Classloader loads server-wide resources.)

You must ensure that the `server.policy` file is appropriately set up, or a lifecycle module trying to perform a `System.exec()` may cause a security access violation. For details, see "The server.policy File," on page 57.

The configured properties for a lifecycle module are passed as properties in the `INIT_EVENT`. The JNDI naming context is not available in the `INIT_EVENT`. If a lifecycle module requires the naming context, it can get this in the `STARTUP_EVENT`, `READY_EVENT`, or `SHUTDOWN_EVENT`.

Glossary

This glossary provides definitions for common terms used to describe the Sun ONE Application Server deployment and development environment. For a glossary of standard J2EE terms, please see the J2EE glossary at:

```
http://java.sun.com/j2ee/glossary.html
```

access control The means of securing your Sun ONE Application Server by controlling who and what has access to it.

ACL Access Control List. ACLs are text files that contain lists identifying who can access the resources stored on your Sun ONE Application Server. *See also* general ACL.

activation The process of transferring an enterprise bean's state from secondary storage to memory.

Administration interface The set of browser based forms used to configure and administer the Sun ONE Application Server. S*ee also* CLI.

administration server An application server instance dedicated to providing the administrative functions of the Sun ONE Application Server, including deployment, browser-based administration, and access from the command-line interface (CLI) and Integrated Development Environment (IDE).

administrative domain Multiple administrative domains is a feature within the Sun ONE Application Server that allows different administrative users to create and manage their own domains. A domain is a set of instances, created using a common set of installed binaries in a single system.

API Applications Program Interface. A set of instructions that a computer program can use to communicate with other software or hardware that is designed to interpret that API.

applet A small application written in Java that runs in a web browser. Typically, applets are called by or embedded in web pages to provide special functionality. By contrast, a *servlet* is a small application that runs on a server.

application A group of components packaged into an `.ear` file with a J2EE application deployment descriptor. *See also* component, module.

application client container *See* container.

application server A reliable, secure, and scalable software platform in which business applications are run. Application servers typically provide high-level services to applications, such as component lifecycle, location, and distribution and transactional resource access,

application tier A conceptual division of a J2EE application:

client tier: The user interface (UI). End users interact with client software (such as a web browser) to use the application.

server tier: The business logic and presentation logic that make up your application, defined in the application's components.

data tier: The data access logic that enables your application to interact with a data source.

assembly The process of combining discrete components of an application into a single unit that can be deployed. S*ee also* deployment.

asynchronous communication A mode of communication in which the sender of a message need not wait for the sending method to return before it continues with other work.

attribute A name-value pair in a request object that can be set by a servlet. Also a name-value pair that modifies an element in an XML file. Contrast with *parameter*. More generally, an attribute is a unit of metadata.

auditing The method(s) by which significant events are recorded for subsequent examination, typically in error or security breach situations.

authentication The process by which an entity (such as a user) proves to another entity (such as an application) that it is acting on behalf of a specific identity (the user's security identity). Sun ONE Application Server supports basic, form-based, and SSL mutual authentication. *See also* client authentication, digest authentication, host-IP authentication, pluggable authentication.

authorization The process by which access to a method or resource is determined. Authorization in the J2EE platform depends upon whether the user associated with a request through authentication is in a given security role. For example, a human resources application may authorize managers to view personal employee information for all employees, but allow employees to only view their own personal information.

backup store A repository for data, typically a file system or database. A backup store can be monitored by a background thread (or sweeper thread) to remove unwanted entries.

bean-managed persistence Data transfer between an entity bean's variables and a data store. The data access logic is typically provided by a developer using Java Database Connectivity (JDBC) or other data access technologies. *See also* container-managed persistence.

bean-managed transaction Where transaction demarcation for an enterprise bean is controlled programmatically by the developer. *See also* container-managed transaction.

BLOB Binary Large OBject. A data type used to store and retrieve complex object fields. BLOBs are binary or serializable objects, such as pictures, that translate into large byte arrays, which are then serialized into container-managed persistence fields.

BMP *See* bean-managed persistence.

BMT *See* bean-managed transaction.

broker The Sun ONE Message Queue entity that manages JMS message routing, delivery, persistence, security, and logging, and which provides an interface that allows an administrator to monitor and tune performance and resource use.

business logic The code that implements the essential business rules of an application, rather than data integration or presentation logic.

CA See certificate authority or connector architecture.

cached rowset A `CachedRowSet` object permits you to retrieve data from a data source, then detach from the data source while you examine and modify the data. A cached row set keeps track both of the original data retrieved, and any changes made to the data by your application. If the application attempts to update the original data source, the row set is reconnected to the data source, and only those rows that have changed are merged back into the database.

Cache Control Directives Cache-control directives are a way for Sun ONE Application Server to control what information is cached by a proxy server. Using cache-control directives, you override the default caching of the proxy to protect sensitive information from being cached, and perhaps retrieved later. For these directives to work, the proxy server must comply with HTTP 1.1.

callable statement A class that encapsulates a database procedure or function call for databases that support returning result sets from stored procedures.

certificate Digital data that specifies the name of an individual, company, or other entity, and certifies that the public key included in the certificate belongs to that entity. Both clients and servers can have certificates.

certificate authority A company that sells certificates over the Internet, or a department responsible for issuing certificates for a company's intranet or extranet.

cipher A cryptographic algorithm (a mathematical function), used for encryption or decryption.

CKL Compromised Key List. A list, published by a certificate authority, that indicates any certificates that either client users or server users should no longer trust. In this case, the key has been compromised. *See also* CRL.

classloader A Java component responsible for loading Java classes according to specific rules. *See also* classpath.

classpath A path that identifies directories and JAR files where Java classes are stored. *See also* classloader.

CLI Command-line interface. An interface that enables you to type executable instructions at a user prompt. S*ee also* Administration interface.

client authentication The process of authenticating client certificates by cryptographically verifying the certificate signature and the certificate chain leading to the CA on the trust CA list. See also authentication, certificate authority.

client contract A contract that determines the communication rules between a client and the EJB container, establishes a uniform development model for applications that use enterprise beans, and guarantees greater reuse of beans by standardizing the relationship with the client.

CMP *See* container-managed persistence.

CMR *See* container-managed relationship.

CMT *See* container-managed transaction.

co-locate To position a component in the same memory space as a related component in order avoid remote procedure calls and improve performance.

column A field in a database table.

commit To complete a transaction by sending the required commands to the database. *See* rollback, transaction.

component A web application, enterprise bean, message-driven bean, application client, or connector. *See also* application, module.

component contract A contract that establishes the relationship between an enterprise bean and its container.

configuration The process of tuning the server or providing metadata for a component. Normally, the configuration for a specific component is kept in the component's deployment descriptor file. *See also* administration server, deployment descriptor.

connection factory An object that produces connection objects that enable a J2EE component to access a resource. Used to create JMS connections (TopicConnection or QueueConnection) which allow application code to make use of the provided JMS implementation. Application code uses the JNDI Service to locate connection factory objects using a JNDI Name.

Connection Pool allows highly efficient access to a database by caching and reusing physical connections, thus avoiding connection overhead and allowing a small number of connections to be shared between a large number of threads. *See also* JDBC connection pool

connector A standard extension mechanism for containers to provide connectivity to EISs. A connector is specific to an EIS and consists of a resource adapter and application development tools for EIS connectivity. The resource adapter is plugged in to a container through its support for system level contracts defined in the connector architecture.

connector architecture An architecture for the integration of J2EE applications with EISs. There are two parts to this architecture: a EIS vendor-provided resource adapter and a J2EE server that allows this resource adapter to plug in. This architecture defines a set of contracts that a resource adapter has to support to plug in to a J2EE server, for example, transactions, security and resource management.

container An entity that provides life cycle management, security, deployment, and runtime services to a specific type of J2EE component. Sun ONE Application Server provides web and EJB containers, and supports application client containers. *See also* component.

container-managed persistence Where the EJB container is responsible for entity bean persistence. Data transfer between an entity bean's variables and a data store, where the data access logic is provided by the Sun ONE Application Server. *See also* bean-managed persistence.

container-managed relationship A relationship between fields in a pair of classes where operations on one side of the relationship affect the other side.

container-managed transaction Where transaction demarcation for an enterprise bean is specified declaratively and automatically controlled by the EJB container *See also* bean-managed transaction.

control descriptor A set of enterprise bean configuration entries that enable you to specify optional individual property overrides for bean methods, plus enterprise bean transaction and security properties.

conversational state Where the state of an object changes as the result of repeated interactions with the same client. *See also* persistent state.

cookie A small collection of information that can be transmitted to a calling web browser, then retrieved on each subsequent call from that browser so the server can recognize calls from the same client. Cookies are domain-specific and can take advantage of the same web server security features as other data interchange between your application and the server.

CORBA Common Object Request Broker Architecture. A standard architecture definition for object-oriented distributed computing.

COSNaming Service An an IIOP-based naming service.

CosNaming provider To support a global JNDI name space (accessible to IIOP application clients), Sun ONE Application Server includes J2EE based CosNaming provider which supports binding of CORBA references (remote EJB references).

create method A method for customizing an enterprise bean at creation.

CRL Certificate Revocation List. A list, published by a certificate authority, that indicates any certificates that either client users or server users should no longer trust. In this case, the certificate has been revoked. *See also* CKL.

data access logic Business logic that involves interacting with a data source.

database A generic term for Relational Database Management System (RDBMS). A software package that enables the creation and manipulation of large amounts of related, organized data.

database connection A database connection is a communication link with a database or other data source. Components can create and manipulate several database connections simultaneously to access data.

data source A handle to a source of data, such as a database. Data sources are registered with the iPlanet Application Server and then retrieved programmatically in order to establish connections and interact with the data source. A data source definition specifies how to connect to the source of data.

DataSource Object A DataSource object has a set of properties that identify and describe the real world data source that it represents.

declarative security Declaring security properties in the component's configuration file and allowing the component's container (for instance, a bean's container or a servlet engine) to manage security implicitly. This type of security requires no programmatic control. Opposite of programmatic security. *See* container-managed persistence.

declarative transaction See container-managed transaction.

decryption The process of transforming encrypted information so that it is intelligible again.

delegation An object-oriented technique for using the composition of objects as an implementation strategy. One object, which is responsible for the result of an operation, delegates the implementation to another object, its delegatee. For example, a classloader often delegates the loading of some classes to its parent.

deployment The process of distributing the files required by an application to an application server to make the application available to run on the application server. *See also* assembly.

deployment descriptor An XML file provided with each module and application that describes how they should be deployed. The deployment descriptor directs a deployment tool to deploy a module or application with specific container options and describes specific configuration requirements that a deployer must resolve.

destination resource An objects that represents Topic or Queue destinations. Used by applications to read / write to Queues or publish / subscribe to Topics. Application code uses the JNDI Service to locate JMS resource objects using a JNDI Name.

digest authentication A for of authentication that allows the user to authenticate based on user name and password without sending the user name and password as cleartext.

digital signature an electronic security mechanism used to authenticate both a message and the signer.

directory server See Sun ONE Directory Server.

Distinguished Name *See* DN, DN attribute.

distributable session A user session that is distributable among all servers in a cluster.

distributed transaction A single transaction that can apply to multiple heterogeneous databases that may reside on separate servers.

Document Root The document root (sometimes called the primary document directory) is the central directory that contains all the virtual server's files you want to make available to remote clients.

Domain Registry The Domain Registry is a single data structure that contains domain-specific information, for all the domains created and configured on an installation of Sun ONE Application Server, such as domain name, domain location, domain port, domain host.

DN Distinguished Name. The string representation for the name of an entry in a directory server.

DN attribute Distinguished Name attribute. A text string that contains identifying information for an associated user, group, or object.

DTD Document Type Definition. A description of the structure and properties of a class of XML files.

dynamic redeployment The process of redeploying a component without restarting the server.

dynamic reloading The process of updating and reloading a component without restarting the server. By default, servlet, JavaServer Page (JSP), and enterprise bean components can be dynamically reloaded. Also known as versioning.

EAR file Enterprise ARchive file. An archive file that contains a J2EE application. EAR files have the `.ear` extension. *See also* JAR file.

e-commerce Electronic commerce. A term for business conducted over the Internet.

EIS Enterprise Information System. This can be interpreted as a packaged enterprise application, a transaction system, or a user application. Often referred to as an EIS. Examples of EISs include: R/3, PeopleSoft, Tuxedo, and CICS.

EJB container *See* container.

EJB QL EJB Query Language. A query language that provides for navigation across a network of entity beans defined by container-managed relationships.

EJB technology An enterprise bean is a server-side component that encapsulates the business logic of an application. The business logic is the code that fulfills the purpose of the application. In an inventory control application, for example, the enterprise beans might implement the business logic in methods called `checkInventoryLevel` and `orderProduct`. By invoking these methods, remote clients can access the inventory services provided by the application. *See also* container, entity bean, message-driven bean, and session bean.

ejbc utility The compiler for enterprise beans. It checks all EJB classes and interfaces for compliance with the EJB specification, and generates stubs and skeletons.

element A member of a larger set; for example, a data unit within an array, or a logic element. In an XML file, it is the basic structural unit. An XML element contains subelements or data, and may contain attributes.

encapsulate To localize knowledge within a module. Because objects encapsulate data and implementation, the user of an object can view the object as a black box that provides services. Instance variables and methods can be added, deleted, or changed, but if the services provided by the object remain the same, code that uses the object can continue to use it without being rewritten.

encryption The process of transforming information so it is unintelligible to anyone but the intended recipient.

entity bean An enterprise bean that relates to physical data, such as a row in a database. Entity beans are long lived, because they are tied to persistent data. Entity beans are always transactional and multi-user aware. *See* message-driven bean, read-only bean, session bean.

ERP Enterprise Resource Planning. A multi-module software system that supports enterprise resource planning. An ERP system typically includes a relational database and applications for managing purchasing, inventory, personnel, customer service, shipping, financial planning, and other important aspects of the business.

event A named action that triggers a response from a module or application.

external JDNI resource Allows the JNDI Service to act as a bridge to a remote JNDI server.

facade Where an application-specific stateful session bean is used to manage various Enterprise JavaBeans (EJBs).

factory class A class that creates persistence managers. *See also* connection factory.

failover A recovery process where a bean can transparently survive a server crash.

finder method Method which enables clients to look up a bean or a collection of beans in a globally available directory.

File Cache The file cache contains information about files and static file content. The file cache is turned on by default.

firewall an electronic boundary that allows a network administrator to restrict the flow of information across networks in order to enforce security.

form action handler A specially defined method in servlet or application logic that performs an action based on a named button on a form.

FQDN Fully Qualified Domain Name. The full name of a system, containing its hostname and its domain name.

general ACL A named list in the Sun ONE Directory Server that relates a user or group with one or more permissions. This list can be defined and accessed arbitrarily to record any set of permissions.

generic servlet A servlet that extends `javax.servlet.GenericServlet`. Generic servlets are protocol-independent, meaning that they contain no inherent support for HTTP or any other transport protocol. Contrast with HTTP servlet.

global database connection A database connection available to multiple components. Requires a resource manager.

global transaction A transaction that is managed and coordinated by a transaction manager and can span multiple databases and processes. The transaction manager typically uses the XA protocol to interact with the database backends. *See* local transaction.

granularity level The approach to dividing an application into pieces. A *high level of granularity* means that the application is divided into many smaller, more narrowly defined Enterprise JavaBeans (EJBs). A *low level of granularity* means the application is divided into fewer pieces, producing a larger program.

group A group of users that are related in some way. Group membership is usually maintained by a local system administrator. *See* user, role.

handle An object that identifies an enterprise bean. A client may serialize the handle, and then later deserialize it to obtain a reference to the bean.

Heuristic Decision The transactional mode used by a particular transaction. A transaction has to either Commit or Rollback.

home interface A mechanism that defines the methods that enable a client to create and remove an enterprise bean.

host-IP authentication A security mechanism used for of limiting access to the Administration Server, or the files and directories on a web site by making them available only to clients using specific computers.

HTML Hypertext Markup Language. A coding markup language used to create documents that can be displayed by web browsers. Each block of text is surrounded by codes that indicate the nature of the text.

HTML page A page coded in HTML and intended for display in a web browser.

HTTP Hypertext Transfer Protocol. The Internet protocol that fetches hypertext objects from remote hosts. It is based on TCP/IP.

HTTP servlet A servlet that extends `javax.servlet.HttpServlet`. These servlets have built-in support for the HTTP protocol. Contrast with generic servlet.

HTTPS HyperText Transmission Protocol, Secure. HTTP for secure transactions.

IDE Integrated Development Environment. Software that allows you to create, assemble, deploy, and debug code from a single, easy-to-use interface.

IIOP Internet Inter-ORB Protocol. Transport-level protocol used by both Remote Method Invocation (RMI) over IIOP and Common Object Request Broker Architecture (CORBA).

IIOP Listener The IIOP listener is a listen socket that listens on a specified port and accepts incoming connections from CORBA based client application

IMAP Internet Message Access Protocol.

IP address A structured, numeric identifier for a computer or other device on a TCP/IP network. The format of an IP address is a 32-bit numeric address written as four numbers separated by periods. Each number can be zero to 255. For example, 123.231.32.2 could be an IP address.

isolation level *See* transaction isolation level.

J2EE Java 2 Enterprise Edition. An environment for developing and deploying multi-tiered, web-based enterprise applications. The J2EE platform consists of a set of services, application programming interfaces (APIs), and protocols that provide the functionality for developing these applications.

JAF The JavaBeans Activation Framework (JAF) integrates support for MIME data types into the Java platform. See Mime Types.

JAR file Java ARchive file. A file used for aggregating many files into one file. JAR files have the.`jar` extension.

JAR file contract Java ARchive contract that specifies what information must be in the enterprise bean package.

JAR file format Java ARchive file format. A platform-independent file format that aggregates many files into one file. Multiple applets and their requisite components (class files, images, sounds, and other resource files) can be bundled in a JAR file and subsequently downloaded to a browser in a single HTTP transaction. The JAR files format also supports file compression and digital signatures.

JavaBean A portable, platform-independent reusable component model.

Java IDL Java Interface Definition Language. APIs written in the Java programming language that provide a standards-based compatibility and connectivity with Common Object Request Broker Architecture (CORBA).

JavaMail session An object used by an application to interact with a mail store. Application code uses the JNDI Service to locate JavaMail session resources objects using a JNDI name.

JAXM Java API for XML Messaging. Enables applications to send and receive document-oriented XML messages using the SOAP standard. These messages can be with or without attachments.

JAXP Java API for XML Processing. A Java API that supports processing of XML documents using DOM, SAX, and XSLT. Enables applications to parse and transform XML documents independent of a particular XML processing implementation.

JAXR Java API for XML Registry. Provides a uniform and standard Java API for accessing different kinds of XML registries. Enables users to build, deploy and discover web services.

JAX-RPC Java API for XML-based Remote Procedure Calls. Enables developers to build interoperable web applications and web services based on XML-based RPC protocols.

JDBC Java Database Connectivity. A standards-based set of classes and interfaces that enable developers to create data-aware components. JDBC implements methods for connecting to and interacting with data sources in a platform- and vendor-independent way.

JDBC connection pool A pool that combines the JDBC data source properties used to specify a connection to a database with the connection pool properties.

JDBC resource A resource used to connect an application running within the application server to a database using an existing JDBC connection pool. Consists of a JNDI name (which is used by the application) and the name of an existing JDBC connection pool.

JDK Java Development Kit. The software that includes the APIs and tools that developers need to build applications for those versions of the Java platform that preceded the Java 2 Platform. *See also* JDK.

JMS Java Message Service. A standard set of interfaces and semantics that define how a JMS client accesses the facilities of a JMS message service. These interfaces provide a standard way for Java programs to create, send, receive, and read messages.

JMS-administered object A pre-configured JMS object—a connection factory or a destination—created by an administrator for use by one or more JMS clients.

The use of administered objects allows JMS clients to be provider-independent; that is, it isolates them from the proprietary aspects of a provider. These objects are placed in a JNDI name space by an administrator and are accessed by JMS clients using JNDI lookups.

JMS client An application (or software component) that interacts with other JMS clients using a JMS message service to exchange messages.

JMS connection factory The JMS administered object a JMS client uses to create a connection to a JMS message service.

JMS destination The physical destination in a JMS message service to which produced messages are delivered for routing and subsequent delivery to consumers. This physical destination is identified and encapsulated by an JMS administered object that a JMS client uses to specify the destination for which it is producing messages and/or from which it is consuming messages.

JMS messages Asynchronous requests, reports, or events that are consumed by JMS clients. A message has a header (to which additional fields can be added) and a body. The message header specifies standard fields and optional properties. The message body contains the data that is being transmitted.

JMS provider A product that implements the JMS interfaces for a messaging system and adds the administrative and control functions needed for a complete product.

JMS Service Software that provides delivery services for a JMS messaging system, including connections to JMS clients, message routing and delivery, persistence, security, and logging. The message service maintains physical destinations to which JMS clients send messages, and from which the messages are delivered to consuming clients.

JNDI Java Naming and Directory Interface. This is a standard extension to the Java platform, providing Java technology-enabled applications with a unified interface to multiple naming and directory services in the enterprise. As part of the Java Enterprise API set, JNDI enables seamless connectivity to heterogeneous enterprise naming and directory services.

JNDI name A name used to access a resource that has been registered in the JNDI naming service.

JRE Java Runtime Environment. A subset of the Java Development Kit (JDK) consisting of the Java virtual machine, the Java core classes, and supporting files that provides runtime support for applications written in the Java programming language. *See also* JDK.

JSP JavaServer Page. A text page written using a combination of HTML or XML tags, JSP tags, and Java code. JSPs combine the layout capabilities of a standard browser page with the power of a programming language.

jspc utility The compiler for JSPs. It checks all JSPs for compliance with the JSP specification.

JTA Java Transaction API. An API that allows applications and J2EE servers to access transactions.

JTS Java Transaction Service. The Java service for processing transactions.

key-pair file *See* trust database.

LDAP Lightweight Directory Access Protocol. LDAP is an open directory access protocol that runs over TCP / IP. It is scalable to a global size and millions of entries. Using Sun ONE Directory Server, a provided LDAP server, you can store all of your enterprise's information in a single, centralized repository of directory information that any application server can access through the network.

LDIF LDAP Data Interchange Format. Format used to represent Sun ONE Directory Server entries in text form.

lifecycle event A stage in the server life cycle, such as startup or shutdown.

lifecycle module A module that listens for and performs its tasks in response to events in the server life cycle.

Listener A class, registered with a posting object, that says what to do when an event occurs.

local database connection The transaction context in a local connection is local to the current process and to the current data source, not distributed across processes or across data sources.

local interface An interface that provides a mechanism for a client that is located in the same Java Virtual Machine (JVM) with a session or entity bean to access that bean.

local session A user session that is only visible to one server.

local transaction A transaction that is native to one database and is restricted within a single process. Local transactions work only against a single backend. Local transactions are typically demarcated using JDBC APIs. *See also* global transaction.

mapping The ability to tie an object-oriented model to a relational model of data, usually the schema of a relational database. The process of converting a schema to a different structure. Also refers to the mapping of users to security roles.

MDB *See* message-driven bean.

message-driven bean An enterprise bean that is an asynchronous message consumer. A message-driven bean has no state for a specific client, but its instance variables may contain state across the handling of client messages, including an open database connection and an object reference to an EJB object. A client accesses a message-driven bean by sending messages to the destination for which the message-driven bean is a message listener.

messaging A system of asynchronous requests, reports, or events used by enterprise applications that allows loosely coupled applications to transfer information reliably and securely.

metadata Information about a component, such as its name, and specifications for its behavior.

management information base (MIB) A tree-like structure that defines the variables the master SNMP agent can access. The MIB provides access to the HTTP server's network configuration, status, and statistics. Using SNMP, you can view this information from the network management workstation (NMS). *See also* network management station (NMS) and SNMP.

MIME Data Type MIME (Multi-purpose Internet Mail Extension) types control what types of multimedia files your system supports.

module A web application, enterprise bean, message-driven bean, application client, or connector that has been deployed individually, outside an application. *See also* application, component, lifecycle module.

network management station (NMS) A machine used to remotely manage a specific network. Usually, the NMS software will provide a graph to display collected data or use that data to make sure the server is operating within a particular tolerance. *See also* SNMP.

NTV Name, Type, Value.

object persistence *See* persistence.

O/R mapping tool Object-to-relational [database] tool. A mapping tool within the Sun ONE Application Server Administrative interface that creates XML deployment descriptors for entity beans.

package A collection of related classes that are stored in a common directory. They are often literally packaged together in a Java archive JAR file. *See also* assembly, deployment.

parameter A name/value pair sent from the client, including form field data, HTTP header information, and so on, and encapsulated in a request object. Contrast with attribute. More generally, an argument to a Java method or database-prepared command.

passivation A method of releasing a bean's resources from memory without destroying the bean. In this way, a bean is made to be persistent, and can be recalled without the overhead of instantiation.

permission A set of privileges granted or denied to a user or group. *See also* ACL.

persistence For enterprise beans, the protocol for transferring the state of an entity bean between its instance variables and an underlying database. Opposite of transience. For sessions, the session storage mechanism.

persistence manager The entity responsible for the persistence of the entity beans installed in the container.

persistent state Where the state of an object is kept in persistent storage, usually a database.

pluggable authentication A mechanism that allows J2EE applications to use the Java Authentication and Authorization Service (JAAS) feature from the J2SE platform. Developers can plug in their own authentication mechanisms.

point-to-point delivery model Producers address messages to specific queues; consumers extract messages from queues established to hold their messages. A message is delivered to only one message consumer.

pooling The process of providing a number of preconfigured resources to improve performance. If a resource is pooled, a component can use an existing instance from the pool rather than instantiating a new one. In the Sun ONE Application Server, database connections, servlet instances, and enterprise bean instances can all be pooled.

POP3 Post Office Protocol

prepared command A database command (in SQL) that is precompiled to make repeated execution more efficient. Prepared commands can contain parameters. A prepared statement contains one or more prepared commands.

prepared statement A class that encapsulates a `QUERY`, `UPDATE`, or `INSERT` statement that is used repeatedly to fetch data. A prepared statement contains one or more prepared commands.

presentation layout The format of web page content.

presentation logic Activities that create a page in an application, including processing a request, generating content in response, and formatting the page for the client. Usually handled by a web application.

primary key The unique identifier that enables the client to locate a particular entity bean.

primary key class name A variable that specifies the fully qualified class name of a bean's primary key. Used for JNDI lookups.

principal The identity assigned to an entity as a result of authentication.

private key *See* public key cryptography.

process Execution sequence of an active program. A process is made up of one or more threads.

programmatic security The process of controlling security explicitly in code rather than allowing the component's container (for instance, a bean's container or a servlet engine) to handle it. Opposite of declarative security.

programmer-demarcated transaction *See* bean-managed transaction.

property A single attribute that defines the behavior of an application component. In the `server.xml` file, a property is an element that contains a name/value pair.

public key cryptography A form of cryptography in which each user has a public key and a private key. Messages are sent encrypted with the receiver's public key; the receiver decrypts them using the private key. Using this method, the private key never has to be revealed to anyone other than the user.

publish/subscribe delivery model Publishers and subscribers are generally anonymous and may dynamically publish or subscribe to a topic. The system distributes messages arriving from a topic's multiple publishers to its multiple subscribers.

QOS QOS (Quality of Service) refers to the performance limits you set for a server instance or virtual server. For example, if you are an ISP, you might want to charge different amounts of money for virtual servers depending on how much bandwidth is provided. You can limit two areas: the amount of bandwidth and the number of connections.

queue An object created by an administrator to implement the point-to-point delivery model. A queue is always available to hold messages even when the client that consumes its messages is inactive. A queue is used as an intermediary holding place between producers and consumers.

RAR file Resource ARchive. A JAR archive that contains a resource adapter.

RDB Relational database.

RDBMS Relational database management system.

read-only bean An entity bean that is never modified by an EJB client. *See also* entity bean.

realm A scope over which a common security policy is defined and enforced by the security administrator of the security service. Also called a *security policy domain* or *security domain* in the J2EE specification.

remote interface One of two interfaces for an Enterprise JavaBean. The remote interface defines the business methods callable by a client.

request object An object that contains page and session data produced by a client, passed as an input parameter to a servlet or JavaServer Page (JSP).

resource manager An object that acts as a facilitator between a resource such as a database or message broker, and client(s) of the resource such as Sun ONE Application Server processes. Controls globally-available data sources.

resource reference An element in a deployment descriptor that identifies the component's coded name for the resource.

response object An object that references the calling client and provides methods for generating output for the client.

ResultSet An object that implements the `java.sql.ResultSet` interface. `ResultSets` are used to encapsulate a set of rows retrieved from a database or other source of tabular data.

reusable component A component created so that it can be used in more than one capacity, for instance, by more than one resource or application.

RMI Remote Method Invocation. A Java standard set of APIs that enable developers to write remote interfaces that can pass objects to remote processes.

RMIC Remote Method Invocation Compiler.

role A functional grouping of subjects in an application, represented by one or more groups in a deployed environment. *See also* user, group.

rollback Cancellation of a transaction.

row A single data record that contains values for each column in a table.

RowSet An object that encapsulates a set of rows retrieved from a database or other source of tabular data. `RowSet` extends the `java.sql.ResultSet` interface, enabling `ResultSet` to act as a JavaBeans component.

RPC Remote Procedure Call. A mechanism for accessing a remote object or service.

runtime system The software environment in which programs run. The runtime system includes all the code necessary to load programs written in the Java programming language, dynamically link native methods, manage memory, and handle exceptions. An implementation of the Java virtual machine is included, which may be a Java interpreter.

SAF Server Application Function. A function that participates in request processing and other server activities

schema The structure of the underlying database, including the names of tables, the names and types of columns, index information, and relationship (primary and foreign key) information.

Secure Socket Layer *See* SSL.

security A screening mechanism that ensures that application resources are only accessed by authorized clients.

serializable object An object that can be deconstructed and reconstructed, which enables it to be stored or distributed among multiple servers.

server instance A Sun ONE Application Server can contain multiple instances in the same installation on the same machine. Each instance has its own directory structure, configuration, and deployed applications. Each instance can also contain multiple virtual servers. *See also* virtual server.

servlet An instance of the `Servlet` class. A servlet is a reusable application that runs on a server. In the Sun ONE Application Server, a servlet acts as the central dispatcher for each interaction in an application by performing presentation logic, invoking business logic, and invoking or performing presentation layout.

servlet engine An internal object that handles all servlet metafunctions. Collectively, a set of processes that provide services for a servlet, including instantiation and execution.

servlet runner The part of the servlet engine that invokes a servlet with a request object and a response object. *See* servlet engine.

session An object used by a servlet to track a user's interaction with a web application across multiple HTTP requests.

session bean An enterprise bean that is created by a client; usually exists only for the duration of a single client-server session. A session bean performs operations for the client, such as calculations or accessing other EJBs. While a session bean may be transactional, it is not recoverable if a system crash occurs. Session bean objects can be either stateless (not associated with a particular client) or stateful (associated with a particular client), that is, they can maintain conversational state across methods and transactions. *See also* stateful session bean, stateless session bean.

session cookie A cookie that is returned to the client containing a user session identifier. *See also* sticky cookie.

session timeout A specified duration after which the Sun ONE Application Server can invalidate a user session. *See* session.

single sign-on A situation where a user's authentication state can be shared across multiple J2EE applications in a single virtual server instance.

SMTP Simple Mail Transport Protocol

SNMP SNMP (Simple Network Management Protocol) is a protocol used to exchange data about network activity. With SNMP, data travels between a managed device and a network management station (NMS). A managed device is anything that runs SNMP: hosts, routers, your web server, and other servers on your network. The NMS is a machine used to remotely manage that network.

SOAP The Simple Object Access Protocol (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.

SQL Structured Query Language. A language commonly used in relational database applications. SQL2 and SQL3 designate versions of the language.

SSL Secure Sockets Layer. A protocol designed to provide secure communications on the Internet.

state **1.** The circumstances or condition of an entity at any given time. **2.** A distributed data storage mechanism which you can use to store the state of an application using the Sun ONE Application Server feature interface `IState2`. *See also* conversational state, persistent state.

stateful session bean A session bean that represents a session with a particular client and which automatically maintains state across multiple client-invoked methods.

stateless session bean A session bean that represents a stateless service. A stateless session bean is completely transient and encapsulates a temporary piece of business logic needed by a specific client for a limited time span.

sticky cookie A cookie that is returned to the client to force it to always connect to the same server process. *See also* session cookie.

stored procedure A block of statements written in SQL and stored in a database. You can use stored procedures to perform any type of database operation, such as modifying, inserting, or deleting records. The use of stored procedures improves database performance by reducing the amount of information that is sent over a network.

streaming A technique for managing how data is communicated through HTTP. When results are streamed, the first portion of the data is available for use immediately. When results are not streamed, the whole result must be received before any part of it can be used. Streaming provides a way to allow large amounts of data to be returned in a more efficient way, improving the perceived performance of the application.

Sun ONE Directory Server The Sun ONE version of Lightweight Directory Access Protocol (LDAP). Every instance of Sun ONE Application Server uses Sun ONE Directory Server to store shared server information, including information about users and groups. *See also* LDAP.

Sun ONE Message Queue The Sun ONE enterprise messaging system that implements the Java Message Service (JMS) open standard: it is a JMS provider.

system administrator The person who administers Sun ONE Application Server software and deploys Sun ONE Application Server applications.

table A named group of related data in rows and columns in a database.

thread An execution sequence inside a process. A process may allow many simultaneous threads, in which case it is multi-threaded. If a process executes each thread sequentially, it is single-threaded.

TLS Transport Layer Security. A protocol that provides encryption and certification at the transport layer, so that data can flow through a secure channel without requiring significant changes to the client and server applications.

topic An object created by an administrator to implement the publish/subscribe delivery model. A topic may be viewed as node in a content hierarchy that is responsible for gathering and distributing messages addressed to it. By using a topic as an intermediary, message publishers are kept separate from message subscribers.

transaction A set of database commands that succeed or fail as a group. All the commands involved must succeed for the entire transaction to succeed.

Transaction Attribute A transaction attribute controls the scope of a transaction.

transaction context A transaction's scope, either local or global. *See* local transaction, global transaction.

transaction isolation level Determines the extent to which concurrent transactions on a database are visible to one-another.

transaction manager An object that controls a global transaction, normally using the XA protocol. *See* global transaction.

Transaction Recovery Automatic or manual recovery of distributed transactions.

transience A protocol that releases a resource when it is not being used. Opposite of persistence.

trust database I security file that contains the public and private keys; also referred to as the key-pair file.

UDDI Universal Description, Discovery, and Integration. Provides worldwide registry of web services for discovery and integration.

URI Universal Resource Identifier. Describes a specific resource at a domain. Locally described as a subset of a base directory, so that `/ham/burger` is the base directory and a URI specifies `toppings/cheese.html`. A corresponding URL would be `http://domain:port/toppings/cheese.html`.

URL Uniform Resource Locator. An address that uniquely identifies an HTML page or other resource. A web browser uses URLs to specify which pages to display. A URL describes a transport protocol (for example, HTTP, FTP), a domain (for example, `www.my-domain.com`), and optionally a URI.

user A person who uses an application. Programmatically, a user consists of a user name, password, and set of attributes that enables an application to recognize a client. *See also* group, role.

user session A series of user application interactions that are tracked by the server. Sessions maintain user state, persistent objects, and identity authentication.

versioning *See* dynamic reloading.

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

A web application can be assigned to a specific virtual server. A server instance can have multiple virtual servers. *See also* server instance.

WAR file Web ARchive. A Java archive that contains a web module. WAR files have the.war extension.

web application A collection of servlets, JavaServer Pages, HTML documents, and other web resources, which might include image files, compressed archives, and other data. A web application may be packaged into an archive (a WAR file) or exist in an open directory structure.

Sun ONE Application Server also supports some non-Java web application technologies, such as SHTML and CGI.

web cache An Sun ONE Application Server feature that enables a servlet or JSP to cache its results for a specific duration in order to improve performance. Subsequent calls to that servlet or JSP within the duration are given the cached results so that the servlet or JSP does not have to execute again.

web connector plug-in An extension to a web server that enables it to communicate with the Sun ONE Application Server.

web container *See* container.

web module An individually deployed web application. *See* web application.

web server A host that stores and manages HTML pages and web applications, but not full J2EE applications. The web server responds to user requests from web browsers.

Web Server Plugin The web server plugin is an HTTP reverse proxy plugin that allows you to instruct a Sun One Web Server or Sun ONE Application Server to forward certain HTTP requests to another server.

web service A service offered via the web. A self-contained, self-describing, modular application that can accept a request from a system across the Internet or an intranet, process it, and return a response.

WSDL Web Service Description Language. An XML-based language used to define web services in a standardized way. It essentially describes three fundamental properties of a web service: definition of the web service, how to access that web service, and the location of that web service.

XA protocol A database industry standard protocol for distributed transactions.

XML Extensible Markup Language. A language that uses HTML-style tags to identify the kinds of information used in documents as well as to format documents.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X