Sun Java™ System

Application Server Platform Edition 8.1 Upgrade and Migration Guide

2005Q1

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

Part No: 819-0083

Contents

Preface

This *Upgrade and Migration Guide* describes how Java™ 2 Platform, Enterprise Edition (J2EE™ platform) applications are migrated from the Sun ONE Application Server 6.x/7 (also known as iPlanet Application Server), J2EE Reference Implementation (RI) 1.3 Application Server, Sun Java System Application Server 7 to the Sun Java System Application Server Platform Edition 8.1 product line.

This guide also describes differences between adjacent product releases and configuration options that can result in incompatibility with the product specifications. Specifically, this *Upgrade and Migration Guide* details Sun Java System Application Server 8.1 2005Q1 incompatibility with Sun Java System Application Server 8 2004Q2, Sun Java System Application Server 7 2004Q2, and the Java™ 2 Platform, Enterprise Edition (J2EE™ platform), version 1.4 specification.

This preface contains information about the following topics:

- Who Should Use This Guide
- Before You Read This Book
- How This Guide Is Organized
- Conventions Used in This Book
- Related Documentation
- Contacting Sun Technical Support
- Related Third-Party Web Site References
- Sun Welcomes Your Comments

Who Should Use This Guide

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, Java Server 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

Before You Read This Book

Application Server is a component of Sun Java™ Enterprise System, a software infrastructure that supports enterprise applications distributed across a network or Internet environment. You should be familiar with the documentation provided with Sun Java Enterprise System, which can be accessed online at `http://docs.sun.com/app/docs/prod/entsys.05q1#hic.`

How This Guide Is Organized

This guide is organized as follows:

Table 1 How This Guide is Organized

Chapter	Description
Chapter 1, "Application Server Compatibility Issues"	Discusses the incompatibilities between Application Server 8.1 and Application Server 7/8.
Chapter 2, "J2EE 1.4 Compatibility Issues"	Discusses the J2EE incompatibilities between Application Server 8.1 and Application Server 7/8.
Chapter 3, "Upgrading an Application Server Installation"	Describes the process to upgrade an earlier installation of application server to Application Server 8.1.
Chapter 4, "Understanding Migration"	Discusses the need to migrate applications.
Chapter 5, "Migrating from EJB 1.1 to EJB 2.0"	Describes the process to migrate EJB 1.1 to EJB2.0 specification.
Chapter 6, "Migrating from Application Server 6.x/7.x to Application Server 8.1"	Describes the considerations and strategies to migrate applications from earlier releases of Sun's application servers to Sun Java System Application Server 8 2004Q4.
Chapter 7, "Migrating a Sample Application - an Overview"	Describes the steps to migrate a sample application from Sun ONE Application Server 6.x to Sun Java System Application Server 2004Q4.
Chapter 8, "Migration Tools and Resources"	Lists the tools and resources that aid in automatic migration of applications.
Appendix A, "iBank Application Specification"	Describes the specification of the sample application- iBank.

Conventions Used in This Book

The tables in this section describe the conventions used in this book.

Typographic Conventions

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

Table 2 Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123` (Monospace)	API and language elements, HTML tags, web site URLs, command names, file names, directory path names, onscreen computer output, sample code.	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`

`AaBbCc123` (Monospace bold)	What you type, when contrasted with onscreen computer output.	`%` **`su`**
		`Password:`

AaBbCc123 (Italic)	Book titles, new terms, words to be emphasized. A placeholder in a command or path name to be replaced with a real name or value.	Read Chapter 6 in the *User's Guide*. These are called *class* options. Do *not* save the file. The file is located in the *install-dir*/`bin` directory.

Symbols

The following table describes the symbol conventions used in this book.

Table 3 Symbol Conventions

Symbol	Description	Example	Meaning
`[ ]`	Contains optional 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.
-	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.

Table 3 Symbol Conventions *(Continued)*

Symbol	Description	Example	Meaning
>	Indicates menu item selection in a graphical user interface.	File > New > Templates	From the File menu, choose New. From the New submenu, choose Templates.

Default Paths and File Names

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

Table 4 Default Paths and File Names

Term	Description

install_dir	By default, the Application Server installation directory is located here:
	• Sun Java Enterprise System installations on the Solaris™ platform:
	`/opt/SUNWappserver/appserver`
	• Sun Java Enterprise System 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	By default, the directory containing all domains is located here:
	• Sun Java Enterprise System installations on the Solaris platform:
	`/var/opt/SUNWappserver/domains/`
	• Sun Java Enterprise System installations on the Linux platform:
	`/var/opt/sun/appserver/domains/`
	• All other installations:
	install_dir`/domains/`

Table 4 Default Paths and File Names

Term	Description
domain_dir	By default, each domain directory is located here: *domain_root_dir*/*domain_dir* In configuration files, you might see *domain_dir* represented as follows: `${com.sun.aas.instanceRoot}`

Shell Prompts

The following table describes the shell prompts used in this book.

Table 5 Shell Prompts

Shell	Prompt
C shell on UNIX or Linux	*machine-name*%
C shell superuser on UNIX or Linux	*machine-name*#
Bourne shell and Korn shell on UNIX or Linux	`$`
Bourne shell and Korn shell superuser on UNIX or Linux	`#`
Windows command line	`C:\`

Related Documentation

The `http://docs.sun.com`SM web site enables you to access Sun technical documentation online. You can browse the archive or search for a specific book title or subject.

Books in This Documentation Set

The Sun Java System Application Server manuals are available as online files in Portable Document Format (PDF) and Hypertext Markup Language (HTML).

The following table summarizes the books included in the Application Server core documentation set.

Table 6 Books in This 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 Sun Java System Application Server product.
Installation Guide	Installing the Sun Java System Application Server software and its components.
Developer's Guide	Creating and implementing Java™ 2 Platform, Enterprise Edition (J2EE™ platform) applications intended to run on the Sun Java System Application Server that follow the open Java standards model for J2EE components and APIs. Includes general information about developer tools, security, assembly, deployment, debugging, and creating lifecycle modules.
J2EE 1.4 Tutorial	Using J2EE 1.4 platform technologies and APIs to develop J2EE applications and deploying the applications on the Sun Java System Application Server.
Administration Guide	Configuring, managing, and deploying the Sun Java System Application Server subsystems and components from the Administration Console.
Administration Reference	Editing the Sun Java System Application Server configuration file, `domain.xml`.
Upgrade and Migration Guide	Migrating your applications to the new Sun Java System Application Server programming model, specifically from Application Server 6.x and 7. This guide also describes differences between adjacent product releases and configuration options that can result in incompatibility with the product specifications.
Troubleshooting Guide	Solving Sun Java System Application Server problems.
Error Message Reference	Solving Sun Java System Application Server error messages.
Reference Manual	Utility commands available with the Sun Java System Application Server; written in manpage style. Includes the `asadmin` command line interface.

Other Server Documentation

For other server documentation, go to the following:

- Message Queue documentation
  `http://docs.sun.com/db?p=prod/s1.s1msgqu`
- Directory Server documentation
  `http://docs.sun.com/coll/DirectoryServer_04q2`
- Web Server documentation
  `http://docs.sun.com/coll/S1_websvr61_en`

Accessing Sun Resources Online

For product downloads, professional services, patches and support, and additional developer information, go to the following:

- Download Center
  http://wwws.sun.com/software/download/
- Professional Services
  http://www.sun.com/service/sunps/sunone/index.html
- Sun Enterprise Services, Solaris Patches, and Support
  http://sunsolve.sun.com/
- Developer Information
  http://developers.sun.com/prodtech/index.html

Contacting Sun Technical Support

If you have technical questions about this product that are not answered in the product documentation, go to http://www.sun.com/service/contacting.

Related Third-Party Web Site References

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 document title and part number. The part number is a seven-digit or nine-digit number that can be found on the title page of the book or at the top of the document. For example, the title of this book is Sun Java System Application Server Platform Edition 8.1 *2005Q1 Upgrade and Migration Guide,* and the part number is *819-0083*.

Chapter 1

Application Server Compatibility Issues

The Sun Java System Application Server 8.1 2005Q1 (Application Server 8.1) is upward binary-compatible with Sun Java System Application Server 8 2004Q2 (Application Server 8) and with Sun Java System Application Server 7 2004Q2 (Application Server 7) except for the incompatibilities noted below. J2EE applications that run on versions 7 and 8 also work on version 8.1 except for the incompatibilities noted below.

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

- Web Server Features
- Security Realms
- Sun Deployment Descriptor: sun-web.xml
- encodeCookies Property
- CORBA Performance Option
- File Formats
- Certificate Database
- Tools Interoperability
- Primary Key Attribute Values
- Command Line Interface: start-appserv and stop-appserv
- Command Line Interface: asadmin

Web Server Features

Application Server 8.1 replaces the Web server shipped with Application Server 7 with a Java-based web container. As a result, the following web server-specific features are no longer supported in version 8.1:

- `cgi-bin`, `shtml`
- SNMP support
- NSAPI plugin 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 8.1. 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 8.1, 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 8.1 Options Contrary to J2EE 1.4 Specification Recommendations” on page 32.

encodeCookies Property

The `encodeCookies` property of the `sun-web-app` element in the `sun-web.xml` file performs URL encoding of cookies if set to true. If set to false, no encoding of cookies is performed. 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 8.1, the default value is false.

In general, 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. This property might be removed in a future release.

CORBA Performance Option

In Application Server 7, users were able to specify the following system property to optionally turn on some 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 8.1. If you are using the system property reference above, you must remove it to avoid interfering with the default optimization.

File Formats

In Application Server 8.1, `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 8.1 can be used to migrate the `server.xml` from Application Server 7 to `domain.xml` for Application Server 8.1.

The `lib/dtds/sun-domain_1_1.dtd`. file for Application Server 8.1 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`)

Certificate Database

Application Server 8.1 uses Java Keystore (JKS) as the keystore format. The NSS format used in Application Server 7 is not supported. The upgrade tool included in the product can be used to migrate existing NSS keystores to JKS keystores.

Tools Interoperability

As a general rule, tools are not interoperable between Application Server 7 and 8.1. Users must upgrade their Application Server 7 tools to work with Application Server 8.1.

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). In Application Server 8.1, 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 *Sun Java System Application Server 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`
`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`

Table 1-1 Primary Key Attributes

Item	Primary Key Field or Attribute
`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`
`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 8.1, 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 `start-domain`. This only works if you execute `start-domain` on the machine on which the domain you are starting is installed.

Command Line Interface: asadmin

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

- Subcommands
- Error Codes for Start and Stop Subcommands
- Options
- Dotted Names
- Tokens in Attribute Values
- Nulls in Attribute Values

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

Subcommands

Subcommands are backward compatible except as noted below.

The following sub command is deprecated and ignored:

- `reconfig`

Application Server 8.1 can only create one instance, so these subcommands are not supported.

- `create-instance`
- `delete-instance`
- `list-instances`
- `start-instance`

- `stop-instance`
- `show-instance-status`
- `restart-instance`

The following subcommands are no longer supported in Application Server 8.1, because the software license key and web core were removed, and because controlled functions from web server features are no longer supported:

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

Error Codes for Start and Stop Subcommands

For Application Server 7, the exit codes returned by 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 8.1, 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.
`--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

The following use of dotted names in `asadmin` `get` and `set` subcommands are *not* backward compatible:

- Default server name is `server` instead of `server1`
- `server.`*resource* becomes `domain.resources.`*resource*

- `server.`*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 8.1

In Application Server 8.1, 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 8.1, see the *Sun Java System Application Server Administration Guide*.

The table below displays a one-to-one mapping of the incompatibilities in dotted names between Application Server 7 and 8.1. 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 8 Dotted Names
server_instance`.http-listener.`*listener_id* *server_instance*`.http-service.http-listener.`*listener_id*	`server.http-service.http-listener.`*listener_id* `server-config.http-service.http-listener.`*listener_id*
server_instance`.orb` *server_instance*`.iiop-service`	`server.iiop-service` `server-config.iiop-service`
server_instance`.orblistener` *server_instance*`.iiop-listener`	`server.iiop-service.iiop-listener.`*listener_id* `server-config.iiop-service.iiop-listener.`*listener_id*
server_instance`.jdbc-resource.`*jndi_name*	`server.resources.jdbc-resource.`*jndi_name* `domain.resources.jdbc-resource.`*jndi_name*
server_instance`.jdbc-connection-pool.`*pool_id*	`server.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*	`server.resources.external-jndi-resource.`*jndi_name* `domain.resources.external.jndi-resource.`*jndi_name*
server_instance`.custom-resource.`*jndi_name*	`server.resources.custom-resource.`*jndi_name* `domain.resources.custom-resource.`*jndi_name*
server_instance`.web-container.logLevel` (see note below)	`server.log-service.module-log-levels.web-container` `server-config.log-service.module-log-levels.web-container`

Table 1-3 Incompatible Dotted Names Between Versions

Application Server 7 Dotted Names	Application Server 8 Dotted Names
server_instance.web-container.monitoringEnabled (see note below)	server.monitoring-service. module-monitoring-levels.web-container server-config.monitoring-service. module-monitoring-levels.web-container
server_instance.j2ee-application.*application_name* *server_instance*.application.*application_name*	server.applications.j2ee-application. *application_name* domain.applications.j2ee-application. *application_name*
server_instance.ejb-module.*ejb-module_name*	server.applications.ejb-module.*ejb-module_name* domain.applications.ejb-module.*ejb-module_name*
server_instance.web-module.*web-module_name*	server.applications.web-module.*web-module_name* domain.applications.web-module.*web-module_name*
server_instance.connector-module. *connector_module_name*	server.applications.connector-module. *connector_module_name* domain.applications.connector-module. *connector_module_name*
server_instance.lifecycle-module. *lifecycle_module_name*	server.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*	server.http-service.virtual-server. *virtual-server_id* server-config.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_id* *server_instance*.security-service.authrealm. *realm_id*	server.security-service.auth-realm.*realm_id* server-config.security-service-auth-realm. *realm_id*
server_instance. persistence-manager-factory-resource.*jndi_name* *server_instance*.resources.persistence-manager-facto ry-resource.*jndi_name*	server.resources.persistence-manager-factory-reso urce.*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*	server.resources.mail-resource.*jndi_name* domain.resources.mail-resource.*jndi_name*

Table 1-3 Incompatible Dotted Names Between Versions

Application Server 7 Dotted Names	Application Server 8 Dotted Names
server_instance`.profiler`	`server.java-config.profiler` `server-config.java-config.profiler`

NOTE Rows with note in previous table describe attribute names. In these instances, there is not a one-to-one relationship with the dotted names between Application Server 7 and 8.1.

Tokens in Attribute Values

The `asadmin get` command shows raw values in Application Server 8.1 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
```

displays the following value:

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

Nulls in Attribute Values

In Application Server 8, attributes with no values contained nulls. This caused problems in attributes that specified paths. In Application Server 8.1, attributes with no values contain empty strings, as they did in Application Server 7.

Chapter 2

J2EE 1.4 Compatibility Issues

The following topics are covered in this chapter:

- Binary Compatibility
- Source Compatibility
- Incompatibilities in the J2EE 1.4 Platform (since the J2EE 1.3 release)
- JAXP and SAX Incompatibilities
- Application Server 8.1 Options Incompatible with J2EE 1.4 Specification Requirements
- Application Server 8.1 Options Contrary to J2EE 1.4 Specification Recommendations

Binary Compatibility

In this Application Server 8.1 release, the included Java SDK is The Java™ 2 Platform, Enterprise Edition (J2EE™ platform), version 1.4 SDK. This version of the J2EE SDK is upwards binary-compatible with J2EE SDK, v1.3, except for the incompatibilities listed below. This means that, except for the noted incompatibilities, applications built for version 1.3 run correctly in the Sun Java System Application Server 8.1 release. For ease of reference, the version of the J2EE SDK included in this release is referred to throughout this section as J2EE 1.4.

Source Compatibility

Downward source compatibility is not supported. If source files use new J2EE APIs, they are not usable with an earlier version of the J2EE platform.

In general, the policy is as follows:

- Maintenance releases do not introduce any new APIs, so they maintain source-compatibility with one another. However, since J2EE is based on J2SE, a new Application Server release may include a new version of J2SE. Refer to the document on compatibility issues in J2SE for more information:

  `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 J2EE 1.4 Platform (since the J2EE 1.3 release)

The Sun Java System Application Server 8.1 release is based on the Java 2 Platform, Enterprise Edition, version 1.4. The Sun Java System Application Server 7 release is based on the Java 2 Platform, Enterprise Edition, version 1.3.

The Sun Java System Application Server 8.1 release is strongly compatible with previous versions of the J2EE platform. Almost all existing programs should run on the Sun Java System Application Server 8.1 release without modification. However, there are some minor potential incompatibilities that involve rare circumstances and "corner cases" that we are documenting here for completeness.

- Java Servlet Specification Version 2.4 ships with the Sun Java System Application Server 8.1 release, and can be downloaded from the following URL:

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

  Version 2.3 of the specification shipped with the J2EE 1.3 SDK. The following items discuss compatibility issues between these releases.

  - `HttpSessionListener sessionDestroyed`

    In the previous versions of the specification, this method was defined as `Notification that a session was invalidated`. As of this release, this method is changed to `Notification that a session is about to be invalidated` so that it notifies *before* the session invalidation. If the code assumed the previous behavior, it must be modified to match the new behavior.

  - `ServletRequest` methods `getRemotePort`, `getLocalName`, `getLocalAddr`, `getLocalPort`

    The following methods are added in the `ServletRequest` interface in this version of the specification. Be aware that this addition causes source incompatibility in some cases, such as when a developer implements the `ServletRequest` interface. In this case, ensure that all the new methods are implemented:

    - `public int getRemotePort()` returns the Internet Protocol (IP) source port of the client or last proxy that sent the request.
    - `public java.lang.String getLocalName()` returns the host name of the Internet Protocol (IP) interface on which the request was received.
    - `public java.lang.String getLocalAddr()` returns the Internet Protocol (IP) address of the interface on which the request was received.
    - `public int getLocalPort()` returns the Internet Protocol (IP) port number of the interface on which the request was received.

- Java Server Pages Specification 2.0 ships with the Sun Java System Application Server 8.1 release and is downloadable from the following URL:

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

  JSP Specification 1.2 shipped with the J2EE 1.3 SDK. Where possible, the JSP 2.0 Specification attempts to be fully backward compatible with the JSP 1.2 Specification. In some cases, there are ambiguities in the JSP 1.2 specification that have been clarified in the JSP 2.0 Specification. Because some JSP 1.2 containers behave differently, some applications that rely on container-specific behavior may need to be adjusted to work correctly in a JSP 2.0 environment.

  The following is a list of known backward compatibility issues of which developers who use JSP technology should be aware:

- Tag Library validators that are not namespace aware and that rely solely on the prefix parameter might not correctly validate some JSP 2.0 pages. This is because the XML view might contain tag library declarations in elements other than `jsp:root`, and might contain the same tag library declaration more than once, using different prefixes. The `uri` parameter should always be used by tag library validators instead. Existing JSP pages with existing tag libraries do not create any problems.

- Users may observe differences in I18N behavior on some containers due primarily to ambiguity in the JSP 1.2 specification. Where possible, steps were taken to minimize the impact on backward compatibility and overall, the I18N abilities of technology have been greatly improved.

  In JSP specification versions previous to JSP 2.0, JSP pages in XML syntax ("JSP documents") and those in standard syntax determined their page encoding in the same fashion, by examining the `pageEncoding` or `contentType` attributes of their page directive, defaulting to ISO-8859-1 if neither was present.

  As of the JSP Specification v2.0, the page encoding for JSP documents is determined as described in section 4.3.3 and appendix F.1 of the XML specification, and the `pageEncoding` attribute of those pages is only checked to make sure it is consistent with the page encoding determined as per the XML specification.

  As a result of this change, JSP documents that rely on their page encoding to be determined from their `pageEncoding` attribute will no longer be decoded correctly. These JSP documents must be changed to include an appropriate XML encoding declaration.

  Additionally, in the JSP 1.2 Specification, page encodings are determined on a per translation unit basis whereas in the JSP 2.0 Specification, page encodings are determined on a per-file basis. Therefore, if `a.jsp` statically includes `b.jsp`, and a page encoding is specified in `a.jsp` but not in `b.jsp`, in the JSP 1.2 Specification `a.jsp`'s encoding is used for `b.jsp`, but in the JSP 2.0 Specification, the default encoding is used for `b.jsp`.

- The type coercion rules (shown in Table JSP.1-11 in the JSP 2.0 Specification) have been reconciled with the EL coercion rules. There are some exceptional conditions that no longer result in an exception in the JSP 2.0 Specification. In particular, when passing an empty `String("")` to an attribute of a numeric type, a translation error or a `NumberFormatException` used to occur, whereas in the JSP 2.0 Specification, a `0` is passed in instead. See Table JSP.1-11 in the JSP 2.0

Specification for details. In general, this is not expected to cause any problems because these would have been exceptional conditions in the JSP 1.2 Specification and the specification allowed for these exceptions to occur at either translation time or request time.

- The JSP container uses the version of `web.xml` to determine the default behavior of various container features. The following is a list of items of which JSP developers should be aware when upgrading their `web.xml` from Servlet version 2.3 Specification to Servlet version 2.4 Specification.
  - EL expressions are ignored by default in applications created with JSP 1.2 technology. When upgrading a Web application to the JSP 2.0 Specification, EL expressions are interpreted by default. The escape sequence `\$` can be used to escape EL expressions that should not be interpreted by the container. Alternatively, the `isELIgnored` page directive attribute, or the `el-ignored` configuration element can deactivate EL for entire translation units. Users of JSTL 1.0 need to either upgrade their `taglib/` imports to the JSTL 1.1 URIs, or they need to use the `_rt` versions of the tags (for example `c_rt` instead of `c`, or `fmt_rt` instead of `fmt`).
  - Files with an extension of `.jspx` are interpreted as JSP documents by default. Use the JSP configuration element `is-xml` to treat `.jspx` files as regular JSP pages. There is no way to disassociate `.jspx` from the JSP container.
  - The escape sequence `\$` was not reserved in the JSP 1.2 Specification. Any template text or attribute value that appeared as `\$` in the JSP 1.2 Specification used to output `\$` but now outputs just `$`.

JAXP and SAX Incompatibilities

Sun Java System Application Server 8.1 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 here:

http://java.sun.com/j2se/1.5.0/docs/guide/xml/jaxp/JAXP-Compatibility_150.html#SAX

Application Server 8.1 Options Incompatible with J2EE 1.4 Specification Requirements

Sun Java System Application Server 8.1 is compatible with the Java 2 Platform, Enterprise Edition specification by default. In this case, all portable J2EE programs run on the Application Server without modification. However, as allowed by the J2EE compatibility requirements, it is possible to configure applications to use features of the Sun Java System Application Server 8.1 that are not compatible with the J2EE specification.

The `pass-by-reference` element in the `sun-ejb-jar.xml` file only applies to remote calls. As defined in the EJB 2.0 specification, section 5.4, 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 2.0 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's 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 *Developer's Guide*.

Application Server 8.1 Options Contrary to J2EE 1.4 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 2.3 specification, section 9.7.2. If set to `false`, the classloader delegation behavior complies with what is recommended in the Servlet 2.3 specification, section 9.7.2.

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 *Developer's Guide*.

Chapter 3

Upgrading an Application Server Installation

You can upgrade to Sun Java System Application Server Platform Edition 8.1 (hereafter called Application Server) from Sun Java(TM) System Application Server 7.x (formerly Sun ONE(TM) Application Server 7.x) or a Sun Java System 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.

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

Table 3-1 Supported Upgrade Paths

Source Installation	8.1Platform Edition	8.1 Enterprise Edition
7.XPE	X	X
7.XSE		X
7.XEE		X
8.0PE	X	X
8.1PE		X

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 a graphical user interface (Upgrade Wizard), for completing the upgrade. If you issue the `asupgrade` command with no options, the Upgrade Wizard GUI will be displayed. If the `asupgrade` command is used in command-line mode and all of the required information is not supplied, an interviewer will request information for any required options that were 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 will be 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 Sun Java System Web Server-specific features will be upgraded seamlessly. Configuration files related to HTTP path, CGI bin, SHTML, and NSAPI plug-ins will not be upgraded.

Application archives (EAR files) and component archives (JAR, WAR, and RAR files) that are deployed in the Application Server 7.x/8.0 environment do not require any modification to run on Application Server 8.1.

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.

This chapter discusses the following topics:

- Upgrading Through the Upgrade Utility
- Upgrading Through the Wizard
- Correcting Potential PE Upgrade Problems
- Correcting Potential PE Upgrade Problems

Upgrading Through the Upgrade Utility

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

```
asupgrade [--console ] [--version ] [--help ]
       [--source applicationserver_7.x/8.x_installation]
       [--target applicationserver_8.1_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]
       [--clinstancefile file1 [, file2, file3, ... filen]]
```

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 Sun Java System Application Server 8.1.
-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 8.1 EE.

Table 3-2 asupgrade Utility Command Options

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.
-i	--clinstancefile	The path to the cluster file. The default filename is $AS_INSTALL/conf/clinstance.conf.

The following examples show how to use the asupgrade command-line utility to upgrade an existing application server installation to Application Server 8.1.

Example 1: Upgrading an Application Server 7 Installation to Application Server 8.1 with Prompts for Certificate Migration.

This example shows how to upgrade a Sun Java System Application Server 7 installation to Sun Java System Application Server 8.1. 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/sunas7 --target /home/sjsas8.1
```

Example 2: Upgrading an Application Server 7.0 PE Installation with NSS Certificates to Application Server 8.1 PE

This example shows how to upgrade a Sun Java System Application Server 7.0 PE installation to Sun Java System Application Server 8.1 PE. The NSS certificates from the 7.0 PE source server will be converted to JKS and CA certificates in the 8.1 PE target server.

```
% asupgrade --adminuser admin --passwordfile password.txt
       --source /home/sjsas7.0 --target /home/sjsas8.1
       --domain domain1
       --nsspwdfile /home/sjsas7.0/nsspassword.txt
       --jkspwdfile /home/sjsas7.0/jkspassword.txt
       --capwdfile /home/sjsas7.0/capassword.txt
```

Example 3: Upgrading an Application Server 8.0 PE Installation with JKS and CA Certificates to Application Server 8.1 PE

This example shows how to upgrade a Sun Java System Application Server 8.0 PE installation to Sun Java System Application Server 8.1 PE. JKS and CA certificates will be migrated.

```
% asupgrade --adminuser admin --passwordfile password.txt
       --source /home/sjsas8.0 --target /home/sjsas8.1
       --domain domain1
       --jkspwdfile /home/sjsas8.0/jkspassword.txt
       --capwdfile /home/sjsas8.1/capassword.txt
```

Upgrading Through the Wizard

The Upgrade wizard provides a graphical user interface (GUI). 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:

1. In the Source Installation Directory field, enter the location of the Sun Java System Application Server 7 (formerly Sun ONE™ Application Server 7) or Sun Java System Application Server 8.x installation from which to import the configuration.

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. If a Sun Java System Application Server 7.1 Enterprise Edition installation with clusters and no security certificates is being upgraded to Sun Java Systems Application Server 8.1 Enterprise Edition, press the Next button and continue with Step 10. All other upgrades without certificates continue with Step 12. Continue with Step 4 if security certificates need to be transferred.

4. 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.

5. From the Transfer Security Certificates screen, press the Add Domain button to add domains with certificates to be transferred. The Add Domain dialog displays.

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

7. Click the OK button when done. The Transfer Security Certificates screen will be displayed again.

8. Repeat Step 5 and Step 6 until all the domains that have certificates to be transferred have been added.

9. After all of the domains that contain certificates to be transferred have been added, press the Next button and continue with Step 12 or with Step 10 if cluster configuration information needs to be transferred.

10. If a Sun Java Systems Application Server 7.1 Enterprise Edition installation with clusters is being upgraded to Sun Java Systems Application Server 8.1 Enterprise Edition, the Transfer Cluster Configurations screen will be displayed. Press the Add Cluster button. The Select `clinstance.conf` file dialog box will be displayed. Choose clinstance file and click the Open button. The `clinstance.conf` file will be added to the list.

11. Enter the cluster file name, which contains the cluster configuration information to be migrated. Repeat this process until all the cluster configuration files that need to be migrated have been added, then press the Next button.

12. The Upgrade Results screen displays, showing the status of the upgrade operation in the Results field.
13. Click the Finish button to close the Upgrade Tool when the upgrade process is complete.

Correcting Potential PE Upgrade Problems

This section addresses the following issues that could occur during an upgrade to Application Server 8.1:

- Migrating Additional HTTP Listeners Defined on the Source Server to the Target PE Server
- Eliminating Problems Encountered When A Single Domain has Multiple Certificate Database Passwords

Migrating 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.

Eliminating Problems Encountered When A Single Domain has Multiple Certificate Database Passwords

If the upgrade includes certificates, provide the passwords for the source PKCS12 file and the target JKS keyfile for each domain that contains certificates to be migrated. Since Application Server 7 uses a different certificate store format (NSS) than Application Server 8 PE (JSSE), the migration keys and certificates are converted to the new format. Only one certificate database password per domain is supported. If multiple certificate database passwords are used in a single domain, make all of the passwords the same before starting the upgrade. Then reset the passwords after the upgrade has been completed.

Chapter 4

Understanding Migration

This chapter addresses the following topics:

- J2EE Component Standards
- J2EE Application Components
- Migration and Deployment

J2EE Component Standards

Sun Java System Application Server Platform Edition 8.1 2005Q1(hereafter called Application Server) is a J2EE v1.4-compliant server based on the component standards developed by the Java community. By contrast, Sun Java System Application Server 7 (Application Server 7) is a J2EE v1.3-compliant server and Sun ONE Application Server 6.x (Application Server 6.x) is a J2EE v1.2-compliant server. Between the three J2EE versions, there are considerable differences with the J2EE application component APIs.

The following table characterizes the differences between the component APIs used with the J2EE v1.4-compliant Sun Java System Application Server Platform Edition 8.1, the J2EE v1.3-compliant Sun ONE Application Server 7, and the J2EE v1.2-compliant Sun ONE Application Server 6.x.

Table 4-1 Application Server Version Comparison of APIs for J2EE Components

Component API	Sun ONE Application Server 6.x	Sun Java System Application Server 7	Sun Java System Application Server Platform Edition 8.1
JDK	1.2.2	1.4	1.4
Servlet	2.2	2.3	2.4

Table 4-1 Application Server Version Comparison of APIs for J2EE Components

JSP	1.1	1.2	2.0
JDBC	2.0	2.0	2.1, 3.0
EJB	1.1	2.0	2.0
JNDI	1.2	1.2	1.2.1
JMS	1.0	1.1	1.1
JTA	1.0	1.01	1.01

J2EE Application Components

J2EE 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. J2EE v1.4 architecture includes several component APIs. Prominent J2EE components include:

- Client Application
- Web Application
- Enterprise JavaBean (EJB)
- Connector
- Enterprise Application Archive (EAR)

J2EE components are packaged separately and bundled into a J2EE 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 J2EE application. A J2EE application is composed of one or more enterprise bean(s), Web, or application client component modules. The final enterprise solution can use one J2EE application or be made up of two or more J2EE applications, depending on design requirements.

A J2EE 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 J2EE 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 J2EE application components, modules, and files. For more information on migrating various J2EE components, refer to Chapter 6, "Migrating from Application Server 6.x/7.x to Application Server 8.1.".

For more background information on J2EE, see the following references:

- J2EE tutorial - `http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html`
- J2EE overview - `http://java.sun.com/j2ee/overview.html`
- J2EE topics - `http://java.sun.com/j2ee`

Migration and Deployment

This section describes the need to migrate J2EE applications and the particular files that must be migrated. Following successful migration, a J2EE application is redeployed to the Application Server.

Redeployment is also described in this section.

The following topics are addressed:

- Why is Migration Necessary?
- What Needs to be Migrated
- What is Deployment of Migrated Applications?

Why is Migration Necessary?

Although J2EE 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 J2EE 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 J2EE application components, modules, and files deployed on other established application server platforms. Such differences require mappings between earlier implementation details of the J2EE 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 J2EE 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 J2EE components/applications. The J2EE 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

J2EE 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 J2EE 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 J2EE is an evolving standard, different products can support different versions of J2EE component APIs. This is another aspect that migration addresses.

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.

The act of deploying a migrated application typically refers to using the standard deployment actions outlined in the *Sun Java System Application Server Platform Edition 8.1 Administration Guide.* However, when migration activities are performed with automated tools, such as the Migration Tool for Sun Java System Application Server 8 (for J2EE 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 Migration Tools and Resources for more information about migration tools that are available.

Chapter 5

Migrating from EJB 1.1 to EJB 2.0

Although the EJB 1.1 specification will continue to be supported in Sun Java System Application Server Platform Edition 8.1, the use of the EJB 2.0 architecture is recommended to leverage its enhanced capabilities.

To migrate EJB 1.1 to EJB 2.0 a number of modifications are required, including within the source code of components.

Essentially, the required modifications relate to the differences between EJB 1.1 and EJB 2.0, all of which are described in the following topics.

- EJB Query Language
- Local Interfaces
- EJB 2.0 Container-Managed Persistence (CMP)
- Migrating EJB Client Applications
- Migrating CMP Entity EJBs

EJB Query Language

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.

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.

Local Interfaces

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.

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.

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 2.0 Container-Managed Persistence (CMP)

The EJB 2.0 specification expanded 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.

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.

You must also change the way you code the bean's implementation class. 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 following topics are discussed in this section:

- Defining Persistent Fields
- Defining Entity Bean Relationships
- Message-Driven Beans

Defining Persistent Fields

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`.

In the implementation class, note that you do not declare the CMP and CMR fields as public variables. Instead, you define `get` and `set` methods in the entity bean to retrieve and set the values of these CMP and CMR fields. In this sense, beans using the 2.0 CMP follow the JavaBeans model: instead of accessing instance variables directly, clients use the entity bean's `get` and `set` methods to retrieve and set these instance variables. Keep in mind that the `get` and `set` methods only pertain to variables that have been designated as CMP or CMR fields.

Defining Entity Bean Relationships

As noted previously, the EJB 1.1 architecture does not support CMRs between entity beans. The EJB 2.0 architecture does support both one-to-one and one-to-many CMRs. Relationships are expressed using CMR fields, and these fields are marked as such in the deployment descriptor. You set up the CMR fields in the deployment descriptor using the appropriate deployment tool for your application server.

Similar to CMP fields, the bean does not declare the CMR fields as instance variables. Instead, the bean provides get and set methods for these fields.

Message-Driven Beans

Message-driven beans are another 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). The JMS API is an integral part of the J2EE 1.3 and J2EE 1.4 platform.

Asynchronous messaging allows applications to communicate by exchanging messages so that senders are independent of receivers. The sender sends its message and does not have to wait for the receiver to receive or process that message. This differs from synchronous communication, which requires the component that is invoking a method on another component to wait or block until the processing completes and control returns to the caller component.

Migrating EJB Client Applications

This section includes the following topics:

- Declaring EJBs in the JNDI Context
- Recap on Using EJB JNDI References

Declaring EJBs in the JNDI Context

In Sun Java System Application Server Platform Edition 8.1, EJBs are systematically mapped to the JNDI sub-context "*ejb/*". If we attribute the JNDI name "*Account*" to an EJB, then Sun Java System Application Server Platform Edition 8.1 will automatically create the reference "*ejb/Account*" in the global JNDI context. The clients of this EJB will 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 we are considering is responsible for retrieving the home interface of the EJB, so as to enable a BankTeller object to be instantiated and a remote interface for this object to be retrieved, in order to 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 Sun Java System Application Server Platform Edition 8.1.

Recap on 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 Sun Java System Application Server Platform Edition 8.1. So, if an EJB is given to the JNDI name "*BankTeller*" in its deployment descriptor, the reference to this EJB will be "translated" by Sun Java System Application Server Platform Edition 8.1 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 Sun Java System Application Server Platform Edition 8.1. Nonetheless, it is preferable to stay as close as possible to the J2EE 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

This section describes the steps to migrate your application components from the EJB 1.1 architecture to the EJB 2.0 architecture.

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.

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. CMP2.0 finders cannot return `java.util.Enumeration`.

"Migrating the Bean Class," 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 Platform Edition 8.1.

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
   ```

   ```
   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 like this:

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

   The changed method body (after migration) should be:

   ```
   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. All the protected variables declared in the `ejbPostCreate()`methods in step 5 must be initialized. 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 /*protected variable*/ = getName();    /*abstract accessor*/
//inserted in this step
id /*protected variable*/ = getId();        /*abstract accessor*/
//inserted in this step
            }
```

8. Inside the `ejbLoad` method, you must 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
                   ………..                //already present code
            }
```

9. Similarly, you will have to 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. As a last change to the bean class source (`.java`) file, examine the whole code and replace all occurrences of any `<cmp-field>` variable name 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 Platform Edition 8.1.

Migration of ejb-jar.xml

To migrate the file `ejb-jar.xml` to Sun Java System Application Server Platform Edition 8.1, perform the following steps:

1. In the `ejb-jar.xml`, convert all `<cmp-fields>` to lowercase.
2. In the `ejb-jar.xml` file, 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's 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 in the `ejb-jar.xml`, 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 Platform Edition 8.1 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>
```

Chapter 6

Migrating from Application Server 6.x/7.x to Application Server 8.1

This chapter describes the considerations and strategies that are needed when moving J2EE applications from Application Server 6.x and Application Server 7 to the Application Server Platform Edition 8.1product line. However, Application Server 8.1 provides backward compatibility standard, with Application Server 7 as the baseline. That is, applications developed in Application Server 7 can be deployable directly to Application Server 8.1 with minimum or no changes.

The sections that follow describe issues that arise while migrating the main components of a typical J2EE application from Application Server 6.x/7.x to Application Server Platform Edition 8.1.

This chapter contains the following sections:

The migration issues described in this chapter are based on an actual migration that was performed for a J2EE application called *iBank*, a simulated online banking service, from Application Server 6.x to Sun Java System Application Server Platform Edition 8.1. This application reflects all aspects of a traditional J2EE application.

The following areas of the J2EE 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

The iBank application is presented in detail in *Appendix A - iBank Application Specification*

Migrating Deployment Descriptors

There are two types of deployment descriptors, namely, Standard Deployment Descriptors and Runtime Deployment Descriptors. Standard deployment descriptors are portable across J2EE 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	Target Deployment Descriptor
`ejb-jar.xml` - 1.1	`ejb-jar.xml` - 2.0
`web.xml`	`web.xml`
`application.xml`	`application.xml`

The J2EE standard deployment descriptors `ejb-jar.xml`, `web.xml` and `application.xml` are not modified significantly. However, the `ejb-jar.xml` deployment descriptor is modified to make it compliant with EJB 2.0 specification in order to make the application deployable on Sun Java System Application Server Platform Edition 8.1.

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. This section describes how you can manually create the runtime deployment descriptors and migrate relevant information.

The following table summarizes the deployment descriptor migration mapping.

Source Deployment Descriptor	Target Deployment Descriptor
`ias-ejb-jar.xml`	`sun-ejb-jar.xml`
`<bean-name>-ias-cmp.xml`	`sun-cmp-mappings.xml`
`ias-web.xml`	`sun-web.xml`

The standard deployment descriptors of Application Server 6.x needs modification when moving to Application Server 8.1 because of non-conformance with the DTDs.

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 Platform Edition 8.1 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 (J2EE-standard Web application module) deploying it on the application server.

According to the J2EE 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 8.1does 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 Platform Edition 8.1 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 J2EE 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
- Declaring EJBs in the JNDI Context

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 Declaring EJBs in the JNDI Context in “Migrating from EJB 1.1 to EJB 2.0” on page 51.”

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 8.1does 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 Platform Edition 8.1 does not require any changes to the Java/JSP code. The following changes are, however, still required.

- `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 available at URL `http://java.sun.com/dtd/web-app_2_3.dtd`.. This DTD fortunately, 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">
  ```

- `ias-web.xml`

  In Application Server Platform Edition 8.1, 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," 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 the `web.xml` and `ias-web.xml` files are migrated, 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 supports EJB 2.0. 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 use 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 andApplication Server 8.1do 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 URL `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 URL `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 Understanding Migration, while Application Server 6.x supports the EJB 1.1 specification, Application Server also supports the EJB 2.0 specification. The EJB 2.0 specification introduces the following new features and functions to the architecture:

- Message Driven Beans (MDBs)
- Improvements in Container-Managed Persistence (CMP)
- Container-managed relationships for entity beans with CMP
- Local interfaces
- EJB Query Language (EJB QL)

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

For detailed information on migrating from EJB 1.1 to EJB 2.0, please refer to Chapter 5, "Migrating from EJB 1.1 to EJB 2.0."

EJB Changes Specific to Application Server Platform Edition 8.1

Migrating EJBs from Application Server 6.x to Application Server 8.1 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 URL `http://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 URL `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 finders 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, v2.0, 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.” For information on Message Driven Bean implementation in Application Server Platform Edition 8.1, see *Sun Java System Application Server Platform Edition 8.1, Developer’s Guide to Enterprise Java Bean Technology*.

Migrating Enterprise Applications

According to the J2EE 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 8.1support 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:

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=tes
t&arg2=m
```

Application Server 8.1 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 8.1. Porting such applications to Application Server 8.1 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 8.1:

```
---------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 Migrating a Sample Application - an Overview.

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 JDBC Code

With the JDBC API, there are two methods of database access:

- Establishing Connections Through the DriverManager Interface

  (JDBC 1.0 API), by loading a specific driver and providing a connection URL. This method is used by other Application Servers, such as IBM’s WebSphere 4.0
- Using JDBC 2.0 Data Sources

The `DataSource` interface (JDBC 2.0 API) can be used via a configurable connection pool. According to J2EE 1.2, a data source is accessed through the JNDI naming service

NOTE Application Server does not support the Native Type 2 JDBC drivers bundled with Application Server 6.x. Code that uses the Type 2 drivers to access third party JDBC drivers, must be manually migrated.

Establishing Connections Through the DriverManager Interface

Although this database access method is not recommended, as it is obsolete and is not very effective, there can be some applications that still use this approach.

In this case, the access code is similar to the following:

```
public static final String driver =   "oracle.jdbc.driver.OracleDriver";
public static final String url =
"jdbc:oracle:thin:tmb_user/tmb_user@iben:1521:tmbank";
Class.forName(driver).newInstance();
Properties props = new Properties();
props.setProperty("user", "tmb_user");
props.setProperty("password", "tmb_user");
Connection conn = DriverManager.getConnection(url, props);
```

This code can be fully ported from Application Server 6.x to Application Server, as long as the Application Server is able to locate the classes needed to load the right JDBC driver. In order to make the required classes accessible to the application deployed in the Application Server, place the archive (JAR or ZIP) for the driver implementation in the `/lib` directory of the Application Server installation directory.

Modify the *CLASSPATH* by setting the path for the driver through the Admin Console GUI.

- Click the server instance “server1.”
- Click the tab “JVM Settings” from the right pane.
- Click the option Path Settings and add the path in the classpath suffix text entry box.
- Once the changes are made, click “Save.”
- Apply the new settings.
- Restart the server to modify the configuration file, `server.xml`.

Using JDBC 2.0 Data Sources

Using JDBC 2.0 data sources to access a database provides performance advantages, such as transparent connection pooling, enhanced productivity by simplifying code and implementation, and code portability.

If there is a datasource by the name ‘xyz’ on Application Server 6.x application and you do not want any impact on your JNDI lookup code, make sure that the datasource you create for Application Server 8.1 is prefixed with jdbc. For example: `jdbc/xyz`.

For information on configuring JDBC Datasource, see the *Sun Java System Application Server 8.1 Administrator’s Guide*.

Looking Up the Data Source Via JNDI To Obtain a Connection

To obtain a connection from a data source, do the following:

1. Obtain the initial JNDI context.

   To guarantee portability between different environments, the code used to retrieve an InitialContext object (in a servlet, in a JSP page, or an EJB) is as follows:

   ```
   InitialContext ctx = new InitialContext();
   ```

2. Use a JNDI lookup to obtain a data source reference.

   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 cast as a DataSource type object:

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

3. Use the data source reference to obtain the connection.

   This operation requires the following line of code:

   ```
   conn = ds.getConnection();
   ```

Application Server 6.x and Application Server both follow these technique to obtain a connection from the data source.

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 Platform Edition 8.1

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 the *Sun Java System Application Server Platform Edition 8.1 Developer's Guide to Clients*.

Using ACC in Application Server 6.x and Sun Java System Application Server Platform Edition 8.1

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 Platform Edition 8.1, 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 Platform Edition 8.1. 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 Platform Edition 8.1, 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 Platform Edition 8.1, 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 Platform Edition 8.1.

The default IIOP port is 3700 in Sun Java System Application Server Platform Edition 8.1; 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 J2EE application from Sun ONE Application Server 6.x to Sun Java System Application Server Platform Edition 8.1. 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 J2EE 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 8.1. iBank simulates an online banking service and covers all of the aspects traditionally associated with a J2EE application.

The major points of the J2EE 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

The iBank application is presented in detail in Appendix A, “iBank Application Specification.”

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 64.

Choosing the Target

To start, choose Sun Java System Application Server Platform Edition 8.1 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 Platform Edition 8.1 Installation Guide*.

If you are using Migration Tool for Sun Java System Application Server 8.1 to migrate the components, install the tool. The Migration Tool can be downloaded from the following location:

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

For information on how to use the Migration Tool for Sun Java System Application Server 8.1, 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
- Adding a JDBC Resource
- Adding a Persistence Manager

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.PersistenceManagerFac
toryImpljdo/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 Platform Edition 8.1. The process for deploying an application is described below:

The Sun Java System Application Server Platform Edition 8.1 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 Platform Edition 8.1.

Migration Tool for Sun Java System Application Server 8.1

The (hereafter called Migration Tool) migrates J2EE applications from other server platforms to Sun Java System Application Server Platform Edition 8.1.

The following source platforms are supported for Sun Java System Application Server Platform Edition 8.1:

- Sun ONE Application Server 6.*x*
- Sun Java System Application Server 7
- J2EE Reference Implementation Application Server (RI) 1.3, 1.4 Beta1
- 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

Migration Tool automates the migration of J2EE applications to Sun Java System Application Server Platform Edition 8.1, 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`

For detailed information on how to install and use the tool, see 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 Platform Edition 8.1 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/`

More Migration Information

This section provides references to additional migration documents.

Migrating from KIVA/NAS/NetDynamics Application Servers

For information about migrating KIVA / NAS / NetDynamics applications to Sun ONE Application Server 6.0, see the *Sun ONE Application Server Migration Guide* at the following URL:

`http://docs.sun.com/db/doc/816-5780-10`

For information about migrating KIVA / NAS / NetDynamics applications to Sun ONE Application Server 6.5, see the *Sun ONE Application Server 6.5 Migration Guide* at the following URL:

`http://docs.sun.com/db/doc/816-5793-11`

For information about migrating KIVA / NAS / NetDynamics applications to Sun Java System Application Server 7, see *Sun Java System Application Server 7 Migrating and Redeploying Server Applications Guide* at the following URL:

`http://docs.sun.com/db/doc/817-2158-10`

Appendix A

iBank Application Specification

The iBank application is used as the migration sample. This application simulates a basic online banking service with the following functionality:

- Log on to the online banking service
- View / edit personal details and branch details
- Summary view of accounts showing cleared balances
- Facility to drill down by account to view individual transaction history
- Money transfer service, allowing online transfer of funds between accounts
- Compound interest earnings projection over a number of years for a given principal and annual yield rate

The application is designed after the MVC (Model-View-Controller) model where:

- EJBs are used to define the business and data model components of the application
- Java Server Pages handle the presentation logic and represent the View.
- Servlets play the role of Controllers and handle application logic, taking charge of calling the business logic components and accessing business data via EJBs (the Model), and dispatching processed data for display to Java Server Pages (the View).

Standard J2EE methods are used for assembling and deploying the application components. This includes the definition of deployment descriptors and assembling the application components within the archive files:

- AWAR archive file for the Web application including HTML pages, images, Servlets, JSPs and custom tag libraries, and ancillary server-side Java classes.

- EJB-JAR archive files for the assembling of one or more EJBs, including deployment descriptor, bean class and interfaces, stub and skeleton classes, and other helper classes as required.
- An EAR archive file for the packaging of the enterprise application module that includes the Web application module and the EJB modules used by the application.

The use of standard J2EE assembling methods will be useful in pointing out any differences between Sun ONE Application Server 6.x/7.x and Sun Java System Application Server Platform Edition 8.1, and any issues arising thereof.

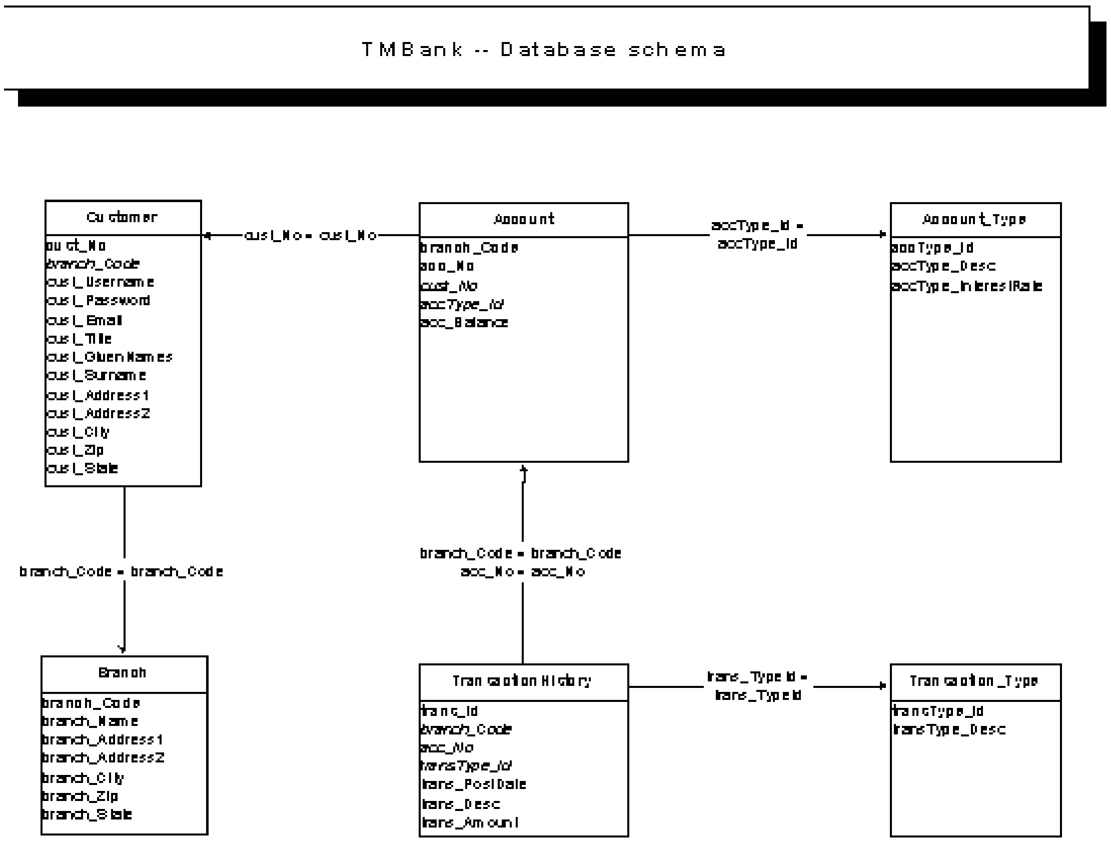
Database Schema

The iBank database schema is derived from the following business rules:

- The iBank company has local branches in major cities.
- A Branch manages all customers within its regional area.
- A Customer has one or more accounts held at their regional branch.
- A customer Account is uniquely identified by the branch code and account number, and also holds the number of the customer to which it belongs. The current cleared balance available is also stored with the account.
- Accounts are of a particular Account Type that is used to distinguish between several kinds of accounts (checking account, savings account, etc.).
- Each Account Type stores a number of particulars that apply to all accounts of this type (regardless of branch or customer) such as interest rate and allowed overdraft limit.
- Every time a customer receives or pays money into/from one of their accounts, the transaction is recorded in a global transaction log, the Transaction History.
- The Transaction History stores details about individual transactions, such as the relevant branch code and account number, the date the transaction was posted (recorded), a code identifying the type of transaction and a complementary description of the particular transaction, and the amount for the transaction.
- Transaction types allow different types of transactions to be distinguished, such as cash deposit, credit card payment, fund transfer between accounts, and so on.

Figure A-1, the entity-relationship diagram shown below, illustrates these business rules.

Figure A-1 Database Schema



The database model translates as a series of table definitions below, where primary key columns are printed in bold type, while foreign key columns are shown in italics.

BRANCH			
BRANCH_CODE	CHAR(4)	NOT NULL	4-digit code identifying the branch
BRANCH_NAME	VARCHAR(40)	NOT NULL	Name of the branch
BRANCH_ADDRESS1	VARCHAR(60)	NOT NULL	Branch postal address, street address, 1st line

BRANCH_ADDRESS2	VARCHAR(60)		Branch postal address, street address, 2nd line
BRANCH_CITY	VARCHAR(30)	NOT NULL	Branch postal address, City
BRANCH_ZIP	VARCHAR(10)	NOT NULL	Branch postal address, Zip code
BRANCH_STATE	CHAR(2)	NOT NULL	Branch postal address, State abbreviation

CUSTOMER			
CUST_NO	INT	NOT NULL	iBank customer number (global)
BRANCH_CODE	CHAR(4)	NOT NULL	References this customer's branch
CUST_USERNAME	VARCHAR(16)	NOT NULL	Customer's login username
CUST_PASSWORD	VARCHAR(10)	NOT NULL	Customer's login password
CUST_EMAIL	VARCHAR(40)		Customer's e-mail address
CUST_TITLE	VARCHAR(3)	NOT NULL	Customer's courtesy title
CUST_GIVENNAMES	VARCHAR(40)	NOT NULL	Customer's given names
CUST_SURNAME	VARCHAR(40)	NOT NULL	Customer's family name
CUST_ADDRESS1	VARCHAR(60)	NOT NULL	Customer postal address, street address, 1st line
CUST_ADDRESS2	VARCHAR(60)		Customer postal address, street address, 2nd line
CUST_CITY	VARCHAR(30)	NOT NULL	Customer postal address, City
CUST_ZIP	VARCHAR(10)	NOT NULL	Customer postal address, Zip code
CUST_STATE	CHAR(2)	NOT NULL	Customer postal address, State abbreviation

ACCOUNT_TYPE			
ACCTYPE_ID	CHAR(3)	NOT NULL	3-letter account type code
ACCTYPE_DESC	VARCHAR(30)	NOT NULL	Account type description
ACCTYPE_INTERESTRATE	DECIMAL(4,2)	DEFAULT 0.0	Annual interest rate

ACCOUNT

BRANCH_CODE	CHAR(4)	NOT NULL	branch code (primary-key part 1)
ACC_NO	CHAR(8)	NOT NULL	account no. (primary-key part 2)
CUST_NO	INT	NOT NULL	Customer to whom accounts belongs
ACCTYPE_ID	CHAR(3)	NOT NULL	Account type, references ACCOUNT_TYPE
ACC_BALANCE	DECIMAL(10,2)	DEFAULT 0.0	Cleared balance available

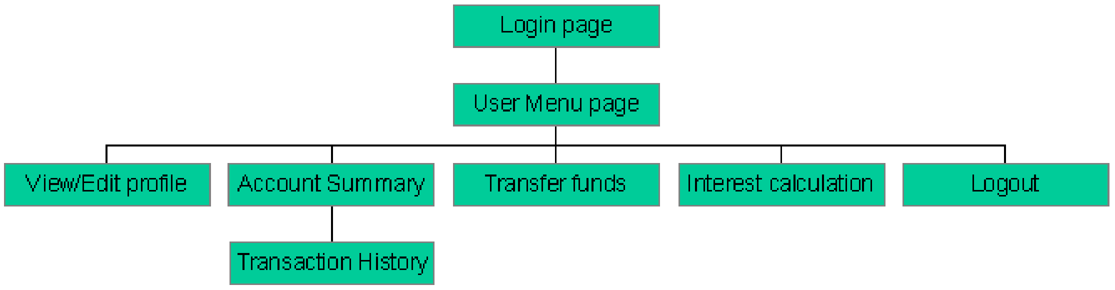
TRANSACTION_TYPE			
TRANSTYPE_ID	CHAR(4)	NOT NULL	A 4-letter transaction type code
TRANSTYPE_DESC	VARCHAR(40)	NOT NULL	Human-readable description of code

TRANSACTION_HISTORY			
TRANS_ID	LONGINT	NOT NULL	Global transaction serial no
BRANCH_CODE	CHAR(4)	NOT NULL	key referencing ACCOUNT part 1
ACC_NO	CHAR(8)	NOT NULL	key referencing ACCOUNT part 2
TRANSTYPE_ID	CHAR(4)	NOT NULL	References TRANSACTION_TYPE
TRANS_POSTDATE	TIMESTAMP	NOT NULL	Date & time transaction was posted
TRANS_DESC	VARCHAR(40)		Additional details for the transaction
TRANS_AMOUNT	DECIMAL(10,2)	NOT NULL	Money amount for this transaction

Application Navigation and Logic

Figure A-2 provides a high-level view of application navigation.

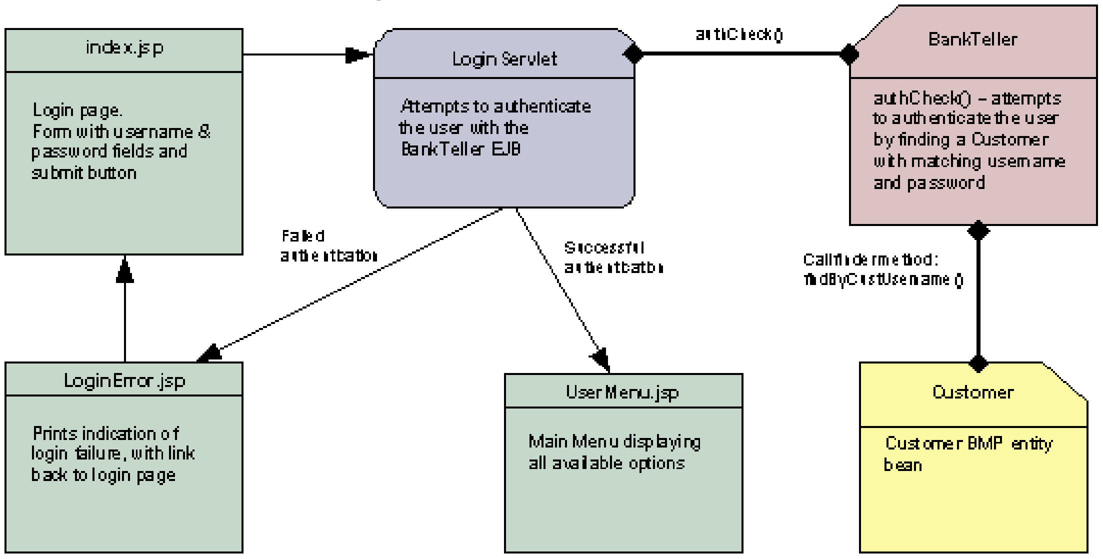
Figure A-2 Application Navigation and Logic



Login Process

Figure A-3 shows the login process used in the iBank application.

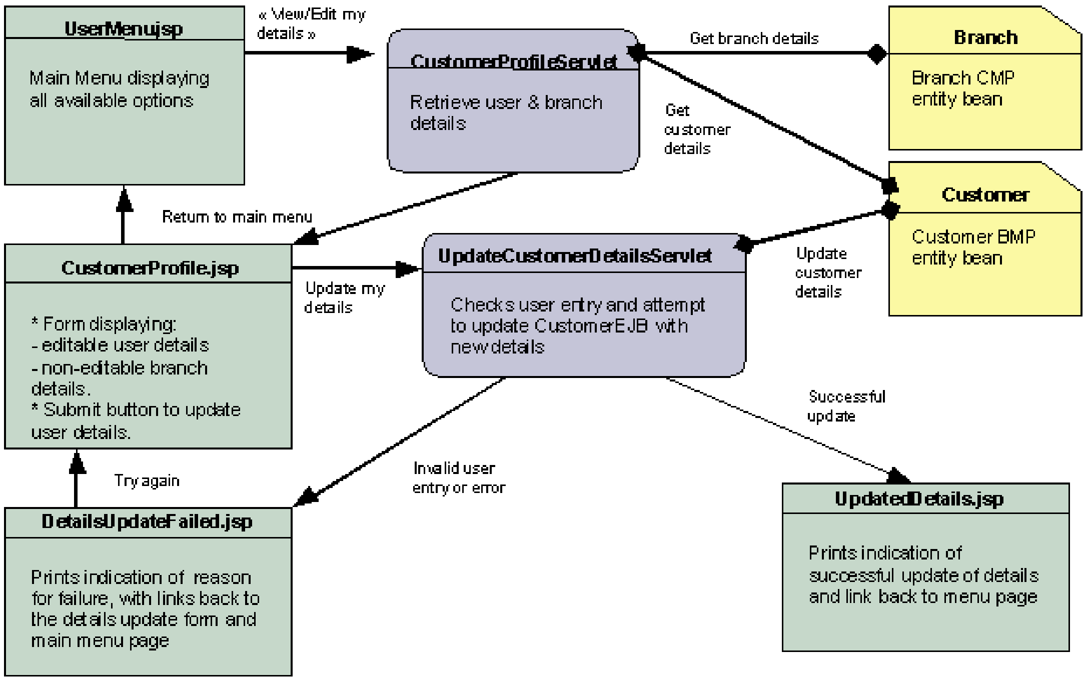
Figure A-3 Login Process


index.jsp
Login page. Form with username & password fields and submit button
Login Servlet
Attempts to authenticate the user with the BankTeller EJB
authCheck()
BankTeller
authCheck() – attempts to authenticate the user by finding a Customer with matching username and password
Failed authentication
Successful authentication
Call finder method: findByCustUsername()
LoginError.jsp
Prints indication of login failure, with link back to login page
UserMenu.jsp
Main Menu displaying all available options
Customer
Customer BMP entity bean

View/Edit Details

Figure A-4 shows the view / edit details process used in the iBank application.

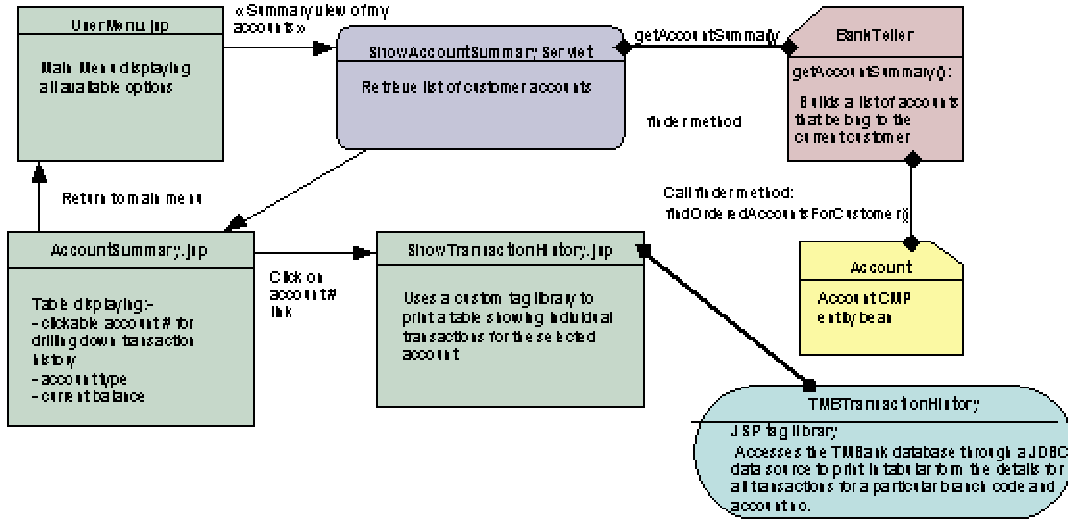
Figure A-4 View / Edit Details Process


UserMenu.jsp
Main Menu displaying all available options
« View/Edit my details »
CustomerProfileServlet
Retrieve user & branch details
Get branch details
Branch
Branch CMP entity bean
Get customer details
Customer
Customer BMP entity bean
Return to main menu
CustomerProfile.jsp
* Form displaying:
- editable user details
- non-editable branch details.
* Submit button to update user details.
Update my details
UpdateCustomerDetailsServlet
Checks user entry and attempt to update CustomerEJB with new details
Update customer details
Successful update
Try again
Invalid user entry or error
DetailsUpdateFailed.jsp
Prints indication of reason for failure, with links back to the details update form and main menu page
UpdatedDetails.jsp
Prints indication of successful update of details and link back to menu page

Account Summary and Transaction History

Figure A-5 shows how the account summary and transaction history work in the iBank application.

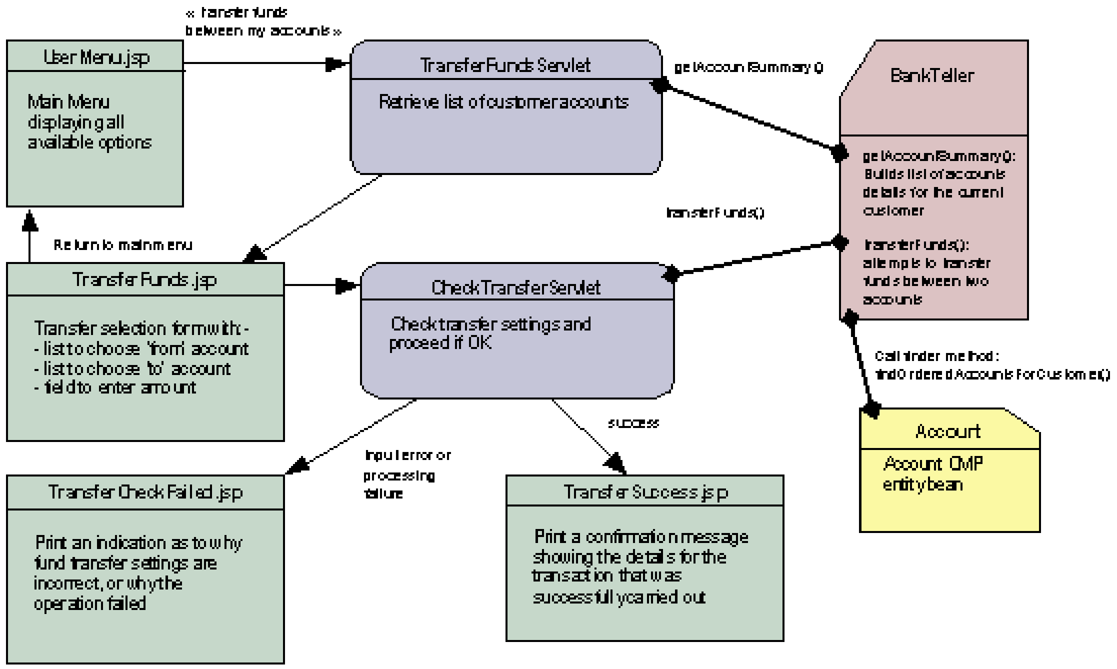
Figure A-5 Account Summary and Transaction History


UserMenu.jsp
Main Menu displaying all available options
«Summary view of my accounts»
ShowAccountSummary Servlet
Retrieve list of customer accounts
getAccountSummary
BankTeller
getAccountSummary():
Builds a list of accounts that belong to the current customer
finder method
Call finder method:
findOrderedAccountsForCustomer()
Return to main menu
AccountSummary.jsp
Table displaying:-
- clickable account # for drilling down transaction history
- account type
- current balance
Click on account# link
ShowTransactionHistory.jsp
Uses a custom tag library to print a table showing individual transactions for the selected account
Account
Account CMP entity bean
TMBTransactionHistory
JSP tag library
Accesses the TMBank database through a JDBC data source to print in tabular form the details for all transactions for a particular branch code and account no.

Fund Transfer

Figure A-6 shows how funds are transferred in the iBank application.

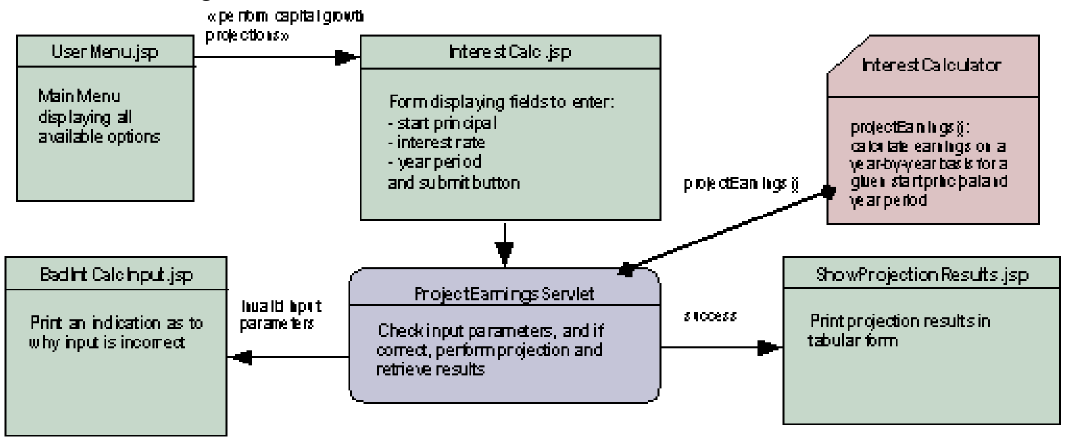
Figure A-6 Fund Transfer


« transfer funds between my accounts »
UserMenu.jsp
Main Menu displaying all available options
TransferFunds Servlet
Retrieve list of customer accounts
getAccountSummary()
BankTeller
getAccountSummary(): Builds list of accounts details for the current customer
transferFunds(): attempts to transfer funds between two accounts
transferFunds()
Return to main menu
TransferFunds.jsp
Transfer selection form with: - list to choose 'from' account - list to choose 'to' account - field to enter amount
CheckTransferServlet
Check transfer settings and proceed if OK
Call finder method: findOrderedAccountsForCustomer()
Account
Account CMP entity bean
success
input error or processing failure
TransferCheckFailed.jsp
Print an indication as to why fund transfer settings are incorrect, or why the operation failed
TransferSuccess.jsp
Print a confirmation message showing the details for the transaction that was successfully carried out

Interest Calculation

Figure A-7 shows how interest is calculated in the iBank application.

Figure A-7 Interest Calculation


«perform capital growth projections»
UserMenu.jsp
Main Menu displaying all available options
InterestCalc.jsp
Form displaying fields to enter:
- start principal
- interest rate
- year period
and submit button
Interest Calculator
projectEarnings():
calculate earnings on a year-by-year basis for a given start principal and year period
projectEarnings()
BadIntCalcInput.jsp
Print an indication as to why input is incorrect
invalid input parameters
ProjectEarningsServlet
Check input parameters, and if correct, perform projection and retrieve results
success
ShowProjectionResults.jsp
Print projection results in tabular form

Application Components

Data Components

Each table in the database schema is encapsulated as an entity bean:

Entity Bean	**Database Table**
Account	ACCOUNT table
AccountType	ACCOUNT_TYPE table
Branch	BRANCH table
Customer	CUSTOMER table
Transaction	TRANSACTION_HISTORY table
TransactionType	TRANSACTION_TYPE table

All entity beans use container-managed persistence (CMP), except `Customer`, which uses bean-managed persistence (BMP).

Currently, the application only makes use of the `Account`, `AccountType`, `Branch`, and `Customer` beans.

Business Components

Business components of the application are encapsulated by session beans.

The `BankTeller` bean is a stateful session bean that encapsulates all interaction between the customer and the system. `BankTeller` is notably in charge of the following activities:

- Authenticating a customer through the `authCheck()` method
- Giving the list of accounts for the customer through the `getAccountSummary()` method
- Transferring funds between accounts on behalf of the customer through the `transferFunds()` method

The `InterestCalculator` bean is a stateless session bean that encapsulates financial calculations. It is responsible for providing the compound interest projection calculations, through the `projectEarnings()` method.

Application Logic Components (Servlets)

Component name	**Purpose**
LoginServlet	Authenticates the user with the BankTeller session bean (`authCheck()` method), creates the HTTP session and saves information pertaining to the user in the session.Upon successful authentication, forwards request to the main menu page (`UserMenu.jsp`)
CustomerProfileServlet	Retrieves customer and branch details from the Customer and Branch entity beans and forwards request to the view/edit details page (`CustomerProfile.jsp`).
UpdateCustomerDetailsServlet	Attempts to effect customer details changes amended in `CustomerProfile.jsp` by updating the Customer entity bean after checking validity of changes. Redirects to `UpdatedDetails.jsp` if success, or to `DetailsUpdateFailed.jsp` in case of incorrect input.
ShowAccountSummaryServlet	Retrieves the list of customer accounts from the BankTeller session bean (`getAccountSummary()` method) and forwards request to `AccountSummary.jsp` for display.
TransferFundsServlet	Retrieves the list of customer accounts from the BankTeller session bean (`getAccountSummary()` method) and forwards request to `TransferFunds.jsp` allowing the user to set up the transfer operation.

CheckTransferServlet	Checks the validity of source and destination accounts selected by the user for transfer and the amount entered. Calls the `transferFunds()` method of the BankTeller session bean to perform the transfer operation. Redirects the user to `CheckTransferFailed.jsp` in case of input error or processing error, or to `TransferSuccess.jsp` if the operation was successfully carried out.
ProjectEarningsServlet	Retrieves the interest calculation parameters defined by the user in InterestCalc.jsp and calls the `projectEarnings()` method of the InterestCalculator stateless session bean to perform the calculation, and forwards results to the `ShowProjectionResults.jsp` page for display. In case of invalid input, redirects to `BadIntCalcInput.jsp`

Presentation Logic Components (JSP Pages)

Component name	**Purpose**
`index.jsp`	Index page to the application that also serves as the login page.
`LoginError.jsp`	Login error page displayed in case of invalid user credentials supplied. Prints an indication as to why login was unsuccessful.
`Header.jsp`	Page header that is dynamically included in every HTML page of the application
`CheckSession.jsp`	This page is statically included in every page in the application and serves to verify whether the user is logged in (i.e. has a valid HTTP session). If no valid session is active, the user is redirected to the `NotLoggedIn.jsp` page.
NotLoggedIn.jsp	Page that the user gets redirected to when they try to access an application page without having gone through the login process first.
UserMenu.jsp	Main application menu page that the user gets redirected to after successfully logging in. This page provides links to all available actions.
`CustomerProfile.jsp`	Page displaying editable customer details and static branch details. This page allows the customer to amend their correspondence address.
`UpdatedDetails.jsp`	Page where the user gets redirected to after successfully updating their details.
`DetailsUpdateFailed.jsp`	Page where the user gets redirected if an input error prevents their details to be updated.
`AccountSummaryPage.jsp`	This page displays the list of accounts belonging to the customer in tabular form listing the account no, account type and current balance. Clicking on an account no. in the table causes the application to present a detailed transaction history for the selected account.

`ShowTransactionHistory.jsp`	This page prints the detailed transaction history for a particular account no. The transaction history is printed using a custom tag library.
`TransferFunds.jsp`	This page allows the user to set up a transfer from one account to another for a specific amount of money.
`TransferCheckFailed.jsp`	When the user chooses incorrect settings for fund transfer, they get redirected to this page.
`TransferSuccess.jsp`	When the fund transfer set-up by the user can successfully be carried out, this page will be displayed, showing a confirmation message.
`InterestCalc.jsp`	This page allows the user to enter parameters for a compound interest calculation.
`BadIntCalcInput.jsp`	If the parameters for compound interest calculation are incorrect, the user gets redirected to this page.
`ShowProjectionResults.jsp`	When an interest calculation is successfully carried out, the user is redirected to this page that displays the projection results in tabular form.
`Logout.jsp`	Exit page of the application. This page removes the stateful session bean associated with the user and invalidates the HTTP session.
`Error.jsp`	In case of unexpected application error, the user will be redirected to this page that will print details about the exception that occurred.

Potential Migration Issues

While many of application design choices made are certainly debatable especially in the “real-world” context, care was taken to ensure that these choices enable the sample application to encompass as many potential issues as possible as one would face in the process of migrating a typical J2EE application.

This section will go through the potential issues that you might face when migrating a J2EE application, and the corresponding component of iBank that was included to check for this issue during the migration process.

With respect to the selected migration areas to address, this section specifically looks at the following technologies:

Servlets

The iBank application includes a number of servlets, that enable us to detect potential issues with:

- The use of generic functionality of the Servlet API
- Storage / retrieval of attributes in the HTTP session and HTTP request
- Retrieval of servlet context initialization parameters
- Page redirection

Java Server Pages

With respect to the JSP specification, the following aspects have been addressed:

- Use of JSP declarations, scriptlets, expressions, and comments
- Static includes (`<%@ include file="…" %>`): notably tested with the inclusion of the `CheckSession.jsp` file in every page)
- Dynamic includes (`<jsp:include page=… />`): this is catered for by the dynamic inclusion of `Header.jsp` in every page
- Use of custom tag libraries: a custom tag library is used in the file `ShowTransactionHistory.jsp`
- Error pages for JSP exception handling: the `Error.jsp` page is the application error redirection page

JDBC

The iBank application accesses a database via a connection pool and the data source, both programmatically (BMP entity bean, BankTeller session bean, custom tag library) and declaratively (with the CMP entity beans).

Enterprise Java Beans

The iBank application uses a variety of Enterprise Java Beans.

Entity Beans

Bean-managed persistence (`Customer` bean): allows us to test the following:

- JNDI lookup of initial context
- Pooled data source access via JDBC

- Definition of a BMP custom finder ("`findByCustUsername()`")

Container-managed persistence ("`Account`" and "`Branch`" beans): allow us to test the following:

- Object/Relational mapping with the development tool and within the deployment descriptor
- Use of composite primary keys (`Account`)
- Definition of custom CMP finders (with the "`Account`" bean, and its `findOrderedAccountsForCustomer()` method). This is the occasion to look at differences in declaring the query logic in the deployment descriptor, and also to have a complex example returning a collection of objects.

Session Beans

Stateless session beans: `InterestCalculator` allows us to test the following:

- Using and deploying a stateless session bean
- Calling a business method for calculations

Stateful session beans: `BankTeller` allows us to test the following:

- Looking up various interfaces using JNDI and initial contexts
- Using JDBC to perform database queries
- Using various transactional attributes on bean methods
- Using container-demarcated transactions
- Maintaining conversational state between calls
- Business methods acting as front-ends to entity beans (e.g., the "`getAccountSummary()`" method)

Application Assembly

The iBank application is assembled by following the J2EE standard procedures. It contains the following components:

- A Web application archive file for the Web application module, and EJB-JAR archives for the EJBs
- An enterprise application archive file (EAR file) for the final packaging of the Web application and EJB modules

Index