Sun Java System Application Server Platform Edition 9 Upgrade and Migration Guide

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

Part No: 819–3663

061102@15490

Contents

Preface

This *Sun Java System Application Server Platform Edition 9 Upgrade and Migration Guide* describes how to upgrade and migrate Java™ applications from the Sun Java System Application Server 8.x to the Sun Java SystemApplication Server 9 product line. This guide also describes how to migrate Java applications from Sun ONE Application Server 6.x/7 (also known as iPlanet Application Server), Java 2 Enterprise Edition (J2EE™) Reference Implementation (RI) 1.3 Application Server, Sun Java System Application Server 8.x, WebLogic Application Server, WebSphere Application Server, JBoss, and so on to Application Server 9.

This guide also describes differences between adjacent product releases and configuration options that can result in incompatibility with the product specifications. Specifically, this *Sun Java System Application Server Platform Edition 9 Upgrade and Migration Guide* details Sun Java System Application Server 9 incompatibility with Sun Java System Application Server 8 2004Q2, Sun Java System Application Server 7 2004Q2 and incompatibilities with the J2EE 1.4 specification.

Who Should Use This Book

The intended audience for this guide is the system administrator, network administrator, application server administrator, and web developer who has an interest in migration issues.

This guide assumes you are familiar with the following topics:

- HTML
- Application Servers
- Client/Server programming model
- Internet and World Wide Web
- Windows 2000 and/or Solaris™ operating systems
- Java programming
- Java APIs as defined in specifications for EJBs, JavaServer Pages (JSP)
- Java Database Connectivity (JDBC)
- Structured database query languages such as SQL
- Relational database concepts
- Software development processes, including debugging and source code control

How This Book Is Organized

This guide is organized as follows:

Chapter 1 , discusses the incompatibilities between Application Server 9 and Application Server6.x/7.x/8.

Chapter 2, discusses the Java incompatibilities between Application Server 9 and Application Server.

Chapter 3describes how to upgrade an earlier installation of Application Server to Application Server 9.

Chapter 4discusses the need for migrating applications.

Chapter 5describes how to migrate EJB 1.1 to EJB 2.0/EJB3.0 specification.

Chapter 6, describes the considerations and strategies to migrate applications from other application servers to Sun Java System Application Server 9 .

Chapter 7 describes the migration of the main components of a typical Java EE application from Sun ONE Application Server 6.x to Sun Java System Application Server 9.

Chapter 8 lists the tools and resources that aid in automatic migration of applications.

Application Server Documentation Set

The Application Server documentation set describes deployment planning and system installation. The URL for stand-alone Application Server documentation is `http://docs.sun.com/app/docs/coll/1310.4`. For an introduction to Application Server, refer to the books in the order in which they are listed in the following table.

TABLE P–1 Books in the Application Server Documentation Set

Book Title	Description
Release Notes	Late-breaking information about the software and the documentation. Includes a comprehensive, table-based summary of the supported hardware, operating system, JDK, and JDBC/RDBMS.
Quick Start Guide	How to get started with the Application Server product.
Installation Guide	Installing the software and its components.
Developer's Guide	Creating and implementing Java Enterprise Edition 5 applications intended to run on the Application Server that follow the open Java standards model for Java EE components and APIs. Includes information about developer tools, security, debugging, deployment, and creating lifecycle modules.

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

Book Title	Description
Java EE 5 Tutorial	Using Java EE 5 platform technologies and APIs to develop Java EE applications.
Administration Guide	Configuring, managing, and deploying Application Server subsystems and components from the Administration Console.
Administration Reference	Editing the Application Server configuration file, `domain.xml`.
Upgrade and Migration Guide	Migrating your applications to the new Application Server programming model, specifically from Application Server 6.x, 7.x, and 8.x. This guide also describes differences between adjacent product releases and configuration options that can result in incompatibility with the product specifications.
Troubleshooting Guide	Solving Application Server problems.
Error Message Reference	Solving Application Server error messages.
Reference Manual	Utility commands available with the Application Server; written in man page style. Includes the `asadmin` command line interface.

Related Books

[Use this section to link to related or recommended documentation sets and third-party books.]

For other Sun Java System server documentation, go to the following:

- Message Queue documentation
- Directory Server documentation
- Web Server documentation

The URL for all documentation about Sun Java Enterprise System (Java ES) and its components is `http://docs.sun.com/prod/entsys.06q3`.

Default Paths and File Names

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

TABLE P–2 Default Paths and File Names

Placeholder	Description	Default Value
install-dir	Represents the base installation directory for Application Server.	Sun Java Enterprise System (Java ES) installations on the Solaris platform: `/opt/SUNWappserver/appserver` Java ES installations on the Linux platform: `/opt/sun/appserver/` Other Solaris and Linux installations, non-root user: *user's home directory*`/SUNWappserver` Other Solaris and Linux installations, root user: `/opt/SUNWappserver` Windows, all installations: *SystemDrive*`:\Sun\AppServer`
domain-root-dir	Represents the directory containing all domains.	Java ES installations on the Solaris platform: `/var/opt/SUNWappserver/domains/` Java ES installations on the Linux platform: `/var/opt/sun/appserver/domains/` All other installations: *install-dir*`/domains/`
domain-dir	Represents the directory for a domain. In configuration files, you might see *domain-dir* represented as follows: `${com.sun.aas.instanceRoot}`	*domain-root-dir*/*domain-dir*

Typographic Conventions

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

TABLE P–3 Typographic Conventions

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

`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`**
		`Password:`

AaBbCc123	A placeholder to be replaced with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized (note that some emphasized items appear bold online)	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file.

Symbol Conventions

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

TABLE P–4 Symbol Conventions

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

Accessing Sun Resources Online

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

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

- Downloads of Sun products
- Services and solutions
- Support (including patches and updates)
- Training
- Research
- Communities (for example, Sun Developer Network)

Third-Party Web Site References

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

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

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. To share your comments, go to `http://docs.sun.com` and click Send Comments. In the online form, provide the full document title and part number. The part number is a 7-digit or 9-digit number that can be found on the book's title page or in the document's URL. For example, the part number of this book is 819-3663.

CHAPTER 1

Application Server Compatibility Issues

Application Server 9 is upward binary compatible with Application Server 8.x and with Application Server 7/6.x except for the incompatibilities noted below. Java EE applications that run on versions 6.x, 7.x, and 8.x also work on version 9 except for the incompatibilities listed in this chapter.

The topics discussed in this chapter include incompatibilities in the following areas:

HTTP File Caching

HTTP file caching has been re-introduced in Application Server 9. This feature was present in Application Server 8.1 Enterprise Edition but was discontinued in Application Server 8.2.

Classpath Changes

In Application Server 7.x and 8.x, most of the Application Server core classes were loaded by the by the System Classloader, which also loaded Java Virtual Machine (JVM) classes. In Application Server 9, the System Classloader loads JVM classes but most of the Application Server core classes have been moved to the new Shared Chain Classloader.

In the Migration Tool, on the JVM Settings page, under the Configuration component, there is a new field called System Classpath. This contains the JVM classes. Classes cannot be deleted from this field, but they can be added. The Server Classpath field no longer shows the Application Server core classes, and its use is discouraged. Use Classpath suffix instead.

Web Server Features

As a result, the following web server-specific features are no longer supported in Application Server 9:

- `cgi-bin`, `shtml`
- Simple Network Management Protocol (SNMP) support
- Netscape API (NSAPI) plug-in APIs
- Native-content-handling features
- Web server tools (`flexanlg`, `htpasswd`)
- HTTP QoS
- Web server configuration files (`*.conf`, `*.acl`, `mime.types`)
- Web server-specific log rotation facility
- Watch dog process (`appserv-wdog`)

Security Realms

The package names of the security realm implementations have been renamed from `com.iplanet.ias.security.auth.realm` in Application Server 7 to `com.sun.enterprise.security.auth.realm` in Application Server 9. Custom realms written using the `com.iplanet.*` classes must be modified.

The `com.sun.enterprise.security.AuthenticationStatus` class has been removed.

The `com.sun.enterprise.security.auth.login.PasswordLoginModule` `authenticate` method implementation has changed as follows:

```
/**
    * Perform authentication decision.
    * <P> Note: AuthenticationStatus and AuthenticationStatusImpl
    * classes have been removed.
    * Method returns silently on success and returns a LoginException
    * on failure.
    *
    * @return void authenticate returns silently on
    *                successful authentication.
    * @throws LoginException on authentication failure.
    *
    */
abstract protected void authenticate()
    throws LoginException;
```

For more information, see:

http://developers.sun.com/prodtech/appserver/reference/techart/as8_authentication/index.html

Sun Deployment Descriptor: `sun-web.xml`

In Application Server 7, the default value for the optional attribute `delegate` was false. In Application Server 9, this attribute defaults to true. This change means that by default the Web application classloader first delegates to the parent classloader before attempting to load a class by itself. For details, see "Application Server 9 Options Contrary to Java EE 5 Specification Recommendations" on page 27.

The `encodeCookies` Property

URL encoding of cookies is performed, if the `encodeCookies` property of the `sun-web-app` element in the `sun-web.xml` file is set to true. In Application Server 7, the default value of the `encodeCookies` property was true. This property was not present in Application Server 8. In Application Server 9, the default value is false.

URL encoding of cookies is unnecessary. Setting this property to true is strongly discouraged. This property is provided only for those rare applications that depended on this behavior in Application Server 7.

CORBA Performance Option

In Application Server 7, users were able to specify the following system property to optionally turn on some Object Request Broker (ORB) performance optimization:

```
-Djavax.rmi.CORBA.UtilClass=com.iplanet.ias.util.orbutil.IasUtilDelegate
```

The ORB performance optimization is turned on by default in Application Server 9. If you are using the preceding system property reference, you must remove it to avoid interfering with the default optimization.

File Formats

In Application Server 9, `domain.xml` is the main server configuration file. In Application Server 7, the main server configuration file was `server.xml`. The DTD file of `domain.xml` is found in `lib/dtds/sun-domain_1_1.dtd`. The upgrade tool included in Application Server 9 can be used to migrate the `server.xml` from Application Server 7 to `domain.xml` for Application Server 9.

The `lib/dtds/sun-domain_1_1.dtd` file for Application Server 9 is fully backward compatible with the corresponding file for Application Server 8, `sun-domain_1_0.dtd`.

In general, the configuration file formats are *not* backward compatible. The following configuration files are *not* supported:

- `*.conf`
- `*.acl`
- `mime.types`
- `server.xml` (replaced with `domain.xml`)

In Application Server 9, the use of ZIP files in the following directories is deprecated.

- Common class loader directory: `domain-dir/lib`
- Extension directory: `domain-dir/lib/ext`
- Endorsed directory:`domain-dir/lib/endorsed`

Tools Interoperability

As a general rule, tools are not interoperable between Application Server 7 and Application Server 9. Users must upgrade their Application Server 7 tools to work with Application Server 9.

System Properties

The default security policy of Application Server 9 does not allow you to change some system properties. For example, in Application Server 7, the read/write permission for `java.util.PropertyPermission` was `"*", "read,write";`. In Application Server 9the read/write permission for `java.util.PropertyPermission` is `"*", "read";`. The administrator can change the security policy to grant write access to specific system properties.

Implicit URL Rewriting

Application Server 6.x supported implicit URL rewriting, in which the web connector plug-in parsed the HTML stream being sent to the browser and appended session IDs to attributes such as `href=` and `frame=`. In Application Server 7.x/ 8.x and Application Server 9, this feature is not available. You need to review your applications and use `encodeURL` and `encodeRedirectURL` on every URL that the applications present to clients (such as mobile phones) that do not support cookies.

Java SE 5 Changes

Application Server 9 no longer bundles JAXP classes. In an Application Server 8.2 installation, JAXP jar files resided in the <*install-directory*>`/lib/endorsed` directory. These classes are now part of Java SE 5 and if you have a standalone application, you need to ensure you are using Java SE 5

Primary Key Attribute Values

In Application Server 7, it was possible to change any field (in the Administration Console) or attribute (in the Command Line Interface (CLI)). In Application Server 9, a field or attribute that is the primary key of an item cannot be changed. However, an item can be deleted and then recreated with a new primary key value. In most cases, the primary key is a name, ID, reference, or JNDI name. The following table lists the primary keys that cannot be changed.

Note – In the `domain.xml` file, a field or attribute is called an *attribute*, and an item is called an *element*. For more information about `domain.xml`, see the Chapter 1, "The domain.xml File," in *Sun Java System Application Server Platform Edition 9 Administration Reference*.

TABLE 1–1 Primary Key Attributes

Item	Primary Key Field or Attribute
`admin-object-resource`	`jndi-name`
`alert-subscription`	`name`
`appclient-module`	`name`
`application-ref`	`ref`
`audit-module`	`name`
`auth-realm`	`name`
`cluster-ref`	`ref`
`cluster`	`name`

TABLE 1–1 Primary Key Attributes *(Continued)*

Item	Primary Key Field or Attribute
config	name
connector-connection-pool	name
connector-module	name
connector-resource	jndi-name
custom-resource	jndi-name
ejb-module	name
external-jndi-resource	jndi-name
http-listener	id
iiop-listener	id
j2ee-application	name
jacc-provider	name
jdbc-connection-pool	name
jdbc-resource	jndi-name
jms-host	name
jmx-connector	name
lb-config	name
lifecycle-module	name
mail-resource	jndi-name
message-security-config	auth-layer
node-agent	name
profiler	name
element-property	name
provider-config	provider-id
resource-adapter-config	resource-adapter-name
resource-ref	ref
security-map	name
server	name
server-ref	ref

TABLE 1–1 Primary Key Attributes *(Continued)*

Item	Primary Key Field or Attribute
`system-property`	`name`
`thread-pool`	`thread-pool-id`
`virtual-server`	`id`
`web-module`	`name`
`persistence-manager-factory-resource`	`jndi-name`

Command Line Interface: start-appserv and stop-appserv

The `start-appserv` and `stop-appserv` commands are deprecated. Use of these commands results in a warning. Use`asadmin start-domain` and `asadmin stop-domain` instead.

In Application Server 9, the Log Messages to Standard Error field has been removed from the Administration Console. The `log-to-console` attribute in the `domain.xml` file is deprecated and ignored. The `asadmin set` command has no effect on the `log-to-console` attribute. Use the `---verbose` option of the `asadmin start-domain` command to print messages to the window in which you executed the `asadmin start-domain` command. This option works only if you execute the `asadmin start-domain` command on the machine that has the domain you are starting.

Command Line Interface: asadmin

The following sections describe changes to the command line utility `asadmin`:

- "Subcommands" on page 19
- "Error Codes for Start and Stop Subcommands" on page 20
- "Options" on page 20
- "Dotted Names" on page 21
- "Tokens in Attribute Values" on page 23
- "Nulls in Attribute Values" on page 24

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

Subcommands

Subcommands are backward compatible except as noted below.

The `reconfig`subcommand is deprecated and ignored.

The following subcommands are no longer supported in Application Server 9. The software license key and web core were removed, and Application Server 9 no longer supports controlled functions from web server features.

- `install-license`
- `display-license`
- `create-http-qos`
- `delete-http-qos`
- `create-mime`
- `delete-mime`
- `list-mime`
- `create-authdb`
- `delete-authdb`
- `list-authdbs`

Error Codes for Start and Stop Subcommands

For Application Server 7, the error codes for the `start` and `stop` subcommands of the `asadmin` command were based on the desired end state. For example, for `asadmin start-domain`, if the domain was already running, the exit code was 0 (success). If domain startup failed, the exit code was 1 (error).

For Application Server 9, the exit codes are based on whether the commands execute as expected. For example, the `asadmin start-domain` command returns exit code 1 if the domain is already running or if domain startup fails. Similarly, `asadmin stop-domain` returns exit code 1 if the domain is already not running or cannot be stopped.

Options

Options in the following table are deprecated or no longer supported.

TABLE 1–2 Deprecated and Unsupported `asadmin` Options

Option	Deprecated or Unsupported in Subcommands
`--acceptlang`	Deprecated for the `create-virtual-server` subcommand.
`--acls`	Deprecated for the `create-virtual-server` subcommand.
`--adminpassword`	Deprecated for all relevant subcommands. Use `--passwordfile` instead.
`--blockingenabled`	Deprecated for the `create-http-listener` subcommand.
`--configfile`	Deprecated for the `create-virtual-server` subcommand.
`--defaultobj`	Deprecated for the `create-virtual-server` subcommand.

TABLE 1–2 Deprecated and Unsupported `asadmin` Options *(Continued)*

Option	Deprecated or Unsupported in Subcommands
`--domain`	Deprecated for the `stop-domain` subcommand.
`--family`	Deprecated for the `create-http-listener` subcommand.
`--instance`	Deprecated for all remote subcommands. Use `--target` instead.
`--mime`	Deprecated for the `create-virtual-server` subcommand.
`--optionsfile`	No longer supported for any commands.
`--password`	Deprecated for all remote subcommands. Use `--passwordfile` instead.
`--path`	Deprecated for the `create-domain` subcommand. Use `--domaindir` instead.
`--resourcetype`	Deprecated for all relevant subcommands. Use `--restype` instead.
`--storeurl`	No longer supported for any commands.
`--target`	Deprecated for all `jdbc-connection-pool`, `connector-connection-pool`, `connector-security-map`, and `resource-adapter-config` subcommands.
`--type`	Deprecated for all relevant subcommands.

Dotted Names

All dotted names used in versions 8.1 or 8.2 are supported in Application Server 9. The following use of dotted names in `asadmin` `get` and `set` subcommands are not backward compatible:

- The default server name is `server` instead of `server1`
- `.resource` becomes `domain.resources.resource`
- `.app-module` becomes `domain.applications.app-module`
- Attributes names format is different. For example,`poolResizeQuantity` is now `pool-resize-quantity`.
- Some aliases supported in Application Server 7 are not supported in Application Server 9

In Application Server 9, the `---passwordfile` option of the `asadmin` command does not read the `password.conf` file, and the upgrade tool does not upgrade this file. For information about creating a password file in Application Server 9, see the *Sun Java System Application Server Platform Edition 9 Administration Guide*.

This table displays a one-to-one mapping of the incompatibilities in dotted names between Application Server 7 and 9. The compatible dotted names are not listed in this table.

TABLE 1–3 Incompatible Dotted Names Between Versions

Application Server 7 Dotted Names	Application Server 9 Dotted Names
server_instance`.http-listener.`*listener_id**server_instance*`.http-service.http-listener.`*listener_id*	`.http-service.http-listener.`*listener_id* `.http-service.http-listener.`*listener_id*
server_instance`.orb`*server_instance*`.iiop-service`	`.iiop-service.iiop-service`
server_instance`.orblistener`*server_instance*`.iiop-listener`	`.iiop-service.iiop-listener.`*listener_id* `.iiop-service.iiop-listener.`*listener_id*
server_instance`.jdbc-resource.`*jndi_name*	`.resources.jdbc-resource.`*jndi_name* `domain.resources.jdbc-resource.`*jndi_name*
server_instance`.jdbc-connection-pool.`*pool_id*	`.resources.jdbc-connection-pool.`*pool_id*`domain.resources.jdbc-connection-pool.`*pool_id*
server_instance`.external-jndi-resource.`*jndi_name**server_instance*`.jndi-resource.`*jndi_name*	`.resources.external-jndi-resource.`*jndi_name*`domain.resources.external.jndi-resource.`*jndi_name*
server_instance`.custom-resource.`*jndi_name*	`.resources.custom-resource.`*jndi_name* `domain.resources.custom-resource.`*jndi_name*
server_instance`.web-container.logLevel` (see note below)	`.log-service.module-log-levels.web-container` `.log-service.module-log-levels.web-container`
server_instance`.web-container.monitoringEnabled` (see note below)	`.monitoring-service.module-monitoring-levels.web-container` `.monitoring-service.module-monitoring-levels.web-container`
server_instance`.j2ee-application.`*application_name**server_instance*`.application.`*application_name*	`.applications.j2ee-application.`*application_name* `domain.applications.j2ee-application.`*application_name*
server_instance`.ejb-module.`*ejb-module_name*	`.applications.ejb-module.`*ejb-module_name*`domain.applications.ejb-module.`*ejb-module_name*
server_instance`.web-module.`*web-module_name*	`.applications.web-module.`*web-module_name*`domain.applications.web-module.`*web-module_name*
server_instance`.connector-module.`*connector_module_name*	`.applications.connector-module.`*connector_module_name* `domain.applications.connector-module.`*connector_module_name*

TABLE 1–3 Incompatible Dotted Names Between Versions *(Continued)*

Application Server 7 Dotted Names	Application Server 9 Dotted Names
server_instance.lifecycle-module. *lifecycle_module_name*	.applications.lifecycle -module.*lifecycle_module_name* domain.application.lifecycle-module.*lifecycle_module_name*
server_instance.virtual-server-class	N/A*
server_instance.virtual-server.*virtual-server_id*	.http-service.virtual-server.virtual-server_id .http-service.virtual-server.virtual-server_id
server_instance.mime.*mime_id*	N/A*
server_instance.acl.*acl_id*	N/A*
server_instance.virtual-server .*virtual-server_id*.auth-db.*auth-db_id*	N/A*
server_instance.authrealm.*realm_idserver_instance*. security-service.authrealm.*realm_id*	.security-service.auth -realm.*realm_id*.security-service-auth-realm.*realm_id*
server_instance.persistence-manager-factory-resource.*jndi_nameserver_instance* .resources.persistence-manager-factory-resource.*jndi_name*	.resources.persistence-manager-factory-resource.*jndi_name*domain.resources. persistence-manager-factory-resource.*jndi_name*
server_instance.http-service.acl.*acl_id*	N/A*
server_instance.mail-resource.*jndi_name*	.resources.mail-resource .*jndi_name*domain.resources.mail -resource.*jndi_name*
server_instance.profiler	.java-config.profiler.java-config.profiler

* — These attribute names in Application Server 7 do not correspond directly with Application Server 9 dotted names.

Tokens in Attribute Values

The `asadmin get` command shows raw values in Application Server 9 instead of resolved values as in Application Server 8. These raw values may be tokens. For example, executing the following command:

```
asadmin get domain.log-root
```

The preceding command displays the following value:

```
${com.sun.aas.instanceRoot}/logs
```

Nulls in Attribute Values

In Application Server 8, attributes with no values contained null. This caused problems in attributes that specified paths. In Application Server 9, attributes with no values contain empty strings, as they did in Application Server 7.

CHAPTER 2

Java EE 5 Compatibility Issues

The following topics are covered in this chapter:

Binary Compatibility

In this Application Server 9 release, the included Java SDK is The Java Platform, Enterprise Edition version 5 SDK (Java EE 5). This version of the Java EE is upwards binary-compatible with J2EE SDK 1.3, 1.4, and 1.5, except for the incompatibilities listed below. This means that, except for the noted incompatibilities, applications built for version 1.4 run correctly in the Sun Java System Application Server 9 release.

Source Compatibility

Downward source compatibility is not supported. If source files use new Java EE APIs, they are not usable with an earlier version of the Java platform.

In general, the policy is as follows:

- Maintenance releases do not introduce any new APIs, so they maintain source compatibility with one another.

  http://java.sun.com/j2se/1.5/compatibility.html
  (`http://java.sun.com/j2se/1.4.2/compatibility.html`)

- Functionality releases and major releases maintain upwards but not downwards source compatibility.

Deprecated APIs are methods and classes that are supported only for backward compatibility, and the compiler generates a warning message whenever one of these is used, unless the `-nowarn` command-line option is used. It is recommended that programs be modified to eliminate the use of deprecated methods and classes, though there are no current plans to remove such methods and classes entirely from the system.

Incompatibilities in the Java EE 5 Platform (since the J2EE 1.4 release)

The Sun Java System Application Server 9 release is based on Java EE 5, Application Server 8 is based on J2EE 1.4, and Application Server 7 release is based on the J2EE 1.3.

Almost all existing programs should run on the Sun Java System Application Server 9 release without modification. However, there are some minor potential incompatibilities that involve rare circumstances and corner cases that we are documenting here for completeness.

- The `java.net.Proxy` class was added in 5.0, making two classes named Proxy — `java.lang.reflect.Proxy` and `java.net.Proxy`. The introduction of `java.net.Proxy` will cause a source-level incompatibility in existing code if a compilation unit has the following type-import-on-demand declarations:

  ```
  import java.lang.reflect.*;
  import java.net.*;
  ```
- Java Servlet Specification Version 2.5 ships with the Sun Java System Application Server 9 release, and can be downloaded from the following URL:

  http://java.sun.com/products/servlet/ (`http://java.sun.com/products/servlet/`)

JAXP and SAX Incompatibilities

Sun Java System Application Server 9 supports JAXP 1.3, which in turn supports SAX 2.0.2. In SAX 2.0.2, `DeclHandler.externalEntityDecl` requires the parser to return the absolute system identifier for consistency with `DTDHandler.unparsedEntityDecl`. This might cause some incompatibilities when migrating applications that use SAX 2.0.0.

To migrate an application that uses SAX 2.0.0 to SAX 2.0.2 without changing the previous behavior of `externalEntityDecl`, you can set the resolve-dtd-uris feature to false. For example:

```
SAXParserFactory spf = SAXParserFactory.newInstance();
spf.setFeature("http://xml.org/sax/features/resolve-dtd-uris",false);
```

Other incompatibilities between SAX 2.0.0 and SAX 2.0.2 are documented in the JAXP Compatibility Guide.

Application Server 9 Options Incompatible with Java EE 5 Specification Requirements

All portable Java EE programs run on the Application Server 9 without modification. However, as allowed by the Java EE compatibility requirements, it is possible to configure applications to use features of the Sun Java System Application Server 9 that are not compatible with the Java EE specification.

The pass-by-reference element in the `sun-ejb-jar.xml` file only applies to remote calls. As defined in the EJB 3.0 specification calls to local interfaces use pass-by-reference semantics.

If the pass-by-reference element is set to its default value of false, the parameter passing semantics for calls to remote interfaces comply with the EJB 3.0 specification, section 5.4. If set to true, remote calls involve pass-by-reference semantics instead of pass-by-value semantics, contrary to this specification.

Portable programs cannot assume that a copy of the object is made during such a call, and thus that it is safe to modify the original. Nor can they assume that a copy is not made, and thus that changes to the object are visible to both caller and callee. When this flag is set to true, parameters and return values are considered read-only. The behavior of a program that modifies such parameters or return values is undefined. For more information about the pass-by-reference element, see the *Sun Java System Application Server Platform Edition 9 Developer's Guide*.

Application Server 9 Options Contrary to Java EE 5 Specification Recommendations

If the `delegate` attribute in the class-loader element of the `sun-web.xml` file is set to its default value of true, classes and resources residing in container-wide library JAR files are loaded in preference to classes and resources packaged within the WAR file, contrary to what is recommended in the Servlet 2.5 specification. If set to false, the classloader delegation behavior complies with what is recommended in the Servlet 2.5 specification.

Do not package portable programs that use the delegate attribute with the value of true with any classes or interfaces that are a part of the J2EE specification. The behavior of a program that includes such classes or interfaces in its WAR file is undefined. For more information about the class-loader element, see the *Sun Java System Application Server Platform Edition 9 Developer's Guide*.

CHAPTER 3

Upgrading an Application Server Installation

You can upgrade to Application Server 9 Platform Edition from Application Server 8.x Platform Edition installation. Information that is transferred includes data about deployed applications, the file realm, security certificates, and other resource and server configuration settings. You can install your upgrade in a new location, or you can upgrade in place by overwriting your previous installation.

This chapter discusses the following topics:

Upgrade Overview

The following table shows supported Application Server upgrades, where PE indicates Platform Edition and EE indicates Enterprise Edition.

TABLE 3–1 Supported Upgrade Paths

Source Installation	9 Platform Edition
7.XPE	Not supported
7.XSE	Not applicable
7.XEE	Not applicable
8.0PE	Supported
8.0 EE	Not applicable
8.1 PE	Supported

TABLE 3–1 Supported Upgrade Paths *(Continued)*

Source Installation	9 Platform Edition
8.1 EE	Not applicable
8.2 PE	Supported
8.2 EE	Not applicable

Note – Before starting the upgrade process, make sure that both the source server (the server from which you are upgrading) and the target server (the server to which you are upgrading) are stopped.

The software provides two methods, a command-line utility (`asupgrade`) and an Upgrade Wizard, for completing the upgrade. If you issue the `asupgrade` command with no options, the Upgrade Wizard is displayed. If you use the`asupgrade` command in command-line mode and if you do not supply all of the required information, an interviewer requests information for the required options you omitted. The Upgrade Wizard automatically detects the version of the specified source server installation.

If a domain contains information about a deployed application and the installed application components do not agree with the configuration information, the configuration is migrated as is without any attempt to reconfigure the incorrect configurations.

During an upgrade, the configuration and deployed applications of a previous version of the Application Server are migrated; however, the runtime binaries of the server are not updated. Database migrations or conversions are also beyond the scope of this upgrade process.

Only those instances that do not use Web Server-specific features are upgraded seamlessly. Configuration files related to HTTP path, CGI bin, SHTML, and NSAPI plug-ins are not be upgraded.

Enterprise Application Archive (EAR) files and component archives (JAR, WAR, and RAR files) that are deployed in the Application Server 8.x environment do not require any modification to run on Application Server 9.

Applications and components that are deployed in the source server are deployed on the target server during the upgrade. Applications that do not deploy successfully on the target server must be migrated using the Migration Tool or `asmigrate` command, then deployed again manually.

If the upgrade includes clusters, specify one or more cluster files. Upon successful upgrade, an upgrade report is generated listing successfully migrated items along with a list of the items that could not be migrated.

Upgrading Through the Command Line

The upgrade utility is run from the command line using the following syntax:

```
asupgrade
[--console ]
[--version ]
[--help ]
[--source applicationserver_8.x_installation]
[--target applicationserver_9_installation]
--adminuser admin_user
[--adminpassword admin_password]
[--masterpassword changeit]
[--passwordfile path_to_password_file]
[--domain domain_name]
[--nsspwdfile NSS_password_filepath]
[--targetnsspwdfile target_NSS_password_filepath]
[--jkspwdfile JKS_password_filepath]
[--capwdfile CA_password_filepath]
]
```

The following table describes the command options in greater detail, including the short form, the long form, and a description.

TABLE 3–2 `asupgrade` Utility Command Options

Short Form	Long Form	Description
`-c`	`---console`	Launches the upgrade command line utility.
`-V`	`---version`	The version of the Upgrade tool.
`-h`	`---help`	Displays the arguments for launching the upgrade utility.
`-t`	`---target`	The installation directory for Application Server 8.2.
`-a`	`---adminuser`	The username of the administrator.
`-w`	`---adminpassword`	The password for the `adminuser`. Although this option can be used, the recommended way to transmit passwords is by using the `-passwordfile` option.
`-m`	`--masterpassword`	The master password that is created during installation. The default value is `changeit`. Although this option can be used, the recommended way to transmit passwords is by using the --passwordfile option. Note: This option is required only if your target server is Application Server 8.2 EE.

TABLE 3–2 `asupgrade` Utility Command Options *(Continued)*

Short Form	Long Form	Description

`-f`	`--passwordfile`	The path to the file that contains the `adminpassword` and `masterpassword`. Content of this file should be in the following format:
		`AS_ADMIN_ADMINPASSWORD=adminpassword`
		`AS_ADMIN_MASTERPASSWORD=masterpassword`

`-d`	`--domain`	The domain name for the migrated certificates.
`-n`	`--nsspwdfile`	The path to the NSS password file.
`-e`	`--targetnsspwdfile`	The path to the target NSS password file.
`-j`	`--jkspwdfile`	The path to the JKS password file.
`-p`	`--capwdfile`	The path to the CA certificate password file.

The following examples show how to use the `asupgrade` command-line utility to upgrade an existing application server installation to Application Server 9.

Example: Upgrading an Application Server 8.0 installation to Application Server 9 with Prompts for Certificate Migration.

This example shows how to upgrade an Application Server 8.0 installation to Application Server 9. You will be prompted to migrate certificates. If you reply no, then no certificates will be migrated.

```
asupgrade --adminuser admin --passwordfile password.txt
 --source /home/sunas8.1 --target /home/sjsas9
```

Example: Upgrading an Application Server 8.0 PE Installation with JKS and CA Certificates to Application Server 9 PE

This example shows how to upgrade a Application Server 8.0 PE installation to Sun Java System Application Server 9 PE. JKS and CA certificates will be migrated.

```
asupgrade --adminuser admin --passwordfile password.txt
--source /home/sjsas8.0
--target /home/sjsas9
--domain domain1
--jkspwdfile /home/sjsas8.0/jkspassword.txt
--capwdfile /home/sjsas9/capassword.txt
```

Upgrading Through the Wizard

The Upgrade wizard provides a GUI, however, using the wizard increases install time and space requirements. You can start the Upgrade wizard in GUI mode from the command line or from the desktop.

To start the wizard,

- On UNIX, change to the `<install_dir>/bin` directory and type `asupgrade`.

- On Windows, double-click the `asupgrade` icon in the `<install_dir>/bin` directory.

If the Upgrade checkbox was selected during the Application Server installation process, the Upgrade Wizard screen will automatically display after the installation completes.

From the Upgrade Wizard screen:

▼ To Use the Upgrade Wizard

1. **In the Source Installation Directory field, enter the location of the existing installation from which to import the configuration.**

   This installation can be Application Server 8.0, Application Server 8.1, or Application Server 8.2.

2. **In the Target Installation Directory field, enter the location of the Application Server installation to which to transfer the configuration.**

   If the upgrade wizard was started from the installation (the Upgrade from Previous Version checkbox was checked during the Application Server installation), the default value for this field will be the directory to which the Application Server software was just installed.

3. **Continue with Step 4 if security certificates need to be transferred. If the source installation has security certificates that must be transferred, check the Transfer Security Certificates checkbox, press the Next button, and the Transfer Security Certificates screen displays.**

4. **From the Transfer Security Certificates screen, press the Add Domain button to add domains with certificates to be transferred.**

   The Add Domain dialog displays.

5. **From the Add Domain dialog, select the domain name that contains the security certificates to migrate and enter the appropriate passwords.**

6. **Click the OK button when done.**

   The Transfer Security Certificates screen will be displayed again.

7. **Repeat steps 4 through 6 until all the domains that have certificates to be transferred have been added.**

8 **After all of the domains that contain certificates to be transferred have been added, press the Next button.**

9 **The Upgrade Results screen displays, showing the status of the upgrade operation in the Results field.**

10 **Click the Finish button to close the Upgrade Tool when the upgrade process is complete.**

Correcting Potential Upgrade Problems

This section addresses issues that could occur during an upgrade to Application Server 9 and explains how you can correct them.

To Transfer JVM Options

When you upgrade from a previous version of the application server, transfer of the previous configuration is required. Since the target configuration files may have new parameters and new preconfigured features, copying the old configuration files to the new server installation is not possible. The values of the old configurations must be transferred to the Application Server 9 configuration format.

The following JVM options are not transferred from the source to the target installation:

- `Dorg.xml.sax.driver`
- `Dcom.sun.jdo.api.persistence.model.multipleClassLoaders`
- `Djava.util.logging.manager`
- `Dcom.sun.aas.imqLib`
- `Dcom.sun.aas.imqBin`
- `Dcom.sun.aas.webServicesLib`
- `Dcom.sun.aas.configRoot 8. Xmx<...>m`

The options that are not transferred are listed down in the upgrade log. The user can manually change such attributes, in the configuration file, if needed.

▼ To Upgrade Additional HTTP Listeners Defined on the Source Server to the Target PE Server

If additional HTTP listeners have been defined in the PE source server, those listeners need to be added to the PE target server after the upgrade:

1 **Start the Admin Console.**

2 **Expand Configuration.**

3. **Expand HTTP Service.**
4. **Expand Virtual Servers.**
5. **Select <server>.**
6. **In the right hand pane, add the additional HTTP listener name to the HTTP Listeners field.**
7. **Click Save when done.**

CHAPTER 4

Understanding Migration

This chapter addresses the following topics:

- "Migration and Deployment" on page 37
- "Java EE 5 Component Standards" on page 39
- "Java EE 5 Application Components" on page 40

Migration and Deployment

This section describes the need to migrate Java EE applications and the particular files that must be migrated. Following successful migration, a Java EE application is redeployed to the Application Server.

Redeployment is also described in this section.

The following topics are addressed:

- "Why is Migration Necessary?" on page 37
- "What Needs to be Migrated?" on page 38
- "What is Deployment of Migrated Applications?" on page 39

Why is Migration Necessary?

Although Java EE specifications broadly cover requirements for applications, they are nonetheless evolving standards. They either do not cover some aspects of applications or leave implementation details to the application providers.

This leads to different implementations of the application servers, also well as difference in the deployment of Java EE components on application servers. The array of available configuration and deployment tools for use with any particular application server product also contributes to the product implementation differences.

The evolutionary nature of the specifications itself presents challenges to application providers. Each of the component APIs are also evolving. This leads to a varying degree of conformance by products. In particular, an emerging product, such as the Application Server, has to contend with differences in Java EE application components, modules, and files deployed on other established application server platforms. Such differences require mappings between earlier implementation details of the Java EE standard, such as file naming conventions, messaging syntax, and so forth.

Moreover, product providers usually bundle additional features and services with their products. These features are available as custom JSP tags or proprietary Java API libraries. Unfortunately, using these proprietary features renders these applications non-portable.

What Needs to be Migrated?

For migration purposes, the Java EE application consists of the following file categories:

- Deployment descriptors (XML files)
- JSP source files that contain Proprietary APIs
- Java source files that contain Proprietary APIs

Deployment descriptors (XML files)

Deployment is accomplished by specifying deployment descriptors (DDs) for standalone enterprise beans (EJB JAR files), front-end Web components (WAR files) and enterprise applications (EAR files). Deployment descriptors are used to resolve all external dependencies of the Java EE components/applications. The Java EE specification for DDs is common across all application server products. However, the specification leaves several deployment aspects of components pertaining to an application dependent on product-implementation.

JSP source files

Java EE specifies how to extend JSP by adding extra custom tags. Product vendors include some custom JSP extensions in their products, simplifying some tasks for developers. However, usage of these proprietary custom tags results in non-portability of JSP files. Additionally, JSP can invoke methods defined in other Java source files as well. The JSPs containing proprietary APIs needs to be rewritten before they can be migrated.

Java source files

The Java source files can be EJBs, servlets, or other helper classes. The EJBs and servlets can invoke standard Java EE services directly. They can also invoke methods defined in helper classes. Java source files are used to encode the business layer of applications, such as EJBs.Vendors bundle several services and proprietary Java API with their products. The use of proprietary Java APIs is a major source of non-portability in applications. Since Java EE is an evolving standard, different products can support different versions of Java EE component APIs. This is another aspect that migration addresses.

For information on how to migrate these components, see Chapter 6

What is Deployment of Migrated Applications?

Deployment refers to deploying a migrated application that was previously deployed on an earlier version of Sun's Application Server, or any third party application server platforms.

Deploying a migrated application is described in Chapter 1, "Assembling and Deploying Applications," in *Sun Java System Application Server Platform Edition 9 Application Deployment Guide*. However, when migration activities are performed with automated tools, such as the Migration Tool for Sun Java System Application Server 9 (for Java EE applications) or the Sun ONE Migration Toolbox (for Netscape Application Servers), there might be post-migration or pre-deployment tasks that are needed (and defined) prior to deploying the migrated application.

See Chapter 8 for more information about migration tools that are available.

Java EE 5 Component Standards

Application Server 9 is a Java EE 5 -compliant server based on the component standards developed by the Java community. Application Server8.0 and 8.1/8.2 versions are J2EE 1,4 compatible, Application Server 7 is a J2EE 1.3-compliant server, and Sun ONE Application Server 6.x is a J2EE 1.2-compliant server. Between the Java EE versions, there are considerable differences with the Java EE application component APIs.

The following table characterizes the differences between the component APIs used with the Java EE-compliant Application Server 9, Application Server 7, and Application Server 6.x.

TABLE 4–1 Application Server Version Comparison of APIs for Java EE Components

Component API	Application Server 6.x	Application Server 7	Application Server 8.x	Sun Java System Application Server 9
JDK	1.2.2	1.4	1.4	1.5
Servlet	2.2	2.3	2.4	2.5
JSP	1.1	1.2	2.0	2.1
JDBC	2.0	2.0	2.1, 3.0	4.0
EJB	1.1	2.0	2.0	3.0
JNDI	1.2	1.2	1.2.1	1.2.1
JMS	1.0	1.1	1.1	1.1
JTA	1.0	1.01	1.01	1.01
JAX-WS	-	-	-	2.0

Java EE 5 Application Components

Java EE 5 simplifies development of enterprise applications by basing them on standardized, modular components, providing a complete set of services to those components, and handling many details of application behavior automatically, without complex programming. Prominent Java EE components include:

- Client Application
- Web Application
- Enterprise Java Bean (EJB)
- Connector
- Enterprise Application Archive (EAR)
- Annotations

Java EE components are packaged separately and bundled into a Java EE application for deployment. Each component, its related files such as GIF and HTML files or server-side utility classes, and a deployment descriptor are assembled into a module and added to the Java EE application. A Java EE application is composed of one or more enterprise bean(s), Web, or application client component modules. The final enterprise solution can use one Java EE application or be made up of two or more Java EE applications, depending on design requirements.

A Java EE application and each of its modules has its own deployment descriptor. A deployment descriptor is an XML document with an .xml extension that describes a component's deployment settings.

A Java EE application with all of its modules is delivered in an Enterprise Archive (EAR) file. An EAR file is a standard Java Archive (JAR) file with an .ear extension. The EAR file contains EJB JAR files, application client JAR files and/or Web Archive (WAR) files.

The migration process is concerned with moving Java EE application components, modules, and files. For more information on migrating various Java EE components, refer to Chapter 6.

For more information on Java EE, see:

- Java EE 5 specification
- Java EE 5 website

CHAPTER 5

Migrating EJB

Although the EJB 1.1 and EJB 2.0 specification will continue to be supported in Sun Java System Application Server 9, the use of the EJB 3.0 architecture is recommended to leverage its enhanced capabilities.

To migrate a number of modifications are required, including within the source code of components.

You can use the Migration Tool to migrate from EJB 1.1 to EJB 2.0. The Migration Tool will only perform EJB 1.1 to EJB 2.0 conversions if you selected the "Migrate CMPs from EJB 1.1 to 2.0 specs." option. You should choose the this option only if all source files (.java files) and the deployment descriptors (ejb-jar.xml and sun-j2ee-ri.xml files) are available in the input source code. This migration requires and modifies all of these files to affect this migration effectively. If, after verification, the Migration Tool is satisfied that a CMP EJB 1.1 enterprise bean can be migrated to EJB 2.0, it will make the appropriate changes.

The modifications required to migrate from EJB 2.0 to EJB 3.0 are related to the differences between EJB 2.0 and EJB 2.0. These differences are described in the following topics.

- "Metadata Annotations" on page 41
- "EJB Query Language" on page 42
- "Local Interfaces" on page 42
- "EJB 3.0 Entity Model " on page 43
- "Migrating EJB Client Applications" on page 45
- "Migrating CMP Entity EJBs" on page 47
- "DTD Changes during EJB Migration" on page 51

Metadata Annotations

An annotation consists of the `@` sign preceding the annotation type, followed by a parenthesized list of element-value pairs. The EJB 3.0 Specification defines a variety of annotation types such as those that specify a bean's type (`@Stateless`, `@Stateful`, `@MessageDriven`, `@Entity`), whether a bean is remotely or locally accessible (`@Remote`, `@Local`), transaction attributes (`@TransactionAttribute`), and security and method permissions (`@MethodPermissions`, `@Unchecked`, `@SecurityRoles`).

Annotations for the EJB 3.0 annotation types generate interfaces required by the class as well as references to objects in the environment. In many cases, defaults can be used instead of explicit metadata annotation elements. In these cases, a developer doesn't have to completely specify a metadata annotation to obtain the same result as if the annotation was fully specified. For example, by default, an entity bean (annotated by `@Entity`) has a default entity type of `CMP`, indicating that it has container-managed persistence. These defaults can make annotating enterprise beans very simple. In fact, in many cases, defaults are assumed when an annotation is not specified. In those cases, the defaults represent the most common specifications. For example, container-managed transaction demarcation (where the container, as opposed to the bean, manages the commitment or rollback of a unit of work to a database) is assumed for an enterprise bean if no annotation is specified. These defaults illustrate the "coding by exception" approach that guides EJB 3.0 technology. The intent is to simplify things for developers by forcing them to code things only where defaults are not adequate.

EJB Query Language

EJB 3.0 has made enhancements to EJB QL to provide additional functionality. Addition of projection, explicit inner and outer join operations, bulk update and delete, subqueries, and group-by. Addition of a dynamic query capability and support for native SQL queries.

The EJB 2.0 specification introduces a query language called *EJB Query Language*, or *EJB QL* to correct many of these inconsistencies and shortcomings. EJB QL is based on SQL92. It defines query methods, in the form of both finder and select methods, specifically for entity beans with container-managed persistence. EJB QL's principal advantage over SQL is its portability across EJB containers and its ability to navigate entity bean relationships.

The EJB 1.1 specification left the manner and language for forming and expressing queries for finder methods to each individual application server. While many application server vendors let developers form queries using SQL, others use their own proprietary language specific to their particular application server product. This mixture of query implementations causes inconsistencies between application servers.

Local Interfaces

Under the EJB 3.0 API, the business interface of an enterprise bean is a plain Java interface, not an `EJBObject` or `EJBLocalObject` interface. However, usage of the earlier `EJBObject` and `EJBLocalObject`interface types continues to be supported under EJB 3.0.

With EJB 2.0, session beans and entity beans can expose their methods to clients through two types of interfaces: a *remote interface* and a *local interface*. The 2.0 remote interface is identical to the remote interface used in the 1.1 architecture, whereby, the bean inherits from RMI, exposes its methods across the network tier, and has the same capability to interact with distributed clients.

However, the local interfaces for session and entity beans provide support for lightweight access from EJBs that are local clients; that is, clients co-located in the same EJB container. The EJB 2.0

specification further requires that EJBs that use local interfaces be within the same application. That is, the deployment descriptors for an application's EJBs using local interfaces must be contained within one `ejb-jar` file.

In the EJB 1.1 architecture, session and entity beans have one type of interface, a remote interface, through which they can be accessed by clients and other application components. The remote interface is designed such that a bean instance has remote capabilities; the bean inherits from RMI and can interact with distributed clients across the network.

The local interface is a standard Java interface. It does not inherit from RMI. An enterprise bean uses the local interface to expose its methods to other beans that reside within the same container. By using a local interface, a bean may be more tightly coupled with its clients and may be directly accessed without the overhead of a remote method call.

In addition, local interfaces permit values to be passed between beans with pass by reference semantics. Because you are now passing a reference to an object, rather than the object itself, this reduces the overhead incurred when passing objects with large amounts of data, resulting in a performance gain.

EJB 3.0 Entity Model

To create the session bean, a developer only needs to code a bean class and annotate it with appropriate metadata. A session bean need not home interface, remote interface, or deployment descriptor to code. All that is required in addition to the bean class is a business interface, and that can be generated by default. The bean class is coded as a plain old Java object (POJO) rather than a class that implements an interface such as `javax.ejb.SessionBean`. Because the bean class does not implement an interface such as javax.ejb.SessionBean, a developer no longer has to implement methods such as `ejbRemove`, `ejbActivate`, or `ejbPassivate` in the bean class. However a developer can implement any or all of these callbacks if they are needed. If the bean class implements one of these callbacks, the EJB container calls it just as it does for EJB 2.1 technology.

The new Java Persistence API in EJB 3.0 defines a new single model for implementing persistence in the Java platform. In EJB 3.0 technology, an entity bean class is a concrete class. It's no longer an abstract class.

The EJB 2.0 specification uses the CMP model. It expanded the existing CMP to allow multiple entity beans to have relationships among themselves. This is referred to as *Container-Managed Relationships* (CMR). The container manages the relationships and the referential integrity of the relationships.According to the EJB 2.0 specification, the implementation class for an entity bean that uses CMP is now defined as an abstract class.

The EJB 1.1 specification presented a more limited CMP model. The EJB 1.1 architecture limited CMP to data access that is independent of the database or resource manager type. It allowed you to expose only an entity bean's instance state through its remote interface; there is no means to expose bean relationships. The EJB 1.1 version of CMP depends on mapping the instance variables of an entity bean class to the data items representing their state in the database or resource manager. The

CMP instance fields are specified in the deployment descriptor, and when the bean is deployed, the deployer uses tools to generate code that implements the mapping of the instance fields to the data items.

The following topics are discussed in this section:

- "Defining Persistent Fields" on page 44
- "Dependency Injection" on page 44
- "Message-Driven Beans" on page 44

Defining Persistent Fields

In EJB 3.0, `get` and `set` methods are concrete, not abstract. In addition, these methods can include logic, something that was not possible previously. This is useful for actions such as validating fields. Another improvement is that access to the persistence fields is not limited to the get and set methods. The persistence fields are also accessible through a bean class's business methods. One restriction however is that in EJB 3.0 technology, only methods within the class can access persistence fields -- in other words, you cannot expose the instance variables outside of the class.

The EJB 2.0 specification lets you designate an entity bean's instance variables as CMP fields or CMR fields. You define these fields in the deployment descriptor. CMP fields are marked with the element `cmp-field`, while container-managed relationship fields are marked with the element `cmr-field`.

Dependency Injection

Previous versions of the EJB architecture forced the developer into complying with the requirements of the EJB container in terms of providing classes and implementing interfaces. By comparison, In EJB 3.0, dependency injection reflects the fact that the bean tells the EJB container what it needs, and then container satisfies those needs.

Message-Driven Beans

EJB 3.0 enhances message-driven beans with support for interceptors.

Message-driven beans were a new feature introduced by the EJB 2.0 architecture. Message-driven beans are transaction-aware components that process asynchronous messages delivered through the Java Message Service (JMS).

Migrating EJB Client Applications

Existing EJB 2.1 and earlier applications are supported to run unchanged in EJB 3.0 containers. All EJB 3.0 implementations support EJB 1.1, EJB 2.0, and EJB 2.1 deployment descriptors for applications written to earlier versions.

- "Clients written to the new EJB 3.0 API" on page 45
- "Declaring EJBs in the JNDI Context" on page 45
- "Using EJB JNDI References" on page 46

Clients written to the new EJB 3.0 API

A client written to the EJB 3.0 API may be a client of a component written to the EJB 2.1 or earlier API. Such clients may access components written to the EJB 3.0 APIs and components written to the earlier EJB APIs within the same transaction. Such clients access components written to the earlier EJB APIs using the EJB 2.1 client view home and component interfaces. The `EJB` annotation may be used for the injection of home interfaces into components that are clients of beans written to the earlier EJB client view.

Declaring EJBs in the JNDI Context

In Application Server 9, EJBs are systematically mapped to the JNDI sub-context *ejb/*. If you attribute the JNDI name *Account* to an EJB, the Application Server 9 automatically creates the reference *ejb/Account* in the global JNDI context. The clients of this EJB therefore have to look up *ejb/Account* to retrieve the corresponding home interface.

Let us examine the code for a servlet method deployed in Sun ONE Application Server 6.x.

The servlet presented here calls on a stateful session bean, BankTeller, mapped to the root of the JNDI context. The method whose code you are considering is responsible for retrieving the home interface of the EJB, to enable a BankTeller object to be instantiated, and a remote interface for this object to be retrieved, so that you can make business method calls to this component.

```
/**
   * Look up the BankTellerHome interface using JNDI.
   */
private BankTellerHome lookupBankTellerHome(Context ctx)
      throws NamingException
{
    try
    {
      Object home = (BankTellerHome) ctx.lookup("ejb/BankTeller");
      return (BankTellerHome) PortableRemoteObject.narrow(home,
                  BankTellerHome.class);
    }
    catch (NamingException ne)
    {
      log("lookupBankTellerHome: unable to lookup BankTellerHome" +
          "with JNDI name 'BankTeller': " + ne.getMessage() );
      throw ne;
    }
}
```

As the code already uses `ejb/BankTeller` as an argument for the lookup, there is no need for modifying the code to be deployed on Application Server 9.

Using EJB JNDI References

This section summarizes the considerations when using EJB JNDI references. Where noted, the consideration details are specific to a particular source application server platform.

Placing EJB References in the JNDI Context

It is only necessary to modify the name of the EJB references in the JNDI context mentioned above (moving these references from the JNDI context root to the sub-context *ejb/*) when the EJBs are mapped to the root of the JNDI context in the existing WebLogic application.

If these EJBs are already mapped to the JNDI sub-context `ejb/` in the existing application, no modification is required.

However, when configuring the JNDI names of EJBs in the deployment descriptor within the Sun Java Studio IDE, it is important to avoid including the prefix `ejb/` in the JNDI name of an EJB. Remember that these EJB references are *automatically* placed in the JNDI `ejb/` sub-context with Application Server 9. So, if an EJB is given to the JNDI name *BankTeller* in its deployment descriptor, the reference to this EJB will be translated by Application Server 9 into `ejb/BankTeller`, and this is the JNDI name that client components of this EJB must use when carrying out a lookup.

Global JNDI context versus local JNDI context

Using the global JNDI context to obtain EJB references is a perfectly valid, feasible approach with Application Server 9. Nonetheless, it is preferable to stay as close as possible to the Java EE specification, and retrieve EJB references through the local JNDI context of EJB client applications. When using the local JNDI context, you must first declare EJB resource references in the deployment descriptor of the client part (`web.xml` for a Web application, `ejb-jar.xml` for an EJB component).

Migrating CMP Entity EJBs

With the introduction of EJB 3.0, you can use JDO (in addition to CMP 2.0), which is an architecture that provides a standard way to transparently persist plain Java objects.

EJB 2.x and EJB 3.0 uses CMP 2.0. In order to migrate a CMP 1.1 bean to CMP 2.0, we first need to verify if a particular bean can be migrated. The steps to perform this verification are as follows.

▼ To Verify if a Bean Can be Migrated

1. **From the `ejb-jar.xml` file, go to the `<cmp-fields>` names and check if the optional tag `<prim-key-field>` is present in the `ejb-jar.xml file` and has an indicated value. If it does, go to next step.**

   Look for the `<prim-key-class>` field name in the `ejb-jar.xml`, get the class name and get the `public instance variables` declared in the class. Now see if the signature (name and case) of these variables matches with the `<cmp-field>` names above. Segregate the ones that are found. In these segregated fields, check if some of them start with an upper case letter. If any of them do, then migration cannot be performed.

2. **Look into the bean class source code and obtain the java types of all the `<cmp-field>` variables.**

3. **Change all the `<cmp-field>` names to lowercase and construct accessors from them. For example if the original field name is `Name` and its java type is `String`, the accessor method signature will be:**

   ```
   Public void setName(String name)Public String getName()
   ```

4. **Compare these accessor method signatures with the method signatures in the bean class. If there is an exact match found, migration is not possible.**

5. **Get the custom finder methods signatures and their corresponding SQLs. Check if there is a ″Join′ or ″Outer join′ or an ″OrderBy′ in the SQL, if yes, we cannot migrate, as EJB QL does not support ″joins′, ″Outer join′ and ″OrderBy′.**

6. **Any CMP 1.1 finder, which used `java.util.Enumeration`, must now use `java.util.Collection`. Change your code to reflect this. CMP 2.0 finders cannot return `java.util.Enumeration`.**

   "Migrating the Bean Class" on page 47 explains how to perform the actual migration process.

Migrating the Bean Class

This section describes the steps required to migrate the bean class to Sun Java System Application Server 9.

▼ To Migrate the Bean Class

1. **Prepend the bean class declaration with the keyword `abstract`.**

   For example if the bean class declaration was:

   ```
   Public class CabinBean implements EntityBean // before modification
   ```

   Change it to:

   ```
   abstract Public class CabinBean implements EntityBean // after
   modification
   ```

2. **Prefix the accessors with the keyword `abstract`.**

3. **Insert all the accessors after modification into the source(.java) file of the bean class at class level.**

4. **Comment out all the `cmp` fields in the source file of the bean class.**

5. **Construct protected instance variable declarations from the `cmp-field` names in lowercase and insert them at the class level.**

6. **Read up all the `ejbCreate()` method bodies (there could be more than one `ejbCreate`).**

   Look for the pattern ”`<cmp-field>`=*some value or local variable*’, and replace it with the expression ”abstract mutator method name (*same value or local variable*)’.

   For example, if the `ejbCreate` body before migration is:

   ```
   public MyPK ejbCreate(int id, String name) {
      this.id = 10*id;
      Name = name;   //1
      return null;
   }
   ```

   Change it to:

   ```
   public MyPK ejbCreate(int id, String name) {
      setId(10*id);
      setName(name);   //1
      return null;
   }
   ```

   Note that the method signature of the abstract accessor in `//1` is as per the Camel Case convention mandated by the EJB 2.0 specification. Also, the keyword ”*this*’ may or may not be present in the original source, but it *must be removed* from the modified source file.

7 **Initialize all the protected variables declared in the `ejbPostCreate()` methods in step 5.**

The protected variables will be equal in number with the `ejbCreate()` methods. This initialization will be done by inserting the initialization code in the following manner:

```
protected String name;  //from step 5
protected int id;  //from step 5
public void ejbPostCreate(int id, String name) {
   name = getName();    /*abstract accessor*/ //inserted in this step
   id  = getId();        /*abstract accessor*/ //inserted in this step
}
```

8 **Inside the `ejbLoad` method, set the protected variables to the beans' database state.**

To do so, insert the following lines of code:

```
public void ejbLoad() {
   name = getName();    // inserted in this step
   id = getId();        // inserted in this step
   ...                  // existing code
}
```

9 **Similarly, update the beans' state inside `ejbStore()` so that its database state gets updated.**

But remember, you are not allowed to update the setters that correspond to the primary key outside the `ejbCreate()`, so do not include them inside this method. Insert the following lines of code:

```
public void ejbStore() {
    setName(name);       //inserted in this step
    setId(id);           //Do not insert this if it is a part
                           of the primary key
    ...                  //already present code
}
```

10 **Replace all occurrences of any `<cmp-field>` variable names with the equivalent protected variable name (as declared in step 5).**

If you do not migrate the bean, at the minimum you need to insert the `<cmp-version>1.`*x*`</cmp-version>` tag inside the `ejb-jar.xml` file at the appropriate place, so that the unmigrated bean still works on Sun Java System Application Server 9.

Migration of ejb-jar.xml

To migrate the file `ejb-jar.xml` to Sun Java System Application Server 9, perform the following steps:

▼ To Migrate the EJB Deployment Descriptor

To migrate the EJB deployment descriptor file, `ejb-jar.xml`, edit the file and make the following changes.

1 **Convert all `<cmp-fields>` to lowercase.**

2 **Insert the tag** `<abstract-schema-name>` **after the** `<reentrant>` **tag.**

The schema name will be the name of the bean as in the `< ejb-name>` tag, prefixed with "ias_".

3 **Insert the following tags after the** `<primkey-field>` **tag:**

```
<security-identity>
   <use-caller-identity/>
</security-identity>
```

4 **Use the SQL obtained above to construct the EJB QL from SQL.**

5 **Insert the** `<query>` **tag and all its nested child tags with all the required information just after the** `<security-identity>` **tag.**

Custom Finder Methods

The custom finder methods are the `findBy`... methods (other than the default `findByPrimaryKey` method), which can be defined in the home interface of an entity bean. Since the EJB 1.1 specification does not stipulate a standard for defining the logic of these finder methods, EJB server vendors are free to choose their implementations. As a result, the procedures used to define the methods vary considerably between the different implementations chosen by vendors.

Sun ONE Application Server 6.x uses standard SQL to specify the finder logic.

Information concerning the definition of this finder method is stored in the enterprise bean's persistence descriptor (`Account-ias-cmp.xml)` as follows:

```
<bean-property>
  <property>
    <name>findOrderedAccountsForCustomerSQL</name>
    <type>java.lang.String</type>
    <value>
       SELECT BRANCH_CODE,ACC_NO FROM ACCOUNT where CUST_NO = ?
    </value>
    <delimiter>,</delimiter>
  </property>
</bean-property>
<bean-property>
  <property>
    <name>findOrderedAccountsForCustomerParms</name>
    <type>java.lang.Vector</type>
    <value>CustNo</value>
    <delimiter>,</delimiter>
  </property>
</bean-property>
```

Each `findXXX` finder method therefore has two corresponding entries in the deployment descriptor (SQL code for the query, and the associated parameters).

In Sun Java System Application Server 9 the custom finder method logic is also declarative, but is based on the EJB query language EJB QL.

The EJB-QL language cannot be used on its own. It has to be specified inside the file `ejb-jar.xml`, in the `<ejb-ql>` tag. This tag is inside the `<query>` tag, which defines a query (finder or select method) inside an EJB. The EJB container can transform each query into the implementation of the finder or select method. Here's an example of an `<ejb-ql>` tag:

```
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>hotelEJB</ejb-name>
      ...
      <abstract-schema-name>TMBankSchemaName</abstract-schema-name>
      <cmp-field>
      ...
      <query>
        <query-method>
          <method-name>findByCity</method-name>
            <method-params>
              <method-param>java.lang.String</method-param>
            </method-params>
        </query-method>
        <ejb-ql>
          <![CDATA[SELECT OBJECT(t) FROM TMBankSchemaName
                                    AS t WHERE t.city = ?1]]>
        </ejb-ql>
      </query>
    </entity>
    ...
  </enterprise-beans> ...
</ejb-jar>
```

DTD Changes during EJB Migration

Migrating EJBs from Application Server 6.x to Application Server 9 is done without making any changes to the EJB code. However, the following DTD changes are required.

Session Beans

- The `<!DOCTYPE>` definition must be modified to point to the latest DTDs with J2EE standard DDs, such as `ejb-jar.xml`.
- Replace `ias-ejb-jar.xml` file with the modified version of this file, named `sun-ejb-jar.xml`, created manually according to the DDs. For more details, see the`http://ttp://wwws.sun.com/software/dtd/appserver/sun-ejb-jar_2_1-1.dtd`

- In the `sun-ejb-jar.xml` file, the JNDI name for all the EJBs must be added before "`ejb/`' in all the JNDI names. This is required because, in Application Server 6.5, the JNDI name of the EJB can only be `ejb/`<*ejb-name*> where <*ejb-name*> is the name of the EJB as declared inside the `ejb-jar.xml` file.

  In the Application Server, a new tag has been introduced in the `sun-ejb-jar.xml`. This is where the JNDI name of the EJB is declared.

Note – To avoid changing JNDI names throughout the application, declare the JNDI name of the EJB as `ejb/`<*ejb-name*> inside the <*jndi-name*> tag.

Entity Beans

- The `<!DOCTYPE>` definition must be modified to point to the latest DTDs containing J2EE standard DDs, such as `ejb-jar.xml`.
- Update the `<cmp-version>` tag with the value 1.1, for all CMPs in the `ejb-jar.xml` file.
- Replace all the <*ejb-name*>`-ias-cmp.xml` files with the manually created `sun-cmp-mappings.xml` file. For more information, see `http://wwws.sun.com/software/dtd/appserver/sun-cmp-mapping_1_2.dtd`
- Generate `dbschema` by using the `capture-schema` utility in the Application Server installation's bin directory and place it above `META-INF` folder for Entity beans.
- `Replace the ias-ejb-jar.xml` with the `sun-ejb.jar.xml` in Application Server.
- In Application Server 6.5, the finder's SQL was directly embedded into the <*ejb-name*>`-ias-cmp.xml`. In Application Server, mathematical expressions are used to declare the `<query-filter>` for the various finder methods.

Message Driven Beans

Application Server provides seamless Message Driven Support through the tight integration of Sun Java System Message Queue with the Application Server, providing a native, built-in JMS Service.

This installation provides Application Server with a JMS messaging system that supports any number of Application Server instances. Each server instance, by default, has an associated built-in JMS Service that supports all JMS clients running in the instance.

Both container-managed and bean-managed transactions, as defined in the Enterprise JavaBeans Specification 2.0 and 3.0 are supported.

Message Driven Bean support in iPlanet Application Server was restricted to developers, and used many of the older proprietary APIs. Messaging services were provided by iPlanet Message Queue for Java 2.0. An LDAP directory was also required under iPlanet Application Server to configure the `Queue Connection Factory` object.

The `QueueConnectionFactory`, and other elements required to configure Message Driven Beans in Application Server are now specified in the `ejb-jar.xml` file.

For more information on the changes to deployment descriptors, see “Migrating Deployment Descriptors” on page 56 For information on Message Driven Beans see “Using Message-Driven Beans” in *Sun Java System Application Server Platform Edition 9 Developer’s Guide*.

CHAPTER 6

Migrating from Other Application Servers to Sun Java System Application Server 9

This chapter describes the considerations and strategies that are needed when moving Java EE applications from Application Server 6.x,, 7.x, or from servers such as Weblogic, Websphere servers to the Application Server Platform Edition 9 product line.

The sections that follow describe issues that arise while migrating the main components of a typical Java EE application from other application servers to Application Server Platform Edition 9.

This chapter contains the following sections:

Several migration issues described in this chapter are based on an actual migration that was performed for a Java EE application called *iBank*, a simulated online banking service, from Application Server 6.x to Sun Java System Application Server 9. This application reflects all aspects of a traditional Java EE application.

The following areas of the Java EE specification are covered by the iBank application:

- Servlets, especially with redirection to JSP pages (model-view-controller architecture)
- JSP pages, especially with static and dynamic inclusion of pages
- JSP custom tag libraries
- Creation and management of HTTP sessions
- Database access through the JDBC API
- Enterprise JavaBeans: Stateful and Stateless session beans, CMP and BMP entity beans.
- Assembly and deployment in line with the standard packaging methods of the J2EE application

Migrating Deployment Descriptors

There are two types of deployment descriptors, namely, Standard Deployment Descriptors and Runtime Deployment Descriptors. Standard deployment descriptors are portable across Java EE platform versions and vendors and does not require any modifications. Currently, there are exceptions due to standards interpretation. The following table lists such deployment descriptors.

Source Deployment Descriptor in Sun Application 6.x /7.x	Source Deployment Descriptor in Websphere 4.0 /5.x	Source Deployment Descriptor in WebLogic 4.0 /5.x	Target Deployment Descriptor in 9
`ias-ejb-jar.xml`	`ibm-ejb-jar-bnd.xmi`, `ibm-ejb-jar-ext.xmi`	`weblogic-ejb-jar.xml`	`sun-ejb-jar.xml`
`<bean-name>-ias-cmp.xml`	`ibm-ejb-jar-ext.xmi`	`weblogic-cmp-rdbms-jar.xml`	`sun-cmp-mappings.xml`
`ias-web.xml`	`ibm-web-bnd.xmi`, `ibm-web-ext.xmi`	`weblogic.xml`	`sun-web.xml`

The Java EE standard deployment descriptors `ejb-jar.xml`, `web.xml` and `application.xml` are not modified and therefore need not be changed.

Runtime deployment descriptors are vendor and product specific and are not portable across application servers due to difference in their format. Hence, deployment descriptors require migration. You can use the Migration Tool for Application Server 9 to migrate these deplyment descriptors.

A majority of the information required for creating `sun-ejb-jar.xml` and `sun-web.xml` comes from `ias-ejb-jar.xml` and `ias-web.xml` respectively. However, there is some information that is required and extracted from the home interface (java file) of the CMP entity bean, in case the `sun-ejb-jar.xml` being migrated declares one. This is required to build the `<query-filter>` construct inside the `sun-ejb-jar.xml`, which requires information from inside the home interface of that CMP entity bean. If the source file is not present during the migration time, the `<query-filter>` construct is created, but with missing information (which manifests itself in the form of REPLACE ME phrases in the migrated `sun-ejb-jar.xml`).

Additionally, if the `ias-ejb-jar.xml` contains a `<message-driven>` element, then information from inside this element is picked up and used to fill up information inside both `ejb-jar.xml` and `sun-ejb-jar.xml`. Also, inside the `<message-driven>` element of `ias-ejb-jar.xml`, there is an element `<destination-name>`, which holds the JNDI name of the topic or queue to which the MDB listens. In Application Server 6.5, the naming convention for this jndi name is `cn=<SOME_NAME>.` Since a JMS Topic or Queue with this name is not deployable on Application Server, the application server changes this to `<SOME_NAME>`, and inserts this information in the `sun-ejb-jar.xml`. This change must be reflected for all valid input files, namely, all `.java, .jsp` and `.xml` files. Hence, this JNDI name change is propagated across the application, and if some source files that contain reference to this jndi-name are unavailable, the administrator must make the changes manually so that the application becomes deployable.

Migrating Web Applications

Application Server 6.x support servlets (Servlet API 2.2), and JSPs (JSP 1.1). Sun Java System Application Server 9 supports Servlet API 2.4 and JSP 2.0.

Within these environments it is essential to group the different components of an application (servlets, JSP and HTML pages and other resources) together within an archive file (Java EE-standard Web application module) deploying it on the application server.

According to the Java EE specification, a Web application is an archive file (WAR file) with the following structure:

- A root directory containing the HTML pages, JSP, images and other static resources of the application.
- A `META-INF/` directory containing the archive manifest file `MANIFEST.MF` containing the version information for the SDK used and, optionally, a list of the files contained in the archive.
- A `WEB-INF/` directory containing the application deployment descriptor (`web.xml` file) and all the Java classes and libraries used by the application, organized as follows:
  - A `classes/` sub-directory containing the tree-structure of the compiled classes of the application (servlets, auxiliary classes), organized into packages
    - A `lib/` directory containing any Java libraries (JAR files) used by the application

Migrating Java Server Pages and JSP Custom Tag Libraries

Application Server 6.x complies with the JSP 1.1 specification and Application Server 8.1 complies with the JSP 2.0 specification.

JSP 2.0 specification contains many new features, as well as updates to the JSP 1.1 specification.

These changes are enhancements and are not required to migrate to JSP pages from JSP 1.1 to 2.0.

The implementation of JSP custom tag libraries in Application Server 6.x complies with the J2EE specification. Consequently, migrating JSP custom tag libraries to the Application Server Platform Edition 9does not pose any particular problem, nor require any modifications.

Migrating Servlets

Application Server 6.x supports the Servlet 2.2 API. Sun Java System Application Server 9 supports the Servlet 2.4 API.

Servlet API 2.4 leaves the core of servlets relatively untouched. Most changes are concerned with adding new features outside the core.

The most significant features are:

- Servlets now require JDK 1.2 or later
- Filter mechanisms have been created
- Application lifecycle events have been added
- Internationalization support has been added
- Error and security attributes have been expanded
- `HttpUtils` class has been deprecated
- Several DTD behaviors have been expanded and clarified

These changes are enhancements and are not required to be made when migrating servlets from Servlet API 2.2 to 2.4.

However, if the servlets in the application use JNDI to access resources in the Java EE application (such as data sources or EJBs), some modifications might be needed in the source files or in the deployment descriptor.

These modifications are explained in detail in the following sections:

- "Obtaining a Data Source from the JNDI Context" on page 58
- "Declaring EJBs in the JNDI Context" on page 58

One last scenario might require modifications to the servlet code. Naming conflicts can occur with Application Server 6.x if a JSP page has the same name as an existing Java class. In this case, the conflict must be resolved by modifying the name of the JSP page in question. This in turn can mean editing the code of the servlets that call this JSP page. This issue is resolved in Application Server as it uses a new class loader hierarchy. In the new version of the application server, for a given application, one class loader loads all EJB modules and another class loader loads web module. As these two loaders do not talk with each other, there is no naming conflict.

Obtaining a Data Source from the JNDI Context

To obtain a reference to a data source bound to the JNDI context, look up the data source's JNDI name from the initial context object. The object retrieved in this way is then be *cast* as a DataSource type object:

```
ds = (DataSource)ctx.lookup(JndiDataSourceName);
```

For detailed information, refer to section "Migrating JDBC Code."

Declaring EJBs in the JNDI Context

Please refer to section "Clients written to the new EJB 3.0 API" on page 45 in Chapter 5."

Potential Servlets and JSP Migration Problems

The actual migration of the components of a Servlet / JSP application from Application Server 6.x to Application Server 9does not require any modifications to the component code.

If the Web application is using a server resource, a DataSource for example, the Application Server requires that this resource to be declared inside the web.xml file and, correspondingly, inside the sun-web.xml file. To declare a DataSource called jdbc/iBank, the <resource-ref> tag in the web.xml file is as follows:

```
<resource-ref>
   <res-ref-name>jdbc/iBank</res-ref-name>
   <res-type>javax.sql.XADataSource</res-type>
   <res-auth>Container</res-auth>
   <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

The corresponding declaration inside the sun-web.xml file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<! DOCTYPE FIX ME: need confirmation on the DTD to be used for this file
<sun-web-app>
   <resource-ref>
      <res-ref-name>jdbc/iBank</res-ref-name>
      <jndi-name>jdbc/iBank</jndi-name>
   </resource-ref>
 </sun-web-app>
```

Migrating Web Application Modules

Migrating applications from Application Server 6.x to Sun Java System Application Server 9 does not require any changes to the Java code or Java Server Pages. However, you must change the following files:

- web.xml
- ias-web.xml

The Application Server adheres to J2EE 1.4 standards, according to which, the web.xml file inside a WAR file must comply with the revised DTD at http://java.sun.com/dtd/web-app_2_3.dtd. This DTD is a superset of the previous versions' DTD, hence only the <! DOCTYPE definition needs to be changed inside the web.xml file, which is to be migrated. The modified <! DOCTYPE declaration looks like:

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD
Web Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
```

In Application Server Platform Edition 9, the name of this file is changed to sun-web.xml.

This XML file must declare the Application Server-specific properties and resources that are required by the Web application.

See "Potential Servlets and JSP Migration Problems" on page 58 for information about important inclusions to this file.

If the ias-web.xml of the Application Server 6.5 application is present and does declare Application Server 6.5 specific properties, then this file needs to be migrated to Application Server standards. The DTD file name has to be changed to sun-web.xml. For more details, see URL http://wwws.sun.com/software/dtd/appserver/sun-web-app_2_4-1.dtd

Once you have made these changes to the web.xml and ias-web.xml files, the Web application (WAR file) can be deployed from the Application Server's deploytool GUI interface or from the command line utility asadmin. The deployment command must specific the type of application as web.

Invoke the asadmin command line utility by running asadmin.bat file or the asadmin.sh script in the Application Server's bin directory.

The command at the asadmin prompt is:

```
asadmin deploy -u username -w password
-H hostname
-p adminport
--type web
[--contextroot contextroot]
[--force=true]
[--name component-name]
[--upload=true] filepath
```

Migrating Enterprise EJB Modules

Application Server 6.x supports EJB 1.1, and the Application Server 8.x supports EJB 2.0. Application Server 9 supports EJB 3.0. Therefore, both can support:

- Stateful or stateless session beans
- Entity beans with bean-managed persistence (BMP), or container-managed persistence (CMP)

EJB 2.0, however, introduces a new type of enterprise bean, called a message-driven bean (MDB).

J2EE 1.4 specification dictates that the different components of an EJB must be grouped together in a JAR file with the following structure:

- META-INF/ directory with an XML deployment descriptor named ejb-jar.xml
- The .class files corresponding to the home interface, remote interface, the implementation class, and the auxiliary classes of the bean with their package

Application Server 6.x uses this archive structure. However, the EJB 1.1 specification leaves each EJB container vendor to implement certain aspects as they see fit:

- Database persistence of CMP EJBs (particularly the configuration of mapping between the bean's CMP fields and columns in a database table).
- Implementation of the custom finder method logic for CMP beans.

- Application Server 6.x/7.x and Application Server 9do not handle migrations in the same way, which means that some XML files must be modified:
- The `<!DOCTYPE` definition must be modified to point to the latest DTD URL (in the case of J2EE standard DDs, like `ejb-jar.xml`).
- Replace the `ias-ejb-jar.xml` file with the modified version of this file (for example, file `sun-ejb-jar.xml`, which is created manually according to the DTDs). For more information, see `http://wwws.sun.com/software/dtd/appserver/sun-ejb-jar_2_1-1.dtd`
- Replace all the <*ejb-name*>`-ias-cmp.xml` files with one `sun-cmp-mappings.xml` file, which is created manually. For more information, see `http://wwws.sun.com/software/dtd/appserver/sun-cmp-mapping_1_2.dtd`
- Optionally, for CMP entity beans, use the capture-schema utility in the Application Server's `bin` directory to generate the dbschema. Then place it above the `META-INF` directory for the entity beans.

EJB Migration

As mentioned in Chapter 4, while Application Server 6.x supports the EJB 1.1 specification and Application Server 8 supports EJB 2.0, Application Server 9 supports the EJB 3.0 specification.

Although the EJB 1.1 and 2.0 specification continues to be supported in the Application Server, the use of the EJB 3.0 architecture is recommended to leverage its enhanced capabilities.

For detailed information on migrating from EJB 1.1 to 2.0 to EJB 3.0, please refer to Chapter 5

Migrating Enterprise Applications

According to the Java EE specifications, an enterprise application is an EAR file, which must have the following structure:

- A `META-INF/` directory containing the XML deployment descriptor of the J2EE application called `application.xml`
- The JAR and WAR archive files for the EJB modules and Web module of the enterprise application, respectively

In the application deployment descriptor, the modules that make up the enterprise application and the Web application's context root are defined.

Application server 6.x and the Application Server 9support the J2EE model wherein applications are packaged in the form of an enterprise archive (EAR) file (extension `.ear`). The application is further subdivided into a collection of J2EE modules, packaged into Java archives (JAR files, which have a `.jar` file extension) and EJBs and Web archives (WAR files, which have a `.war` file extension) for servlets and JSPs.

It is essential to follow the steps listed here before deploying an enterprise application:

▼ To Build an EAR File

1 Package EJBs in one or more EJB modules.

2 Package the components of the Web application in a Web module.

3 Assemble the EJB modules and Web modules in an enterprise application module.

4 Define the name of the enterprise application's root context, which will determine the URL for accessing the application.

The Application Server uses a newer class loader hierarchy than Application Server 6.x does. In the new scheme, for a given application, one class loader loads all EJB modules and another class loader loads Web modules. These two are related in a parent child hierarchy where the JAR module class loader is the parent module of the WAR module class loader. All classes loaded by the JAR class loader are available/accessible to the WAR module but the reverse is not true. If a certain class is required by the JAR file as well as the WAR file, then the class file must be packaged inside the JAR module only. If this guideline is not followed it can lead to class conflicts.

Application Root Context and Access URL

There is a major "difference between Application Server 6.x and the Application Server, concerning the applications access URL (root context of the application's Web module. If `AppName` is the name of the root context of an application deployed on a server called `hostname`, the access URL for this application will differ depending on the application server used:

- With Application Server 6.x, which is always used jointly with a Web front-end, the access URL for the application takes the following form (assuming the Web server is configured on the standard HTTP port, 80):

  `http://`*<hostname>*`/NASApp/AppName/`

- With the Application Server, the URL takes the form:

  `http://`*<hostname>*`:`*<portnumber>*`/AppName/`

The TCP port used as default by Application Server is port 8080.

Although the difference in access URLs between Application Server 6.x and the Application Server might appear minor, it can be problematic when migrating applications that make use of absolute URL references. In such cases, it is necessary to edit the code to update any absolute URL references so that they are no longer prefixed with the specific marker used by the Web Server plug-in for Application Server 6.x.

Applications With Form-based Authentication

Applications developed on Application Server 6.5 that use form-based authentication can pass the request parameters to the Authentication Form or the Login page. The Login page could be customized to display the authentication parameters based on the input parameters.

For example:

```
http://gatekeeper.uk.sun.com:8690/NASApp/test/secured/page.jsp?arg1=test&arg2=m
```

Application Server 9 does not support the passing of request parameters while displaying the Login page. The applications that uses form-based authentication, which passes the request parameters can not be migrated to Application Server 9. Porting such applications to Application Server 9 requires significant changes in the code. Instead, you can store the request parameter information in the session, which can be retrieved while displaying the Login page.

The following code example demonstrates the workaround:

Before changing the code in 6.5:

```
---------index-65.jsp -----------
<%@page contentType="text/html"%>
<html>
<head><title>JSP Page</title></head>
<body>
go to the <a href="secured/page.htm">secured a rea</a>
</body>
</html>
----------login-65.jsp--------------
<%@page contentType="text/html"%>
<html>
<head> </head>
<body>
<!-- Print login form -->
<h3>Parameters</h3><br>
out.println("arg1 is " + request.getParameter("arg1"));
out.println("arg2 is " + request.getParameter("arg2"));
</body>
</html>
```

After changing the code in Application Server 9:

```
---------index-81.jsp -----------
<%@page contentType="text/html"%>
<html>
<head><title>JSP Page</title></head>
<body>
<%session.setAttribute("arg1","test"); %>
```

```
<%session.setAttribute("arg2","me"); %>
go to the <a href="secured/page.htm">secured area</a>
</body>
</html>
```

The `index-81.jsp` shows how you can store the request parameters in a session.

```
----------login-81.jsp--------------
<%@page contentType="text/html"%>
<html>
<head> </head>
<body>
<!-- Print login form -->
<h3>Parameters</h3><br>
<!--retrieving the parameters from the session -->
out.println("arg1 is"+(String)session.getAttribute("arg1"));
out.println("arg2 is” + (String)session.getAttribute("arg2"));
</body>
</html>
```

Migrating Proprietary Extensions

A number of classes proprietary to the Application Server 6.x environment might have been used in applications. Some of the proprietary packages used by Application Server 6.x are listed below:

- `com.iplanet.server.servlet.extension`
- `com.kivasoft.dlm`
- `com.iplanetiplanet.server.jdbc`
- `com.kivasoft.util`
- `com.netscape.server.servlet.extension`
- `com.kivasoft`
- `com.netscape.server`

These APIs are not supported in the Application Server. Applications using any classes belonging to the above package must be rewritten to use standard J2EE APIs. Applications using custom JSP tags and UIF framework also need to be rewritten to use standard J2EE APIs.

For a sample migration walkthrough using the iBank application, see Chapter 7.

Migrating UIF

The Application Server does not support the use of Unified Integration Framework (UIF) API for applications. Instead, it supports the use of J2EE Connector Architecture (JCA) for integrating the applications. However, the applications developed in Application Server 6.5 use the UIF. In order to deploy such applications to the Application Server, migrate the UIF to the J2EE Connector Architecture. This section discusses the prerequisites and steps to migrate the applications using UIF to Application Server.

Before migrating the applications, ensure that the UIF is installed on Application Server 6.5. To check for the installation, follow either of the following approaches:

Checking in the Registry Files

UIF is installed as a set of application server extensions. They are registered in the application server registry during the installation. Search for the following strings in the registry to check whether UIF is installed.

Extension Name Set:

- Extension DataObjectExt-cDataObject
- Extension RepositoryExt-cLDAPRepository
- Extension MetadataService-cMetadataService
- Extension RepoValidator-cRepoValidator
- Extension BSPRuntime-cBSPRuntime
- Extension BSPErrorLogExt-cErrorLogMgr
- Extension BSPUserMap-cBSPUserMap

The registry file on Solaris Operating Environment can be found at the following location:

AS_HOME`/AS/registry/reg.dat`

Checking for UIF Binaries in Installation Directories

UIF installers copy specific binary files in to the application server installation. Successfully finding the files listed below, indicates that UIF is installed.

The location of the following files on Solaris and Windows is:

AS_HOME`/AS/APPS/bin`

List of files to be searched on Solaris:

- `libcBSPRlop.so`
- `libcBSPRuntime.so`
- `libcBSPUserMap.so`
- `libcDataObject.so`

- `libcErrorLogMgr.so`
- `libcLDAPRepository.so`
- `libcMetadataService.so`
- `libcRepoValidator.so`
- `libjx2cBSPRuntime.so`
- `libjx2cDataObject.so`
- `libjx2cLDAPRepository.so`
- `libjx2cMetadataService.so`

List of files to be searched on Windows:

- `cBSPRlop.dll`
- `cBSPRuntime.dll`
- `cBSPUserMap.dll`
- `cDataObject.dll`
- `ErrorLogMgr.dll`
- `cLDAPRepository.dll`
- `cMetadataService.dll`
- `cRepoValidator.dll`
- `jx2cBSPRuntime.dll`
- `jx2cDataObject.dll`
- `jx2cLDAPRepository.dll`
- `jx2cMetadataService.dll`

Before migrating the UIF to Application Server, ensure that the UIF API is being used in the applications. To verify its usage:

- Check for the usage of `netscape.bsp` package name in the Java sources
- Check for the usage of `access_cBSPRuntime.getcBSPRuntime` method in the sources. You must call this method to acquire the UIF runtime.

Contact appserver-migration@sun.com for information about UIF migration to the Application Server.

Migrating Rich Clients

This section describes the steps for migrating RMI/IIOP and ACC clients developed in Planet Application Server 6.x to the Application Server.

Authenticating a Client in Application Server 6.x

Application Server 6.x provides a client-side callback mechanism that enables applications to collect authentication data from the user, such as the username and the password.The authentication data collected by the iPlanet CORBA infrastructure is propagated to the application server via IIOP.

If ORBIX 2000 is the ORB used for RMI/IIOP, portable interceptors implement security by providing hooks, or interception points, which define stages within the request and reply sequence.

Authenticating a Client in Sun Java System Application Server 9

The authentication is done based on JAAS (Java Authorization and Authentication System API). If a client does not provide a `CallbackHandler`, then the default `CallbackHandler`, called the `LoginModule`, is used by the ACC to obtain the authentication data.

For detailed instructions on using JAAS for authentication, see Chapter 8, "Configuring Security," in *Sun Java System Application Server Platform Edition 9 Administration Guide*.

Using ACC in Application Server 6.x and Sun Java System Application Server 9

In Application Server 6.x, no separate appclient script is provided. You are required to place the `iasacc.jar` file in the classpath instead of the `iascleint.jar` file. The only benefit of using the ACC for packaging application clients in 6.x is that the JNDI names specified in the client application are indirectly mapped to the absolute JNDI names of the EJBs.

In case of Application Server 6.x applications, a stand-alone client uses the absolute name of the EJB in the JNDI lookup. That is, outside an ACC, the following approach is used to lookup the JNDI:

```
initial.lookup(“ejb/ejb-name”);
initial.lookup(“ejb/module-name/ejb-name”);
```

If your application was developed using Application Server 6.5 SP3, you would have used the prefix "`java:comp/env/ejb/`" when performing lookups via absolute references.

```
initial.lookup("java:comp/env/ejb/ejb-name");
```

In Sun Java System Application Server 9, the JNDI lookup is done on the `jndi-name` of the EJB. The absolute name of the ejb must not be used. Also, the prefix, `java:comp/env/ejb` is not supported in Sun Java System Application Server 9. Replace the `iasclient.jar`, `iasacc.jar`, or `javax.jar` JAR files in the classpath with `appserv-ext.jar`.

If your application provides load balancing capabilities, in Sun Java System Application Server 9, load balancing capabilities are supported only in the form of S1ASCTXFactory as the context factory on the client side and then specifying the alternate hosts and ports in the cluster by setting the `com.sun.appserv.iiop.loadbalancingpolicy` system property as follows:

```
com.sun.appserv.iiop.loadbalancingpolicy=
roundrobin,host1:port1,host2:port2,...,
```

This property provides the administrator with a list of host:port combinations to round robin the ORBs. These host names can also map to multiple IP addresses. If this property is used along with `org.omg.CORBA.ORBInitialHost` and `org.omg.CORBA.ORBInitialPort` as system properties, the round robin algorithm will round robin across all the values provided. If, however, a host name and port number are provided in your code, in the environment object, that value overrides any other system property settings.

The Provider URL to which the client is connected in Application Server 6.5 is the IIOP host and port of the CORBA Executive Engine (CXS Engine). In case of Sun Java System Application Server 9, the client needs to specify the IIOP listener Host and Port number of the instance. No separate CXS engine exists in Sun Java System Application Server 9.

The default IIOP port is 3700 in Sun Java System Application Server 9; the actual value of the IIOP Port can be found in the `domain.xml` configuration file.

CHAPTER 7

Migrating a Sample Application - an Overview

This chapter describes the process for migrating the main components of a typical Java EE application from Sun ONE Application Server 6.x to Sun Java System Application Server 9. This chapter highlights some of the problems posed during the migration of each type of component and suggests practical solutions to overcome such problems.

For this migration process, the Java EE application presented is called *iBank* and is based on the actual migration of the iBank application from Sun ONE Application Server 6.x to Application Server 9. iBank simulates an online banking service and covers all of the aspects traditionally associated with a Java EE application.

The major points of the Java EE specification covered by the iBank application are:

- Servlets, especially with redirection to JSP pages (model-view-controller architecture)
- JSP pages, especially with static and dynamic inclusion of pages
- JSP custom tag libraries
- Creation and management of HTTP sessions
- Database access through the JDBC API
- Enterprise JavaBeans: Stateful and Stateless session beans, CMP and BMP entity beans
- Assembly and deployment in line with the standard packaging methods of the J2EE application

Preparing for Migrating the iBank Application

Before starting the migration process, it in important to understand the differences in the deployment descriptors. For detailed information, see "Migrating Deployment Descriptors" on page 56.

Choosing the Target

To start, choose Sun Java System Application Server 9 as the target migration server. Install the server in the migration environment. For step-by-step instructions on how to install the software, see the *Sun Java System Application Server 9 Installation Guide*.

If you are using Migration Tool for Sun Java System Application Server 9 to migrate the components, install the tool. The Migration Tool can be downloaded from the following location:

http://java.sun.com/j2ee/tools/migration (`http://java.sun.com/j2ee/tools/migration`)

For information on how to use the Migration Tool for Sun Java System Application Server 9.1, see the Migration Tool online help. The iBank application is bundled with the tool.

Identifying the Components of the iBank Application

The iBank application has the following directory structure:

`iBank/docroot/session/entity/misc`

- `/docroot` contains HTML, JSP's and Image files in its root. It also contains the source files for servlets and EJBs in the sub-folder `WEB-INF\\classes` following the package structure `com.sun.bank.*`. A war file is generated using this directory.
- `/session` contains the source code for the session beans following the package structure `com.sun.bank.ejb.session`. This directory forms the EJB module for the session beans.
- `/entity` contains the entity beans following the package structure `com.sun.bank.ejb.entity`. This directory would form the EJB module for entity beans.
- `/misc` contain the sql scripts for the database setup.

Manual Steps in the iBank Application Migration

Most of the migration is done by the Migration Tool. There are some aspects of migration that must be done manually. These steps are documented in the Migration Tool's user's guide and the documentation for the iBank sample application.

Configuring Database Connectivity

In order to deploy an application to the target server, you must add a connection pool, add a JDBC resource and a persistence manager.

This section discusses the following topics:

- "Adding a Connection Pool" on page 71

- "Adding a JDBC Resource" on page 72
- "Adding a Persistence Manager" on page 72

Note – Before you begin these steps, make sure that the domain to which the application will be deployed is in the running state. These instructions assume that the application will be deployed to the default domain, `domain1`.

Use the `asadmin` utility in the Application Server bin directory to perform these tasks.

Adding a Connection Pool

A JDBC connection pool is a group of reusable connections for a particular database. Because creating each new physical connection is time consuming, the server maintains a pool of available connections to increase performance. When an application requests a connection, it obtains one from the pool. When an application closes a connection, the connection is returned to the pool.

Use the `asadmin` `create-jdbc-connection-pool` command to add a connection pool to the server. The syntax of the command is given below.

```
asadmin create-jdbc-connection-pool
      --user  admin_user
      --password admin_password
      --host localhost
      --port portno
      --datasourceclassname dsclassname
      --property User=ibank_user:Password=ibank_user:URL_PROP=db_url TMB
```

where, *dsclassname* is:

- `oracle.jdbc.pool.OracleDataSource` for Oracle
- `com.pointbase.jdbc.jdbcDataSource` for PointBase

URL_PROP is:

- `url` for Oracle
- `DatabaseName` for PointBase

db_url is:

- `jdbc:oracle:thin:@`*ORACLE_HOST*`:1521:SID` for Oracle, where *ORACLE_HOST* is the machine name/IP address on which the database is installed, and SID is the System ID of the Oracle database.
- `jdbc:pointbase:server://`*POINTBASE_HOST:9092*`/migration-samples` for Pointbase, where *POINTBASE_HOST* is the machine name/IP address on which the database is installed. This will be localhost in most cases.

Adding a JDBC Resource

A JDBC resource (data source) provides applications with a means of connecting to a database. Before creating a JDBC resource, you must first create a JDBC connection pool.

Use the `asadmin` `create-jdbc-resource` command to add resource.

```
asadmin create-jdbc-resource --user admin_user --password admin_password --host
localhost --port portno --connectionpoolid TMB jdbc/IBank
```

Adding a Persistence Manager

A persistence manager is required for backward compatibility. To run on version 7 of the Application Server, a persistent manager resource was required for applications with container-managed persistent beans (a type of EJB component).

Use the `asadmin` `create-persistence-resource` command.

```
asadmin create-persistence-resource --user admin_user --password admin_password
--host localhost --port portno --connectionpoolid TMB --factoryclass
com.sun.jdo.spi.persistence.support.sqlstore.impl.
PersistenceManagerFactoryImpljdo/pmf
```

Assembling Application for Deployment

Application Server primarily supports the J2EE model wherein applications are packaged in the form of an enterprise archive (EAR) file (extension .ear). The application is further subdivided into a collection of J2EE modules, packaged into Java archives (JAR, extension .jar) for EJBs and web archives (WAR, extension .war) for servlets and JSPs.

All the JSPs and Servlets must be packaged into WAR file, all EJBs into the JAR file and finally the WAR and the JAR file together with the deployment descriptors in to the EAR file. This EAR file is a deployable component.

Using the `asadmin` Utility to Deploy the iBank Application on Application Server

The last step is to deploy the application on Sun Java System Application Server 9. The process for deploying an application is described below:

The Sun Java System Application Server 9 asadmin command includes a help section on deployment that is accessible from the Help menu.

The command line utility `asadmin` can be invoked by executing `asadmin.bat` file in Windows and `asadmin` file in Solaris Operating Environment that is stored in Application Server's installation's `bin` directory.

At `asadmin` prompt, the command for deployment looks like this:

`asadmin> deploy -u` *username* `-w` *password* `-H` *hostname* `-p` *adminport* *absolute_path_to_application*

After restarting the Application Server, open a browser and go to the following URL to test the application:

`http://`<*machine_name*>`:`<*port_number*>`/ibank`

When prompted, enter one of the available user names and passwords. The main menu page of the iBank application displays.

CHAPTER 8

Migration Tools and Resources

This chapter describes migration tools that help automate the migration process from earlier versions of Sun ONE Application Server, Sun Java System Application Server 7, Netscape Application Server (Kiva), NetDynamics Application Server, and competitive application servers to Sun Java System Application Server 9.

Migration Tool for Sun Java System Application Server 9

The Migration Tool for Sun Java System Application Server 9 (hereafter called Migration Tool) migrates Java EE applications from other server platforms to Sun Java System Application Server 9.

The following source platforms are supported for Sun Java System Application Server 9:

- Sun ONE Application Server 6.*x*
- Sun Java System Application Server 7.x
- WebLogic Application Server (WLS) 5.1, 6.0, 6.1, 8.1
- WebSphere Application Server (WAS) 4.0, 5.x
- Sun ONE Web Server 6.0
- JBoss Application Server 3.0
- TomCat Web Server 4.1
- J2EE Reference Implementation 1.3/1.4

Migration Tool automates the migration of Java EE applications to Sun Java System Application Server 9, without much modification to the source code.

The key features of the tool are:

- Migration of application server-specific deployment descriptors
- Runtime support for selected custom JavaServer Pages (JSP) tags and proprietary APIs
- Conversion of selected configuration parameters with equivalent functionality in Application Server
- Automatic generation of Ant based scripts for building and deploying the migrated application to the target server, Application Server

- Generation of comprehensive migration reports after achieving migration

Download the Migration Tool from the following location:

http://java.sun.com/j2ee/tools/migration/index.html (`http://java.sun.com/j2ee/tools/migration/index.html`)

For detailed information on how to install and use the tool, see the Migration Tool online help.

The Migration Tool specifications and migration process change from time to time, so the sample migration using the tool is not included in this guide. The migration process of a sample application is discussed in the documentation for this tool.

Redeploying Migrated Applications

Most of the applications that are migrated automatically through the use of the available migration tools utilize the standard deployment tasks described in the *Sun Java System Application Server 9 Administration Guide*.

In some cases, the automatic migration is not able to migrate particular methods or syntaxes from the source application. When this occurs, a message displays describing the steps needed to complete the migration. Once these steps are completed, the administrator is able to deploy the application in the standard manner.

J2EE Application Verification Kit

The Java Application Verification Kit (AVK) for the Enterprise helps build and test applications to ensure that they are using the J2EE APIs correctly and to migrate to other J2EE compatible application servers using specific guidelines and rules.

Download the Java Application Verification Kit (AVK) from the following location:

http://java.sun.com/j2ee/verified/ (`http://java.sun.com/j2ee/verified/`)

Index