Oracle9*i* Application Server

Migrating From WebSphere

Release 2 (9.0.2)

May 2002

Part No. A95110-02

ORACLE®

Oracle9*i* Application Server Migrating From WebSphere, Release 2 (9.0.2)

Part No. A95110-02

Contents

2 Comparison of Oracle9*i*AS and WebSphere Features

3   Migrating Servlets

4 Migrating JSPs

5 Migrating Enterprise Java Beans

6 Migrating JDBC Applications

A J2EE Server Comparison

B Migrating from WebSphere 4.0

C   Partner Migration Tools

Index

x

Send Us Your Comments

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most?

If you find any errors or have any other suggestions for improvement, please indicate the document title and part number, and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: infodev_us@oracle.com
- FAX: (650) 506-7227   Attn: Server Technologies Documentation Manager
- Postal service:
  Oracle Corporation
  Server Technologies Documentation
  500 Oracle Parkway, Mailstop 4op11
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

Oracle9*i* Application Server Migrating From WebSphere provides you with the information required for a successful migration from IBM's application server, WebSphere 3.5, to Oracle Corporation's application server, Oracle9*i* Application Server (Oracle9*i*AS).

This preface contains these topics:

- Audience
- Organization
- Related Documentation
- Conventions
- Documentation Accessibility

Audience

Oracle9i Application Server Migrating From WebSphere is intended for those Java
development managers, application developers, and system administrators
responsible for planning and migrating J2EE web applications from WebSphere 3.5
to Oracle9*i* Application Server (Oracle9*i*AS).

To use this document, you need an in-depth understanding of the Java platform and
experience in Java application development, configuration, and deployment. In
addition, you need a thorough understanding of the WebSphere Application Server
environment, as well as the Oracle9*i*AS environment.

Organization

This document contains:

Chapter 1, "Overview"

This chapter provides you with an overview of the J2EE platform, application
servers, and Oracle9*i*AS. In addition, it provides you with an understanding of
what is involved in migrating from WebSphere 3.5 to Oracle9*i*AS.

Chapter 2, "Comparison of Oracle9iAS and WebSphere Features"

This chapter provides a comparison between Oracle Corporation's implementation
of Sun Microsystems' J2EE platform and component specifications and that of
IBM's.

Chapter 3, "Migrating Servlets"

This chapter provides you with an overview of Sun Microsystems' Java Servlet
technology and its implementation in Oracle9*i*AS. In addition, the issues involved
in migrating servlets from WebSphere 3.5 to Oracle9*i*AS are presented.

Chapter 4, "Migrating JSPs"

This chapter provides you with an overview of Sun Microsystems' JavaServer Pages
(JSP) technology and its implementation in Oracle9*i*AS. In addition, the issues
involved in migrating JSP pages from WebSphere 3.5 to Oracle9*i*AS are presented.

Chapter 5, "Migrating Enterprise Java Beans"

This chapter provides you with an overview of Sun Microsystems' Enterprise
JavaBeans (EJB) architecture and its implementation in Oracle9*i*AS. In addition, the

issues involved in migrating EJB components from WebSphere 3.5 to Oracle9*i*AS are presented.

Chapter 6, "Migrating JDBC Applications"

This chapter provides you with an overview of Sun Microsystems' JDBC technology and its implementation in Oracle9*i*AS. In addition, the issues involved in migrating database access code from WebSphere 3.5 to Oracle9*i*AS are presented.

Appendix A, "J2EE Server Comparison"

This chapter compares WebSphere 3.5 to Oracle9*i*AS as a J2EE server in terms of installation and configuration, proprietary extensions, database connections, security, and performance and scalability.

Appendix B, "Migrating from WebSphere 4.0"

This chapter provides the migration strategy and tips for migrating applications from WebSphere Advanced Edition 4.0 to Oracle9*i*AS.

Appendix C, "Partner Migration Tools"

This appendix provides an overview of the tools available from Oracle9*i*AS partners which aid the migration process.

Related Documentation

For more information, see these Oracle resources:

- *Oracle9i Application Server Concepts Guide*
- *Oracle9i Application Server Developer's Guide*
- *Oracle9iAS Containers for J2EE User's Guide*
- *Oracle9iAS Containers for J2EE Servlet Developer's Guide*
- *Oracle9iAS Containers for J2EE Support for JavaServer Pages Reference*
- *Oracle9iAS Containers for J2EE Enterprise JavaBeans Developer's Guide and Reference*
- *Oracle9i JDBC Developer's Guide and Reference*

In North America, printed documentation is available for sale in the Oracle Store at

```
http://oraclestore.oracle.com/
```

Customers in Europe, the Middle East, and Africa (EMEA) can purchase documentation from

```
http://www.oraclebookshop.com/
```

Other customers can contact their Oracle representative to purchase printed documentation.

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

```
http://otn.oracle.com/membership/index.htm
```

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

```
http://otn.oracle.com/docs/index.htm
```

For additional information, see:

- `http://ibm.com/` for more information on WebSphere Application Server

- `http://java.sun.com/` for more information on J2EE

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- Conventions in Text

- Conventions in Code Examples

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an **index-organized table**.

Convention	Meaning	Example
Italics	Italic typeface indicates book titles or emphasis.	*Oracle9i Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.

`UPPERCASE`
`monospace`
`(fixed-width`
`font)`
	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a `NUMBER` column.
		
		You can back up the database by using the `BACKUP` command.
		
		Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view.
		
		Use the `DBMS_STATS.GENERATE_STATS` procedure.

`lowercase`
`monospace`
`(fixed-width`
`font)`
	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values.
	
	Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		Enter `sqlplus` to open SQL*Plus.
		
		The password is specified in the `orapwd` file.
		
		Back up the datafiles and control files in the `/disk1/oracle/dbs` directory.
		
		The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table.
		
		Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true`.
		
		Connect as `oe` user.
		
		The `JRepUtil` class implements these methods.

`lowercase`
`monospace`
`(fixed-width`
`font) italic`
	Lowercase monospace italic font represents placeholders or variables.	You can specify the `parallel_clause`.
		
		Run U`old_release`.SQL where `old_release` refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
[]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (digits [ , precision ])`
{}	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE	DISABLE}`
|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE	DISABLE}` `[COMPRESS	NOCOMPRESS]`
...	Horizontal ellipsis points indicate either:	
	■ That we have omitted parts of the code that are not directly related to the example	`CREATE TABLE ... AS subquery;`
	■ That you can repeat a portion of the code	`SELECT col1, col2, ... , coln FROM employees;`
. . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	
Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);` `acct    CONSTANT NUMBER(4) := 3;`
Italics	Italicized text indicates placeholders or variables for which you must supply particular values.	`CONNECT SYSTEM/system_password` `DB_NAME = database_name`

UPPERCASE	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

lowercase	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.
	
	Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		`SELECT last_name, employee_id FROM employees;`
		`sqlplus hr/hr`
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle Corporation is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at

```
http://www.oracle.com/accessibility/
```

Accessibility of Code Examples in Documentation   JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation   This documentation may contain links to Web sites of other companies or organizations that Oracle Corporation does not own or control. Oracle Corporation neither evaluates nor makes any representations regarding the accessibility of these Web sites.

xx

1

Overview

This chapter provides you with an overview of the J2EE platform, application servers, and the Oracle9*i* Application Server (Oracle9*i*AS). In addition, it provides you with an understanding of what is involved in migrating from WebSphere Advanced Edition 3.5.3 to Oracle9*i*AS.

The chapter contains these topics:

- Overview of J2EE
- What is an Application Server?
- Overview of Oracle9iAS
- J2EE Application Architecture
- Migration Issues
- Using this Guide

Overview of J2EE

The application server market is evolving rapidly. In particular, the most significant development over the last few years is the emergence of Sun Microsystems' Java 2 Platform, Enterprise Edition (J2EE) Specification that promises to create a level of cross-vendor standardization.

The J2EE platform and component specifications define, among other things, a standard platform for developing and deploying multi-tier, web-based enterprise applications.

J2EE provides a solution to the problems encountered by companies moving to a multi-tier computing model. The problems addressed include reliability, scalability, security, application deployment, transaction processing, web interface design, and timely software development. It builds upon the Java 2 Platform, Standard Edition (J2SE) to enable Sun Microsystems' "Write Once, Run Anywhere" paradigm for multi-tier computing.

J2EE consists of the components described in Table 1–1:

Table 1–1   J2EE Standard Architecture Components

Component	Description
J2EE Application Model	An application model for developing multi-tier, thin client services
J2EE Platform	A platform for hosting J2EE applications
J2EE Compatibility Test Suite	A compatibility test suite for verifying that a J2EE platform product meets the requirements set forth in the J2EE platform and component specifications
J2EE Reference Implementation	A reference implementation of the J2EE platform

What is the J2EE Application Model?

The J2EE application model is a multi-tier application model. Application components are managed in the middle tier by containers. A container is a standard runtime environment that provides services, including life cycle management, deployment, and security services, to application components. This container-based model separates business logic from system infrastructure.

What is the J2EE Platform?

The J2EE platform consists of a runtime environment and a standard set of services that provide the necessary functionality for developing multi-tiered, web-based enterprise applications.

The J2EE platform consists of the components described in Table 1–2.

Table 1–2   J2EE Platform Components

Component	Description
J2EE runtime application components	
Application clients	A Java program, typically used for a GUI, that executes on a desktop computer
Applets	A component of a Java program that typically executes in a web browser
Servlets	A Java program, used to generate dynamic content, that executes on a web server
JSPs	A technology used to return dynamic content to a client, typically a web browser
Enterprise JavaBeans (EJB)	An applications architecture for component-based distributed computing
Containers	An entity that provides services for application components, including life cycle management, deployment, and security services
Resource manager drivers	A system-level component that enables network connectivity to external data sources
Database	A set of related files used for the storage of business data and accessible through the JDBC API
J2EE standard services	
HTTP	The standard protocol used by the Internet to send and receive messages between web servers and browsers

Table 1–2   J2EE Platform Components

Component	Description
HTTPS	A protocol used by the Internet to send and receive messages securely between web servers and browsers
Java Transaction API (JTA)	An API that allows applications and application servers to access transactions
RMI-IIOP	RMI: A protocol that enables Java objects to communicate remotely with other Java objects.
	IIOP: A protocol that enables browsers and servers to exchange things other than text.
	RMI-IIOP is a version of RMI that uses the CORBA IIOP protocol
JavaIDL	A standard language for interface specification primarily used for CORBA object interface definition
JDBC	An API that provides connectivity between databases and the J2EE platform
Java Message Service (JMS)	An API that enables the use of enterprise messaging systems
Java Naming and Directory Interface (JNDI)	An API that provides directory and naming services
JavaMail	An API that provides the ability to send and receive e-mail
JavaBeans Activation Framework (JAF)	An API required by the JavaMail API

What is an Application Server?

An application server is software that runs between web-based client programs and back-end databases and legacy applications. They help separate system complexity from business logic, enabling developers to focus on solving business problems. They help reduce the size and complexity of client programs by enabling these programs to share capabilities and resources in an organized and efficient way.

Application servers provide benefits in the areas of usability, flexibility, scalability, maintainability, and interoperability.

Overview of Oracle9*i*AS

Oracle9*i*AS is a comprehensive, integrated application server that provides all of the infrastructure and functionality needed to run every successful e-Business. All development teams face a similar set of challenges—the need to rapidly deliver web sites and applications that run fast over any network and on every device; while providing business intelligence to support operational adjustments and strategic decisions. Oracle9*i*AS enables teams to address all of these e-business challenges.

Oracle9*i*AS has generated a great deal of interest in the application server market, and many organizations are embracing it to deploy their web-based enterprise applications.

Oracle9*i*AS offers the only integrated infrastructure to develop and deploy web sites and applications. It provides a complete J2EE platform for developing enterprise Java applications. It enables developers to develop web applications in any language including Java, Perl, PL/SQL, XML, and Oracle Forms. It enables the reduction of development and deployment costs through a single, unified platform for Java, XML, and SQL.

The J2EE server implementation in Oracle9*i*AS is called or OC4J. OC4J is J2EE 1.2 compliant and runs on the standard JDK. It is lightweight, provides high performance and scalability, and is simple to deploy and manage. OC4J can be deployed in standalone mode, which is ideal for development environments or with Oracle Enterprise Manager (OEM) to provide enterprise-level monitoring and management facilities.

J2EE Application Migration Challenges

The varying degrees of compliance to J2EE standards can make migrating applications from one application server to another a daunting task. Some of the challenges in migrating J2EE applications from one application server to another are:

- Though in theory any J2EE application can be deployed on any J2EE-compliant application server, in practice this is not strictly true.

- Lack of knowledge of the implementation details of the given J2EE application.

- Ambiguity in the meaning of 'J2EE-compliant' (usually, this means the application server has J2EE compliant features, not code-level compatibility with the J2EE specification).

- The number of vendor-supplied extensions to the J2EE standards in use, which differ in deployment methods and reduce portability of Java code from one

application server to another. (For example, there are Websphere-specific libraries associated with servlet engines, EJB containers, and JDBC and JNDI interfaces).

- Differences in clustering, load balancing, and failover implementations among application servers. These differences are sparsely documented, and are thus an even bigger challenge to the migration process.

These challenges make the migration path daunting, uncertain, and difficult to reliably plan and schedule. This chapter addresses the challenges in migrating your applications from WebSphere to Oracle9*i* Application Server, providing an approach to migration with solutions based on the J2EE version 1.2 specification.

J2EE Application Architecture

The J2EE platform provides a multi-tiered distributed application model. Central to the J2EE component-based development model is the notion of containers. Containers are standardized runtime environments that provide specific services to components. Thus, Enterprise Java Beans (EJB) developed for a specific purpose in any organization can expect generic services such as transaction and EJB life cycle management to be available on any J2EE platform from any vendor.

Containers also provide standardized access to enterprise information systems; for example, providing RDBMS access through the JDBC API. Containers also provide a mechanism for selecting application behavior at assembly or deployment time.

As shown in Figure 1–1, the J2EE application architecture is a multi-tiered application model. In the middle tier, components are managed by containers: For example, J2EE Web containers invoke servlet behavior, and EJB containers manage life cycle and transactions for EJBs. The container-based model separates business logic from system infrastructure.

Figure 1–1   J2EE Architecture
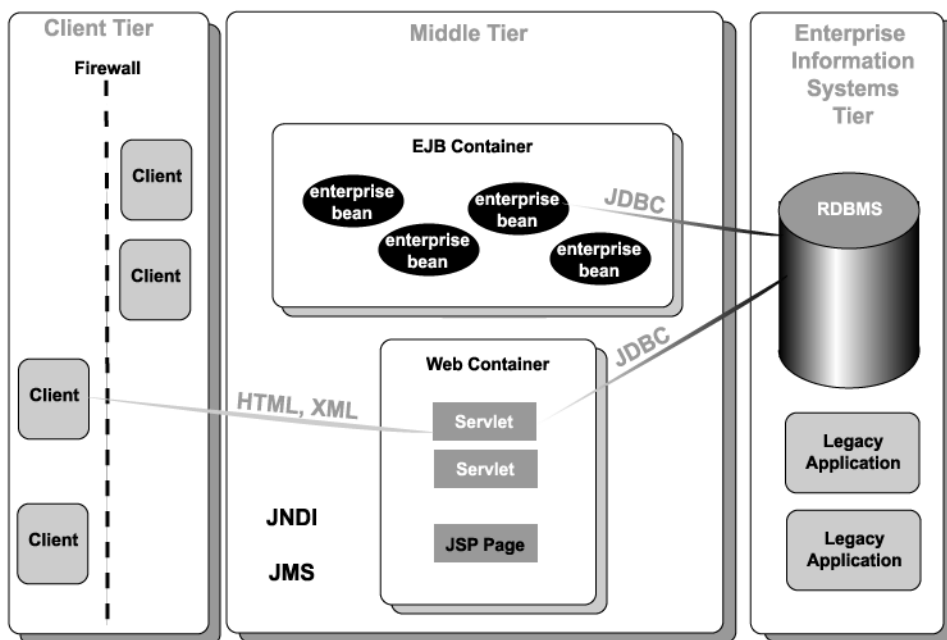
Migration Issues

An issue is a condition that could cause unplanned work. Given the inherent challenges outlined above, it is helpful to examine the applications to be migrated in terms of the following before quantifying the migration effort:

- Portability

- Dependence on vendor-specific APIs

- Adherence to the J2EE specification

Portability

Code may not be portable because it contains embedded references to vendor-specific extensions to the J2EE specification. In such cases, runtime exceptions, (for example, "class not found") may occur when applications are migrated and run from one J2EE-compliant application server to another. In addition, some J2EE

application servers still support deprecated APIs and others are strictly compliant to the J2EE specifications. WebSphere contains extenstions to servlets, JSPs, EJBs, JNDI, and JDBC. In such cases, evaluating code and planning for its modification may be a significant part of the migration effort.

Vendor Specificity

If WebSphere-specific services are in use, migration of those components becomes difficult or unfeasible. These components may need to be redesigned and reimplemented,instead of being identified as migration candidates. This guide does not address complete redesign toward J2EE specifications. For example, applications using Component Broker (IBM ORB) services, CICS or Encina transaction monitors, MQSeries or DB2 libraries are not candidates for migration as defined in this guide.

Deviations from J2EE 1.2 Specification

Different application server vendors have different levels of support for J2EE standards, and some variations in behavior. For example, WebSphere Advanced Edition 3.5.3 is not fully compliant with the J2EE 1.2 specifications, but Oracle9*i* Application Server is. This fact alone raises issues with EJBs, JNDI, and security migration. This guide addresses those issues and explains how to migrate to Oracle9*i* Application Server without major code changes.

Migration Approach

Our approach for this guide is to document our experiences with migrating components and/or example applications from WebSphere Advanced Edition 3.5.3 to Oracle9*i* Application Server Release 2. Guidelines for migrating from WebSphere Advanced Edition 4.0 to Oracle9*i* Application Server Release 2 is discussed in Appendix B, "Migrating from WebSphere 4.0".

We selected some of the examples shipped with WebSphere for this migration excercise.We tested these samples with WebSphere and migrated them to Oracle9*i* Application Server. In doing so, we exposed and documented specific migration issues not identified in the product documentation. As described in "J2EE Application Migration Challenges" these issues exist because WebSphere Advanced Edition 3.5.3 does not comply to the J2EE 1.2 specification, and because WebSphere-specific API extensions are used.

Using this Guide

This guide details the migration of components from WebSphere Advanced Edition 3.5.3 to Oracle9*i* Application Server. While it does not claim to be an exhaustive source of solutions for every possible configuration, it provides solutions for some of the migration issues listed above, which will surface, along with others, in your migration effort. The information in this guide helps you to assess the WebSphere Advanced Edition 3.5.3 applications and plan and execute their migration to Oracle9*i* Application Server. The material in this guide supports these high-level tasks:

- Survey the components according to the issues listed above

- Identify migration candidates

- Prepare the migration environment and tools

- Migrate and test the candidate components

> **Note:** In this document, unless otherwise specified, any reference to WebSphere without a version number implies reference to WebSphere Advanced Edition 3.5.3.

2

Comparison of Oracle9*i*AS and WebSphere Features

WebSphere and Oracle9*i*AS are created from entirely different architectures. WebSphere is based on the IBM SanFrancisco Java application framework and its Component Broker, both of which predate J2EE standards. Oracle9*i*AS has a new lightweight, robust J2EE container that supports the latest J2EE 1.2 standard.

This chapter identifies major differences between WebSphere and Oracle9*i*AS in terms of overall product offering, architecture, clustering and load balancing, J2EE support, and development and deployment tools.

Application Server Product Offerings Comparison

This section describes WebSphere and Oracle9*i*AS application server products.

WebSphere Product Offerings

IBM sells a several technologies under the WebSphere marketing umbrella. The WebSphere Application Server is the core of the WebSphere extended family of products, of which there are three versions, described below.

WebSphere Standard Edition

WebSphere Standard Edition is a Servlet/JSP container layer that runs on top of an HTTP server. It works with a number of popular HTTP servers, including IBM HTTP Server, Microsoft IIS, and Netscape iPlanet server. WebSphere Standard Edition supports static HTML pages, servlets, JavaServer Pages, and XML .

WebSphere Advanced Edition

WebSphere Advanced Edition contains all the features of Standard Edition, and also includes:

- Full support for the Enterprise JavaBeans™ (EJB) component model
- Workload management (WLM) features to support multiple servers within a single administrative domain

WebSphere Advanced Edition is IBM's core J2EE server.

WebSphere Enterprise Edition

WebSphere Enterprise Edition includes all of the features in Advanced Edition, and also includes:

- Component Broker (CB), the IBM Object Request Broker
- TXSeries, the IBM transactional middleware solution with two transaction processing monitors: CICS and Encina

In WebSphere Enterprise Edition, the Component Broker serves both EJBs and CORBA objects. TXSeries provides a a pure transactional environment, for applications that don't require an EJB/component-based/object-oriented programming model. Depending on your requirements, you could use either or both.

Oracle9*i* Application Server

Like WebSphere, Oracle9*i*AS is a platform-independent J2EE application server that can host multi-tier, web-enabled enterprise applications for the Internet and intranets, and which is accessible from browser and standalone clients. It includes Oracle9*i*AS Containers for J2EE (OC4J) a lightweight, scalable J2EE container written in Java, and is J2EE 1.2 certified. In addition, OC4J provides support for J2EE 1.3 features such as:

- Servlets 2.3
- JSP 1.2
- EJB 2.0
- JNDI 1.2
- JavaMail 1.2

- JAF 1.0

- JAXP 1.1

- JCA 1.0

- JAAS 1.0

- JMS 1.0

- JTA 1.0

- JDBC 2.0

Oracle9*i*AS is designed specifically for running large-scale, distributed Java enterprise applications, including Internet commerce sites, enterprise portals and high volume transactional applications. It adds considerable value beyond the J2EE standards in areas critical to the implementation of real world applications, providing an entire suite of integrated solutions that encompass:

- Web services

- Business intelligence

- Management and security

- E-business integration

- Support for wireless clients

- Enterprise portals

- Performance caching

To enable these solutions to be implemented in a reliable and scalable infrastructure, Oracle9*i*AS can be deployed in a redundant architecture using clustering mechanisms. The sections "Architecture Comparison" and "Oracle9iAS Support for Clustering and Load Balancing" in this chapter details the components in and characteristics of Oracle9*i*AS.

Architecture Comparison

This section describes and compares the overall architectures of WebSphere and Oracle9*i*AS.

IBM WebSphere Components

The WebSphere Advanced Edition 3.5.3 consists of the following components:

IBM HTTP Server

IBM's HTTP Server is the Apache HTTP Server (with official product support) with SSL capability implemented by IBM, and IBM tools for managing keys, certificates, and such. The public key technology that SSL uses is patented and requires tracking for licensing purposes. The SSL support is part of IBM's value-add on top of the Apache HTTP Server open source distribution. Neither Apache nor the IBM HTTP Server provide servlet support out-of-the-box.

Web Server Plug-in

The Web Server Plug-in is a module that runs within the web server, using its native APIs, and forwards requests to the WebSphere Application Server. When you install WebSphere, the installation program installs a hook into the web server that intercepts HTTP requests that target a servlet (it examines the incoming URL to determine whether it is a servlet request), and redirects those requests to the servlet engine for processing. Static content is still handled solely by the HTTP Server.

Administrative Server

The Adminsitrative Server must be running on every node that is running a WebSphere Application Server component. It performs the following functions:

- Starting, stopping and monitoring all configured application servers.

- Providing a location service daemon (LSD).

- Providing a persistence name server (PNS).

- Providing a security server.

- Providing a watchdog process to restart the Administrative Server in case of failure.

Figure 2–1   Runtime components of WebSphere Application Server Advanced Edition
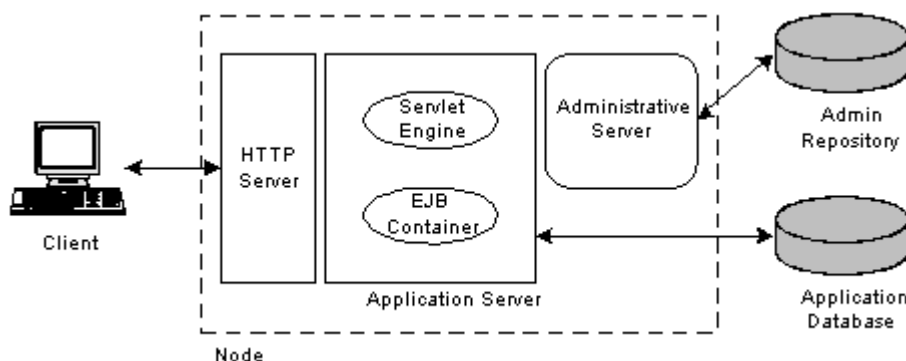
Administrative Repository

The WebSphere Application Server version 3.5.x requires an Administrative Server Repository. The Administrative Server Repository is a relational database containing configuration information. This database is used to store setup, configuration, and state information about the WebSphere Application Server.

Before starting the Administrative Server, WebSphere Application Server checks for the existence of an Administrative Server Repository, which contains descriptive information about the resources that are configured to run on each node in the domain, for example, the names of application servers, the node each server is running on, the enterprise beans installed in each server, and the current state of each server.

The Administrative Server Repository enables the system administrator to manage the domain from any machine, because all configuration information is stored in a central location. Each Administrative Server has a central view of resource configuration information about in the domain. When the administrator modifies a resource configuration, the changes are seen by all administrative servers.

Application Server

In WebSphere, an Application Server is the process that runs servlet and/or EJB-based applications, providing both the servlet run-time components (Servlet Engine, Web applications) and EJB run-time (EJB container). Like the Administrative Server, each WebSphere Application Server runs in its own Java Virtual Machine (JVM).

Oracle9*i*AS Components and Concepts

This section describes components and several concepts peculiar to Oracle9*i*AS. The discussion here provides an overview scope. For more detailed information, refer to the *Oracle9i Application Server Concepts Guide*, *Oracle9i Application Server Administrator's Guide*, and *Oracle9iAS Containers for J2EE User's Guide*.

Oracle9*i*AS Instance

An Oracle9*i*AS instance is a runtime occurrence of an installation of Oracle9*i*AS. An Oracle9*i*AS installation corresponds to an "Oracle home" where the Oracle9*i*AS files are installed. Each Oracle9*i*AS installation can provide only one Oracle9*i*AS instance at runtime. A physical node can have multiple "Oracle homes", and hence, more than one Oracle9*i*AS installation and Oracle9*i*AS instance.

Each Oracle9*i*AS instance consists of several interoperating components that enable Oracle9*i*AS to service user requests in a reliable and scalable manner. These components are Oracle HTTP Server, OC4J instances, Oracle Process Management Notification (OPMN) service, and Distributed Configuration Manager (DCM).

Oracle HTTP Server

Oracle9*i*AS contains two listeners: The Oracle HTTP Server (based on the Apache open source product) and the listener that is part of OC4J, which runs in a separate thread of execution. Each Oracle9*i*AS instance has one Oracle HTTP Server.

The OC4J listener listens to requests coming from the `mod_oc4j` module of the Oracle HTTP Server and forwards them to the appropriate OC4J instance. From a functional viewpoint, the Oracle HTTP Server acts as a proxy server to OC4J, wherein all servlet or JSP requests are redirected to OC4J instances.

`mod_oc4j` communicates with the OC4J listener using the Apache JServ Protocol version 1.3 (AJP 1.3). This protocol load balances JSP and servlet requests between OC4J instances. `mod_oc4j` works with the Oracle HTTP Server as an Apache module. The OC4J listener can also accept HTTP and RMI requests, in addition to AJP 1.3 requests.

The following diagram depicts the Oracle HTTP Server and other Oracle9*i*AS runtime components in a single instance of Oracle9*i*AS.

Figure 2–2   Components of an Oracle9iAS instance
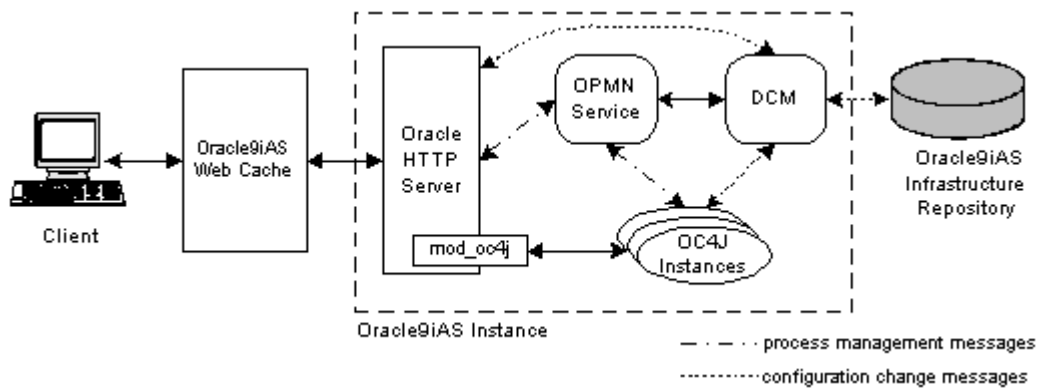
OC4J Instances

An OC4J instance is a logical instantiation of the OC4J implementation in Oracle9*i*AS. This implementation is Java 2 Enterprise Edition (J2EE) complete and written entirely in Java. It executes on the standard Java Development Kit (JDK) 1.3.1 Java Virtual Machine. It has a lower disk and memory footprint than the previous Oracle9*i*AS Java environment and competitive Java application servers. Note that each OC4J instance can consist of more than one JVM process where each process can be executing multiple J2EE containers. The number of JVM processes can be specified for each OC4J instance using the Oracle Enterprise Manager GUI.

Oracle9*i*AS allows several OC4J instances to be clustered together for scalability and high-availibility purposes. When OC4J instances are clustered together, they have the same configuration and applications deployed amongst them. A more in-depth discussion on clustering is found in the section "Oracle9iAS Support for Clustering and Load Balancing" below.

Oracle Process Management Notification (OPMN) Service

Each Oracle9*i*AS instance has an OPMN service which performs monitoring and process management functions within that instance. This service communicates messages between the components in an Oracle9*i*AS instance to enable startup, death-detection and recovery of components. This communication extends to other OPMN services in other Oracle9*i*AS instances belonging to the same cluster as well, thereby allowing other instances in a cluster to be aware of active OC4J and Oracle HTTP Server processes in other Oracle9*i*AS instances (in the same cluster).

The OPMN service also communicates and interfaces with Oracle Enterprise Manager to provide a consolidated interface for monitoring, configurating, and managing Oracle9*i*AS. Oracle9*i*AS components, Oracle HTTP Server, OC4J instances, and Distributed Configuration Manager (described below), use a subscribe-publish messaging mechanism to communicate with the OPMN service. For failover and availibility, the process that implements the OPMN service has a shadow process that restarts the OPMN process if it fails.

Distributed Configuration Manager (DCM)

In order to manage and track configuration changes in the various components in each Oracle9*i*AS instance, a DCM process exists in each Oracle9*i*AS instance to perform those tasks. Each configuration change made to any of the components in a Oracle9*i*AS instance is communicated to the DCM. DCM in turn takes note of the change and records it in the Oracle9*i*AS metadata repository in the infrastructure database. This repository contains the configuration information for all the Oracle9*i*AS instances connected to it through their respective DCMs. All Oracle9*i*AS instances connecting to the same infrastructure repository in this way belong to the same Oracle9*i*AS farm. If any of the Oracle9*i*AS instances fail, the configuration information can be retrieved from the repository for purposes of restarting the instance.

Each DCM also communicates with the OPMN in their respective instances to send notification events on changes in repository data. This allows OPMN to make the corresponding adjustments to the Oracle9*i*AS components.

Oracle9*i*AS Infrastructure Repository

The Oracle9*i*AS infrastructure repository maintains metadata about the Oracle9*i*AS clusters and standalone Oracle9*i*AS instances connected to it. A common and shared infrastructure repository ensures a more robust way of maintaining configuration information and consistency between the clusters and instances. Whenever a new instance is added to a cluster, common configuration information such as the applications deployed is retrieved from the infrastructure repository and propagated to the new instance so that it will behave uniformly with the other instances of the cluster. The infrastructure repository is discussed further in the section "Oracle9iAS Clusters" below.

Oracle9*i*AS Web Cache

Oracle9*i*AS provides a caching solution with the unique capability to cache both static and dynamically generated web content. The Oracle9*i*AS Web Cache significantly improves the performance and scalability of heavily loaded

Oracle9*i*AS web sites by reducing the number of round trips to the web server. In addition, it provides a number of features to ensure consistent and predictable responses. These features include page fragment caching, dynamic content assembly, web server load balancing, Web Cache clustering, and failover. Oracle9*i*AS Web Cache can be used as a load balancer for Oracle9*i*AS instances in a cluster. Web Cache can itself be deployed in its own cluster. Refer to the *Oracle9iAS Database Cache Concepts and Administration Guide.*

Clustering and Load balancing

This section defines and describes clustering and load balancing and their importance to application server operation, It compares the methods for clustering and load balancing used in WebSphere and Oracle9*i*AS.

What is Clustering?

An application server cluster is a group of independent application server instances managed as a single system for higher availability and increased scalability. The main goal of clustering is to minimize response time to user requests and to provide scalability (the ability to add nodes to an existing system with minimal system disruption). Clustering improves manageability, since the system administrator can remotely manage the cluster from a central location. The cluster appears to the administrator as a single system.

Benefits of Clustering: Failover Recovery

Within a cluster of multiple application server instances, a failed application server instance can rely on another instance to take over its workload. Two important characteristics of failover are quick failure detection, and failover without loss of data. The level of failover support varies among application servers. Oracle9*i*AS provides support for both.

What is Load Balancing?

Load balancing is the proportional distribution of client requests (the application server workload) among the servers in the cluster, enabling the maximum number of concurrent requests. The primary goals of load balancing are to optimize usage of available server capacity and provide the most rapid possible response time to clients.

WebSphere Suppport for Clustering and Load Balancing

WebSphere provides clustering and load balancing support through its Administrative Console, with cloning and workload management services.

Clustering in WebSphere

Clustering is implemented in WebSphere with a mechanism called cloning, available in the Administration system. Cloning enables you to create multiple copies of an application server, based on a server that you have already configured.

The clone has the same structure and attributes as the application server on which it is based, but it is not associated with any node, and does not correspond to any real server process running on any node.

WebSphere Application Server supports cloning for servlet engines, Web applications, and servlets for workload management, load balancing, and failover. The servlets, EJBs, and Web resources are shared by the clones, but each clone uses its own JVM to run the application code. This provides identical, yet independent processes for the application to run in.

Load Balancing in WebSphere

The workload management service improves the scalability of the application server environment by grouping multiple application servers into application server groups. Clients then access these application server groups as if they were a single server, and the workload management service distributes the workload among the application servers in the application server groups. An application server can belong to only one application server group. WebSphere workload management supports load balancing for stateless servlets and stateless session beans, and provides a failover mechanism for stateful servlets and stateful session beans.

Servlet load balancing is performed by a servlet redirector. The servlet redirector runs on the Web server in front of the application servers. The redirector balances workload across the servlet engines running in multiple application servers behind the Web server. When a web server HTTP session asks to invoke a servlet, the redirector transfers the request to a servlet engine.

The EJB component workload manager balances the load between Java objects (servlets to EJB components, EJB components to EJB components and stand-alone Java clients to EJB components). For example, when a servlet needs data or begins a transaction through an EJB component, the EJB component workload manager transfers the request to an EJB container (an instance of WebSphere Application Server) or a remote EJB handler.

Oracle9*i*AS Support for Clustering and Load Balancing

Oracle9*i*AS is designed with sophisticated clustering mechanisms. These mechanisms ensure that failover and scalibility are achieved at the infrastructure and application levels. This section describes the clustering and load balancing concepts and capabilities of Oracle9*i*AS and OC4J.

Oracle9*i*AS Clusters

An Oracle9*i*AS cluster is made up of one or more Oracle9*i*AS instances (see Figure 2–3). All Oracle9*i*AS instances in the cluster have the same configuration. The first Oracle9*i*AS instance to join a cluster has its configuration replicated to the second and later instances when they join. In addition to the configuration, deployed OC4J applications are also replicated to the newer instances. Information for the replicated configuration and applications is retrieved from the Oracle9*i*AS infrastructure repository used by the cluster.

Within each cluster, there is no mechanism to load balance or failover the Oracle9*i*AS instances. That is, there is no internal mechanism in the cluster to load balance or failover requests to the Oracle HTTP Server component in the instances. A separate load balancer such as Oracle9*i*AS Web Cache or hardware load balancing product can be used to load balance the Oracle9*i*AS cluster and failover the Oracle HTTP Server instances in the cluster.

Several Oracle9*i*AS clusters and standalone Oracle9*i*AS instances can be further grouped into an Oracle9*i*AS farm. The clusters and instances in this farm share the same Oracle9*i*AS infrastructure repository. For further information on Oracle9*i*AS farms, refer to the *Oracle9i Application Server Administrator's Guide*.

Figure 2–3   An Oracle9iAS cluster using Oracle9iAS Web Cache for load balancing
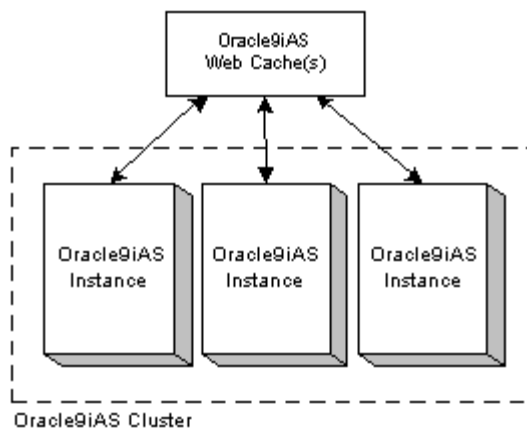
OC4J Islands

An important function of clustering technology in Oracle9*i*AS is that of reducing multicast traffic. With every server sharing its session state with every other server in the cluster, a lot of CPU cycles is consumed as overhead to replicate the session state across all nodes in the cluster. Oracle9*i*AS solves this problem by introducing the concept of OC4J islands, where OC4J processes (JVMs) in an Oracle9*i*AS cluster can be sub-grouped into islands. Session state of applications is replicated only to OC4J processes belonging to the same island rather than all OC4J processes in the Oracle9*i*AS cluster. Hence, state is replicated to a smaller number of processes. OC4J islands are typically configured to span across physical nodes, thereby allowing failover of application state if a node goes down.

Consider an Oracle9*i*AS cluster with four OC4J processes running in two nodes, two processes per node (see Figure 2–4). When the state of an application changes, which could occur at every request from the same client, multicast messages are sent between all four processes to update the state of that application in each process. If these four processes were to be divided into two islands of two processes across two nodes, state replication of the application would only have to occur between processess within the same island. Multicast messages would be required only between the two processes in the island instead of four, reducing replication overhead by half. As a result, network traffic and CPU cycles are reduced.

Figure 2–4 OC4J islands
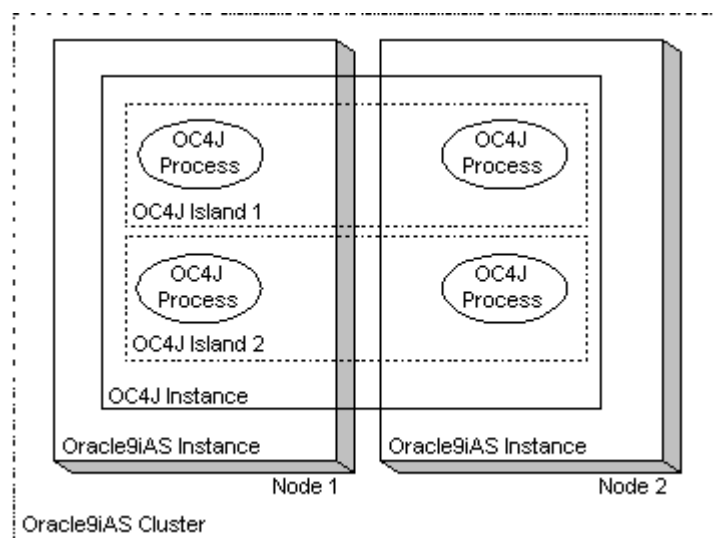
When configuring OC4J islands (using OEM), you can specify the number of OC4J processes for each node that belong to each island. By doing so, you can increase or decrease the number of processes based on the capabilities of the hardware and operating system of each node. For instructions on how to configure Oracle9*i*AS clusters and OC4J islands, refer to *Oracle9i Application Server Administrator's Guide.*

J2EE Support Comparison

This section outlines the differences in the level of support of J2EE specifications between WebSphere and Oracle9*i*AS.

WebSphere J2EE support

WebSphere 3.5.3 is a J2EE server, but is not fully J2EE 1.2 compliant. It supports the following J2EE API specifications:

- Servlet 2.1 (and partial support for Servlet 2.2)
- JSPs - supports .91 and 1.0
- EJBs 1.0+
- JTA 1.0

- JNDI 1.2
- JDBC 2.0
- JMS 1.0

WebSphere is not fully J2EE compliant, since it provides custom extensions to J2EE standards and includes non-standard packages for supporting J2EE features, such as servlet filtering and chaining, security, connection pooling and data access beans, and deployment dersciptors. An application using these extensions and packages requires code-level changes in order to migrate to Oracle9iAS or any other J2EE-compliant application server.

Oracle9iAS OC4J J2EE support

Oracle9iAS OC4J is fully certified with J2EE 1.2.1, having passed Sun Microsystems' Certification Test Suite (CTS). The CTS includes over 5,000 tests designed to assess application portability and the overall quality of a J2EE implementation.

Table 2–1 lists the J2EE technologies and the level of support provided by Oracle9iAS and WebSphere:

Table 2–1   J2EE Technology Support

J2EE Technology	Version Supported by WebSphere 3.5.3	Version Supported by Oracle9iAS
JDK	1.2.2	1.2.2 and 1.3
Servlets	2.1+	2.2 and 2.3
JSPs	1.0	1.2
EJBs	1.0+	1.1 and 2.0
JDBC	2.0	2.0
JNDI	1.2	1.2.1
JTA	1.0	1.0.1
JMS	1.0	1.0.1
JavaMail	None	1.2
JAF	None	1.0.1
JAXP	1.0.1	1.1
JCA	1.0	1.0

Table 2–1   J2EE Technology Support

J2EE Technology	Version Supported by WebSphere 3.5.3	Version Supported by Oracle9*i*AS
JAAS	1.0	1.0

In addition to supporting these standards, Oracle9*i*AS provides a well-thought-out, integrated architecture for building real world J2EE applications, including implementation of standard deployment archives: JAR files for EJBs, Web Archives (WARs) for servlets and JSPs, and Enterprise Archives (EARs) for applications. This ensures smooth server interoperability.

Java Development and Deployment Tools

This section compares the Java tools included with WebSphere and Oracle9*i*AS.

WebSphere Development and Deployment Tools

The WebSphere development environment, tools, and system administration console are described below.

WebSphere Development Tools

VisualAge for Java is is IBM's integrated development environment (IDE) for building J2EE applications. VisualAge for Java offers remote debugging support for JSP pages and other server-side Java logic. A new Servlet SmartGuide generates servlets, JSP components, and HTML prototypes, so that developers can quickly test their code inside the IDE before deploying to a production server. Integration with IBM WebSphere Studio allows for quick addition of content to prototypes, increasing productivity for programmers and web developers. VisualAge also comes with Persistence Builder, a standalone object-relational mapper tool .

WebSphere Studio

The WebSphere Studio provides a tool set for creating, managing and debugging multiplatform Web applications. It includes the following functionality:

- Visual Page Designer for Java Server Pages (JSP), HTML and DHTML.

- Wizards to create database applications, queries, JavaBeans and servlets.

- Deployment of EJBs, servlets and web applications.

WebSphere Administrative Console

The WebSphere Administrative Console provides a GUI for managing the WebSphere domain. A WebSphere domain consists of one or more WebSphere instances (where each instance runs one or more applications). The Administrative Console connects to one of the Administrative servers running in the domain and can be used change to the configuration or run-time state on any machine in a domain. The Administrative Console is used to manage the administrative repository, deploy applications and configure applications.

Oracle9*i*AS Development and Deployment Tools

This section describes development and deployment tools for creating J2EE applications. The tools are part of the Oracle9*i* Developer Suite.

Development Tools

Application developers can use the tools in Oracle JDeveloper to build J2EE compliant applications for deployment on OC4J. JDeveloper is a component in Oracle Internet Developer Suite, a full-featured, integrated development environment for creating multi-tier Java applications. It enables you to develop, debug, and deploy Java client applications, dynamic HTML applications, web and application server components and database stored procedures based on industry-standard models. For creating multi-tier Java applications, JDeveloper has the following features:

- Oracle Business Components for Java (BC4J)

- Web application development

- Java client application development

- Java in the database

- Component-Based Development with JavaBeans

- Simplified database access

- Visual Integrated Development Environment

- Complete J2EE 1.2 support

- Automatic generation of `.ear` files, `.war` files, EJB JAR file, and deployment descriptors.

You can build applications with Oracle JDeveloper and deploy them manually, using Oracle Enterprise Manager. Also note that you are not restricted to using

JDeveloper to build applications; you can deploy applications built with IBM VisualAge or Borland JBuilder on OC4J.

Assembly Tools

Oracle9*i*AS provides a number of assembly tools to configure and package J2EE Applications. The output from these tools is compliant with J2EE standards and is not specific to OC4J. These include:

- A WAR file assembly tool to assemble JSP, servlets, tag libraries and static content into WAR files.

- An EJB assembler, which packages an EJB home, remote interface, deployment descriptor, and the EJB into a standard JAR file.

- An EAR File assembly tool, which assembles WAR Files and EJB JARs into standard EAR files.

- A tag library assembly tool, which assembles JSP tag libraries into standard JAR files.

Administration Tools

Oracle9*i*AS also provides two different administration facilities to configure, monitor, and administer OC4J.

- A graphical management console, integrated with Oracle Enterprise Manager, which provides a single point of administration across Oracle9*i*AS clusters, farms, and OC4J containers.

- A command line tool for performing administrative tasks locally or remotely from a command prompt. (Oracle Enterprise Manager (OEM) is the preferred administration environment over this command line tool as OEM provides a more integrated set of administration services.)

3

Migrating Servlets

This chapter discusses key servlet features and APIs, WebSphere support for servlet APIs and its extensions to standards, and OC4J support for servlet APIs. It also includes a step-by-step migration path for servlets deployed on WebSphere to Oracle9*iAS* OC4J container.

Overview of the Java Servlet API

A servlet is an instance of Java class running with in a web container and servlet engine. Servlets are used for generating dynamic web pages. Servlets receive and respond to requests from web clients, usually via the HTTP protocol.

Servlets have several advantages over traditional CGI programming:

- A servlet does not run in a separate process. This removes the overhead of creating a new process for each request.

- A servlet stays in memory between requests. A CGI program (and probably also an extensive runtime system or interpreter) needs to be loaded and started for each CGI request.

- There is only a single instance which answers all requests concurrently. This saves memory and allows a servlet to easily manage persistent data.

- A servlet can be run by a servlet engine in a restrictive sandbox (similar to how an applet runs in a web browser's sandbox), which allows secure use of servlets.

- Servlets are scalable, providing support for a multi-application server configuration. Servlets also enable data caching, database access, and data sharing with other servlets, JSP files and (in some environments) Enterprise JavaBeans.

The servlet API is specified in two java extension packages: `javax.servlet` and `javax.servlet.http`. Most servlets, however, extend one of the standard implementations of that interface, namely `javax.servlet.GenericServlet` and `javax.servlet.http.HttpServlet`. Of these, the classes and interfaces in `javax.servlet` are protocol independent, while `javax.servlet.http` contain classes specific to HTTP.

The servlet API provides support in four categories:

- Servlet life cycle management

- Access to servlet context

- Utility classes

- HTTP-specific support classes

Table 3–1 identifies the servlet API classes according to the purpose they serve.

Table 3–1   Servlet API Classes

Purpose	Class or Interface
Servlet implementation	`javax.servlet.Servlet`
	`javax.servlet.SingleThreadModel`
	`javax.servlet.GenericServlet`
	`javax.servlet.httpServlet`
Servlet configuration	`javax.servlet.ServletConfig`
Servlet exceptions	`javax.servlet.ServletException`
	`javax.servlet.UnavailableException`
Request/response	`javax.servlet.ServletRequest`
	`javax.servlet.ServletResponse`
	`javax.servlet.ServletInputStream`
	`javax.servlet.ServletOutputStream`
	`javax.servlet.http.HttpServletRequest`
	`javax.servlet.http.HttpServletResponse`

Table 3–1   Servlet API Classes

Purpose	Class or Interface
Session tracking	`javax.servlet.http.HttpSession`
	`javax.servlet.http.HttpSessionBindingListner`
	`javax.servlet.http.HttpSessionBindingEvent`
	`javax.servlet.http.Cookie`
Servlet context	`javax.servlet.ServletContext`
Servlet collaboration	`javax.servlet.RequestDispatcher`

Servlet Lifecycle

Servlets run on the web server platform as part of the same process as the web server itself. The web server is responsible for initializing, invoking, and destroying each servlet instance. A web server communicates with a servlet through a simple interface, `javax.servlet.Servlet`.

This interface consists of three main methods

- `init()`

- `service()`

- `destroy()`

and two ancillary methods:

- `getServletConfig()`

- `getServletInfo()`

The `init()` Method

When a servlet is first loaded, its `init()` method is invoked, and begins initial processing such as opening files or establishing connections to servers. If a servlet has been permanently installed in a server, it is loaded when the server starts.

Otherwise, the server activates a servlet when it receives the first client request for the services provided by the servlet. The `init()` method is guaranteed to finish before any other calls are made to the servlet, such as a call to the `service()` method. The `init()` method is called only once; it is not called again unless the servlet is reloaded by the server.

The `init()` method takes one argument, a reference to a `ServletConfig` object, which provides initialization arguments for the servlet. This object has a method `getServletContext()` that returns a `ServletContext` object, which contains information about the servlet's environment.

The `service()` Method

The `service()` method is the heart of the servlet. Each request from a client results in a single call to the servlet's `service()` method. The `service()` method reads the request and produces the response from its two parameters:

- A `ServletRequest` object with data from the client. The data consists of name/value pairs of parameters and an `InputStream`. Several methods are provided that return the client's parameter information. The `InputStream` from the client can be obtained via the `getInputStream()` method. This method returns a `ServletInputStream`, which can be used to get additional data from the client. If you are interested in processing character-level data instead of byte-level data, you can get a `BufferedReader` instead with `getReader()`.

- A `ServletResponse` represents the servlet's reply back to the client. When preparing a response, the method `setContentType()` is called first to set the MIME type of the reply. Next, the method `getOutputStream()` or `getWriter()` can be used to obtain a `ServletOutputStream` or `PrintWriter`, respectively, to send data back to the client.

There are two ways for a client to send information to a servlet. The first is to send parameter values and the second is to send information via the `InputStream` (or `Reader`). Parameter values can be embedded into a URL. The `service()` method's job is simple--it creates a response for each client request sent to it from the host server. However, note that there can be multiple service requests being processed simultaneously. If a service method requires any outside resources, such as files, databases, or some external data, resource access must be thread-safe.

The `destroy()` Method

The `destroy()` method is called to allow the servlet to clean up any resources (such as open files or database connections) before the servlet is unloaded. If no clean-up operations are required, this can be an empty method.

The server waits to call the `destroy()` method until either all service calls are complete, or a certain amount of time has passed. This means that the `destroy()` method can be called while some longer-running `service()` methods are still

running. It is important that you write your `destroy()` method to avoid closing any necessary resources until all `service()` calls have completed.

Session Tracking

HTTP is a stateless protocol, which means that every time a client requests a resource, the protocol opens a separate connection to the server, and the server doesn't preserve the context from one connection to another; each transaction is a isolated. However, most web applications aren't stateless. Robust Web applications need to interact with with users and remember the user the nature of a given user's requests, making data collected about the user in one request available to the next request from the same user. A classic example would be the shopping cart application, from internet commerce. The Servlet API provides techniques for identifying a session and associating data with it, even over multiple connections. These techniques include the following:

- Cookies
- URL rewriting
- Hidden form fields

To eliminate the need for manually managing the session information within application code (regardless of the technique used), you use the `HttpSession` class of the Java Servlet API. The `HttpSession` interface allows servlets to:

- View and manage information about a session
- Preserve information across multiple user connections, to include multiple page requests as well as connections

Cookies

Cookies are probably the most common approach for session tracking. Cookies store information about a session in a human-readable file on the client's machine. Subsequent sessions can access the cookie to extract information. The server associates a session ID from the cookie with the data from that session. This becomes more complicated when there are multiple cookies involved, when a decision must be made about when to expire the cookie, and when many unique session identifiers are needed. Also, a cookie has a maximum size of 4K, and no domain can have more than 20 cookies. Cookies pose some privacy concerns for users. Some people don't like the fact that a program can store and retrieve information from their local disk, and disable cookies or delete them altogether. Therefore, they are not dependable as a sole mechanism for session tracking.

URL rewriting

The URL rewriting technique works by appending data to the end of each URL that identifies a session. The server associates the identifier with data it has stored about the session. The URL is constructed using an HTTP GET, and may include a query string containing pairs of parameters and values. For example:

```
http://www.server.com/getPreferences?uid=username&bgcolor=red&fgcolor=blue.
```

Hidden form fields in HTML

Hidden form fields are another way to store information about the session. The hidden data can be retrieved later by using the HTTPServletRequest object. When a form is submitted, the data is included in the GET or POST. A note of caution though: form fields can be used only on dynamically generated pages,so their use is limited. And there are security holes: people can view the HTML source to see the stored data.

The `HttpSession` object

No matter the technique(s) used to collect session data, it must be stored somewhere. The `HttpSession` object can be used to store the session data from a servlet and associate it with a user.

The basic steps for using the `HttpSession` object are:

1. Obtain a session object

2. Read or write to it

3. Terminate the session by expiring it, or allowing it to expire on its own

A session persists for a certain time period, up to forever, depending on the value set in the servlet. A unique session ID is used to track multiple requests from the same client to the server. Persistence is valid within the context of the Web application, which may encompass multiple servlets. A servlet can access an object stored by another servlet; the object is distinguished by name and is considered bound to the session. These objects (called attributes when set and get methods are performed on them) are available to other servlets within the scope of a request, a session, or an application.

Servlets are used to maintain state between requests, which is cumbersome to implement in traditional CGI and many CGI alternatives. Only a single instance of the servlet is created, and each request simply results in a new thread calling the servlet's `service` method (which calls `doGet` or `doPost`). So, shared data simply has to be placed in a regular instance variable (field) of the servlet. Thus,the servlet

can access the appropriate ongoing calculation when the browser reloads the page and can keep a list of the N most recently requested results, returning them immediately if a new request specifies the same parameters as a recent one. Of course, the normal rules that require authors to synchronize multithreaded access to shared data still apply to servlets.

Servlets can also store persistent data in the `ServletContext` object, available through the `getServletContext` method. `ServletContext` has `setAttribute` and `getAttribute` methods that enable storage of arbitrary data associated with specified keys. The difference between storing data in instance variables and storing it in the `ServletContext` is that the `ServletContext` is shared by all servlets in the servlet engine or in the Web application.

J2EE Web Applications

A Web application, as defined in the servlet specification, is a collection of servlets, JavaServer Pages (JSPs), Java utility classes and libraries, static documents such as HTML pages, images , client side applets, beans, and classes, and other Web resources that are set up in such a way as to be portably deployed across any servlet-enabled Web server. A Web applications, can be contained in entirety within a single archive file and deployed by placing the file into a specific directory.

Web Application Archive (WAR)

Web application archive files have the extension `.war`. WAR files are `.jar` files (created using the `jar` utility) saved with an alternate extension. The JAR format allows JAR files to be stored in compressed form and have their contents digitally signed. The `.war` file extension was chosen over `.jar` to distinguish them for certain operations. An example of a WAR file listing is shown below:

```
index.html
howto.jsp
feedback.jsp
images/banner.gif
images/jumping.gif
WEB-INF/web.xml
WEB-INF/lib/jspbean.jar
WEB-INF/classes/MyServlet.class
WEB-INF/classes/com/mycorp/frontend/CorpServlet.class
WEB-INF/classes/com/mycorp/frontend/SupportClass.class
```

On install, a WAR file can be mapped to any URI prefix path on the server. The WAR file then handles all requests beginning with that prefix. For example, if the

WAR file above were installed under the prefix `/demo`, the server would use it to handle all requests beginning with `/demo`. A request for `/demo/index.html` would serve the `index.html` file from the WAR file. A request for `/demo/howto.jsp` or `/demo/images/banner.gif` would also serve content from the WAR file.

About the `WEB-INF` directory

The `WEB-INF` directory is special. The files in it are not served directly to the client; instead, they contain Java classes and configuration information for the Web application. The directory behaves like a JAR file's `META-INF` directory; it contains metainformation about the archive contents. The `WEB-INF/classes` directory contains the class files for the Web application's servlets and supporting classes. `WEB-INF/lib` contains classes stored in JAR files. For convenience, web server class loaders automatically look to `WEB-INF/classes` and `WEB-INF/lib` for their classes—no extra install steps are necessary.

The servlets under `WEB-INF` in the example Web application listing can be invoked using URIs like `/demo/servlet/MyServlet` and `/demo/servlet/com.mycorp.frontend.CorpServlet`.

Note that every request for this application begins with `/demo`, even requests for servlets.

The `web.xml` file in the `WEB-INF` directory defines descriptors for a Web Application. This file contains configuration information about the Web application in which it resides and is used to register your servlets, define servlet initialization parameters, register JSP tag libraries, define security constraints, and other Web Application parameters .

Differences between Servlet 2.0, 2.1 and 2.2

The Servlet API in the J2EE specification is continously evolving. In a span of two years Servlet API 2.0 , 2.1, 2.2 has been published; the most recent version as of this writing is Servlet API 2.3. The fundamental architecture of servlets has not changed much, so most of the API is still relevant. However, there are enhancements and some new functionality, and some APIs have been deprecated.

This section will cover the major difference between Servlet API 2.0 , 2.1 ,2.2 and 2.3 draft specification.

Highlights of the Java Servlet API 2.1

The Servlet 2.1 API highlights include:

- A request dispatcher wrapper for each resource (servlet)

  A request dispatcher is a wrapper for resources that can process HTTP requests (such as servlets andJSPs) and files related to those resources (such as static HTML and GIFs). The servlet engine generatesa single request dispatcher for each servlet or JSP when it is instantiated. The request dispatcher receives client requests and dispatches the request to the resource.

- A servlet context for each application

  In Servlet API 2.0, the servlet engine generated a single servlet context that was shared by all servlets. The Servlet API 2.1 provides a single servlet context per application, which facilitates partitioning applications. As explained in the description of the application programming model, applications on the same virtual host can access each other's servlet context.

- Deprecated HTTP session context

  The Servlet API 2.0 `HttpSessionContext` interface grouped all of the sessions for a Web server into a single session context. Using the session context interface methods, a servlet could get a list of the session IDs for the session context and get the session associated with an ID. As a security safeguard, this interface has been deprecated in the Servlet API 2.1. The interface methods have been redefined to return null.

New Features in the Java Servlet API 2.2

The Servlet API 2.2 specification changed the term 'servlet engine', replacing it with 'servlet container'. This change is indicative of the Java Servlet API is now a required API of the Java 2 Platform, Enterprise Edition (J2EE) specification and, throughout J2EE's terminology, container is preferred over engine. Servlet API 2.2 introduced the following new features:

- Web Applications (as discussed above)

- References to external data sources, such as JNDI. Enables adding resources into the JNDI lookup table, such as database connections. Allows the resources to be located by servlets using a simple name lookup.

- Parameter information for the application (initiallization parameters for the application).

- Registered servlet names. Provides a place to register servlets and give them names. Previously, each server had a different process for registering servlets, making deployment difficult.

- Servlet initialization parameters. Enables passing parameters to servlets parameters at initialization time. This is a new, standard way to accomplish what used to be a server dependent process.

- Servlet load order. Specifies which servlets are preloaded, and in what order.

- Security constraints. Dictate which pages must be protected, and by what mechanism. Include built-in form-based authentication.

Servlet API 2.3

The Servlet API 2.3 leaves the core of servlets relatively untouched. Additions and changes include:

- JDK 1.2 or later is required

- A filter mechanism has been created

- Application lifecycle events have been added

- Additional internationalization support has been added

- The technique to express inter-JAR dependencies has been formalized

- Rules for class loading have been clarified

- Error and security attributes have been added

- The `HttpUtils` class has been deprecated

- Several DTD behaviors have been expanded and clarified

Filters and Servlet Chaining

Filtering support is provided as a part of the Servlet 2.3 API. WebSphere Advanced Edition 3.5.3 achieves similar filtering functionality with a WebSphere-specific package. Oracle9*i*AS Containers for J2EE (OC4J) supports the Java servlet 2.3 filtering specification.

Filtering is a method of loading and invoking servlets in a web server. Both local and remote servlets can be part of a servlet chain (defined below). There are restrictions, however, on chaining the local internal servlets, and these restrictions are specific to the J2EE container used. For example, in WebSphere, if an internal servlet is used in a chain, it must be the first servlet in the chain. Internal servlets include: file servlet, pageCompile servlet, ssInclude servlet, and template servlet.

Servlet Chains

For some requests, a chain of ordered servlets can be invoked rather than just one servlet. The input from the browser is sent to the first servlet in the chain and the output from the last servlet in the chain is the response sent back to the browser. Each servlet in the chain receives inputs from, and transmits outputs to, the servlet before and after it, respectively. A chain of servlets can be triggered for an incoming request by using:

- Servlet aliasing to indicate a chain of servlets for a request

- MIME types to trigger the next servlet in the chain

WebSphere Servlet API Support

WebSphere version 3.5.2 and version 3.5.3 maintain compatibility with existing applications while simultaneously supporting the Java Servlet API 2.2 specification. But this support is partial, and you can choose only one. To ensure compatibility, a new option was added to servlet container properties in the Administrative console. This new option, the Select Servlet Engine Mode, is located on the Servlet Engine Properties view. The Select Servlet Engine Mode option toggles between the following two different 'runtime' modes:

WebSphere Advanced Edition 3.5.3 Compatibility Mode

This mode maintains behavior with existing WebSphere Application Server v3.5 and v3.5.1 applications at the expense of full compliance with the Java Servlet API 2.2 specification. In compatibility mode, the servlet engine is Servlet 2.2 specification level compliant, except for the method and behavior changes noted below. This capability is provided to allow existing WebSphere Advanced Edition v3.5 and v3.5.1 applications to successfully execute until they are migrated to fully compliant Servlet 2.2 level applications.

Full Servlet 2.2 Compliance Mode

This mode maintains compliance with the Java Servlet API 2.2 specification at the expense of compatibility with existing WebSphere Application Server v3.5 and v3.5.1 applications.

The default mode is the Compatibility Mode. You select the desired mode using the Administrative Console, Servlet Engine General tab.

Servlet 2.2 API Support

WebSphere Advanced Edition 3.5.3 has partial support for the Servlet 2.2 API. The supported API features are:

- Response Buffering
- Multiple Error page support
- Welcome File Lists
- New request mapping logic
- Session Timeout per web application
- Mime mapping table per web application
- Request Dispatchers by name
- Request Dispatchers by relative path
- Duplicate Header support (`addHeader()`, `getHeaders(name)` APIs)
- Initialization parameters on a web application
- Internationalization improvements (`getLocale()`, `getLocales()`)
- New APIs `getServletName()`

The following Servlet 2.2 API features are not supported:

- J2EE Security Roles
  - Security deployment information in `web.xml`
  - `isUserInRole()`
  - `getUserPrincipal()` API
- J2EE-style Form Login
- J2EE References
  - EJB reference
  - Resource Reference
  - Environment Reference
  - Environment Entry
  - Refrence deployment information in `web.xml`
  - Security deployment information in `web.xml`

WebSphere Extensions to the Servlet API

The WebSphere Application Server includes its own packages that extends and adds to the Java Servlet API. The extensions and additions are provided to manage session state, create personalized Web pages, generate better servlet error reports, and access databases.

The Application Server API packages and classes are:

- `com.ibm.servlet.personalization.sessiontracking`

    Records the referral page that led a visitor to a web site, tracks the visitor's position within the site, and associates user identification with the session. IBM has also added session clustering support to the API.

- `com.ibm.websphere.servlet.session.IBMSession` interface

    Extends `HttpSession` for session support and increases Web administrators' control in a session cluster environment.

- `com.ibm.servlet.personalization.userprofile` package

    Provides an interface for maintaining detailed information about web visitors and incorporate it in your applications, so that you can provide a personalized user experience. This information stored it in a database.

- `com.ibm.websphere.userprofile` package

    User profile enhancements.

- `com.ibm.websphere.servlet.error.ServletErrorReport` class

    Enables the application to provide more detailed and tailored messages to the client when errors occur.

- `com.ibm.websphere.servlet.event` package

    Provides listener interfaces for notifications of application lifecycle events, servlet lifecycle events, and servlet errors. The package also includes an interface for registering listeners.

- `com.ibm.websphere.servlet.filter` package

    Provides classes that support servlet chaining. The package includes the `ChainerServlet`, the `ServletChain` object, and the `ChainResponse` object.

- `com.ibm.websphere.servlet.request` package

Provides an abstract class, `HttpServletRequestProxy`, for overloading the servlet engine's `HttpServletRequest` object. The overloaded request object is forwarded to another servlet for processing. The package also includes the `ServletInputStreamAdapter` class for converting an `InputStream` into a `ServletInputStream` and proxying all method calls to the underlying `InputStream`.

■   `com.ibm.websphere.servlet.response` package

Provides an abstract class, `HttpServletResponseProxy`, for overloading the servlet engine's `HttpServletResponse` object. The overloaded response object is forwarded to another servlet for processing. The package includes the `ServletOutputStreamAdapter` class for converting an `OutputStream` into a `ServletOutputStream` and proxying all method calls to the underlying `OutputStream`.

The package also includes the `StoredResponse` object that is useful for caching a servlet response that contains data that is not expected to change for a period of time, for example, a weather forecast.

Oracle9*iAS* Servlet API Suport

Oracle9*iAS* OC4J is a fully compliant implementation of the Java Servlets 2.2 and 2.3 specifications. As such, standard Java Servlets 2.2 code will work correctly. WebSphere Advanced Edition, on the other hand, has partial support for the Java Servlets 2.2 specification as described above. In particular, the security support remains at the Servlet 2.1 level, and there is no support for J2EE references that would normally be defined in the `web.xml` file associated with the Web application. There is also no direct support for J2EE Web Abpplications.

Because of these differences in API support and WebSphere extensions, an application may require code level changes before it can be migrated if it uses extensions or deprecated method calls. Since WebSphere does not support J2EE deployment descriptors, existing applications must be packaged into the J2EE Web Application structure before deployment on Oracle9*iAS* OC4J.

Migrating Standalone Servlets to OC4J

We migrated example servlets provided with WebSphere Advanced Edition 3.5.3. Some of these examples were not migrated because they used WebSphere-specific extensions. For example, we did not migrate `AbstractLoginServlet` because it uses a single sign-on package specific to WebSphere.

We migrated these servlets (located in in `WebSphereInstallHome/Servlets`):

- Custom Login Servlet

- HelloWorldServlet

- SessionServlet

In addition to these, we migrated packaged Web Applications that use WebSphere-specific deployment descriptors.

These examples were migrated without code changes. All that was required was to place these servlets in
<ORACLE_HOME>/j2ee/home/default-web-app/WEB-INF/classes in UNIX or *<ORACLE_HOME>*\j2ee\home\default-web-app\WEB-INF\classes in NT. The OC4J servlet container loads these servlets automatically. You can invoke these servlets from a browser using
http://hostname:portnumber/servlet/HelloWorldServlet.

WebSphere provides another way of deploying standalone servlets (that is, servlets that require initialization parameters and configuration information). These servlets are deployed in WebSphere using a deployment descriptor whose name is the name of the servlet and ends with .servlet. This WebSphere-specific deployment descriptor must be migrated to the J2EE Web application deployment descriptor before it can be deployed in OC4J.

Example 3–1   SnoopServlet.servlet Deployment Descriptor

```
<servlet>
  <name>snoop</name>
  <description>snoop servlet</description>
  <code>SnoopServlet</code>
  <servlet-path>/servlet/snoop/*</servlet-path>
  <servlet-path>/servlet/snoop2/*</servlet-path>
  <init-parameter>
    <name>param1</name>
    <value>test-value1</value>
  </init-parameter>
  <autostart>false</autostart>
</servlet>
```

Sample `.servlet` file: `SnoopServlet.servlet`

The Snoop Servlet can be migrated by placing Servlet in
<ORACLE_HOME>/j2ee/home/default-web-app/WEB-INF/classes in UNIX or *<ORACLE_HOME>*\j2ee\home\default-web-app\WEB-INF\classes in NT

and editing web.xml located in the following directory:

UNIX:

`<ORACLE_HOME>/j2ee/home/default-web-app/WEB-INF`

NT:

`<ORACLE_HOME>\j2ee\home\default-web-app\WEB-INF`

Migrated SnoopServlet Deployment Descriptor

```
<web-app>
  <servlet>
    <servlet-name>snoop</servlet-name>
    <description>snoop servlet</description>
    <servlet-class>SnoopServlet</servlet-class>
    <servlet-path>/servlet/snoop/*</servlet-path>
    <servlet-path>/servlet/snoop2/*?/servlet-path?
    <init-param>
      <param-name>param1</param-name>
      <param-value>test-value1</param-value>
    </init-param>
    <autostart>false</autostart>
  </servlet>
</web-app>
```

Migrating Cluster-aware applications to OC4J

Clustering and load balancing are two key features of an enterprise application server. These features make the application server available, fault tolerant, and scalable. The load balancer replicates state of an individual node to the cluster of instances so that if a node fails, the state information is preserved elsewhere. The cluster configuration provided by OC4J accomplishes the following:

- Maximizes use of resources

  If an application cannot make full use of a machine's resources, OC4J can help make more efficient useof the processing power.

- Maximize throughput

  OC4J can dramatically increase the number of requests an application can serve concurrently.

- Minimize risks of single points of failure

  OC4J builds redundancy into your configuration. If one instance fails, others can continue to process requests.

WebSphere and Oracle9*iAS* OC4J both provide clustering and load balancing session failover. Oracle9*iAS* OC4J also supports HTTP tunneling of RMI requests and responses without clustering. If you have a cluster-aware application running on WebSphere, it can be migrated to an OC4J instance (a set of OC4J processes, equivalent to a cluster).

island. The OC4J configuration incorporates the concept of islands. An island is a set of OC4J processes that have uniform application configuration and replicated application state. An island is a subset of processes within an OC4J instance.

Configuring an OC4J Island (in OC4J standalone mode)

> **Note:** The instructions in this section show you how to configure a island manually. This can be done in a development environment where OC4J is running in standalone mode. If you are configuring an island in an Oracle9*iAS* cluster, **use the Oracle Enterprise Manager web pages or the `dcmctl` command line utility**. Information on using these can be found in *Oracle9i Application Server Administrator's Guide* and *Oracle9iAS Containers for J2EE User's Guide*

The following steps explain how to configure an OC4J island:

1. Install your web application on all of the nodes in your cluster.

   a. First, make sure that the nodes you are using in your cluster have the same web application installed. If you do not want to install the application in two places, you can place it on a shared drive that both servers access.

   b. Start all your nodes and check that the web-applications are working correctly on all of them.

2. Set up your web-application to replicate its state to the cluster.

   a. Edit the `orion-web.xml` deployment descriptor for the web application, located at the following directory:

   UNIX:
   `<ORACLE_HOME>/j2ee/home/application-deployments/`
   `application-name/web-app-name/`

NT:
```
<ORACLE_HOME>\j2ee\home\application-deployments\
application-name\web-app-name\
```

b. If you want to add clustering for all web applications in the site, edit the
    `orion-web.xml` of the global web application located at the following
    directory:

UNIX:
```
<ORACLE_HOME>/j2ee/home/config/
global-web-application.xml
```

NT:
```
<ORACLE_HOME>\j2ee\home\config\
global-web-application.xml
```

Add the following to the main body of the `<orion-web-app>` tag:

```
<cluster-config/>
```

3. Optional: Specify the multicast host and IP address on which to transmit and
    receive cluster data.

4. Optional: Specify the port on which to transmit and receive cluster data.

5. Specify the ID (number) of the node to identify itself within the cluster. The
    default is localhost.

6. Optional: Repeat steps 4, 5 and 6 for all the nodes in your cluster.

The `HTTPSession` data will now be replicated (as long as it is serializable, or
an EJB reference). Note, however, that if the EJBs are located on a server that
goes down, the references might become invalid. The `ServletContext` data is
also replicated.

> **Note:** It is important to understand that load balancing, in this
> case, is implemented for the web-component, not the EJB (EJBs
> have a different way of load balancing using client stubs). When
> using multiple islands, you may want to use different multicast IP
> addresses, to enable smart routing of multicast packets in your
> network, and just send traffic on certain IP addresses to certain
> servers.

7. Configure your islands.

Islands are connected to a certain site rather than to a web-application. To configure an island:

a. Edit the `web-site.xml` file for the website your web application is deployed on (for example, `default-web-site.xml` if you are clustering the default web-site). Add the following to the `<web-site>` tag:

```
cluster-island="1"
```

If your cluster has more than one island, you will specify different island values for the servers that belong to different islands. State is shared only within an island.

b. Specify the host the web-site is serving using the `host="<hostname/ip address>"` attribute in the `<web-site>` tag.

8. Tell the servers about the load balancer. In the same file, the `web-site.xml` for your web site, you also specify where the load balancer for the site is located.

a. In the main body of the `<website>` tag, add:

```
<frontend host="balancer hostname" port="balancer port" />
```

where `balancer hostname` and `balancer port` are the hostname and port of the server that will be running the load balancer.

9. In the `/WEB-INF/web.xml` or NT equivalent of your application, put in the tag

```
</distributable>
```

This tag indicates that the application is distributable (a feature of the J2EE 1.2 specification).

10. Access the load balancer's host and port with a browser. You will notice how the request is sent to a server. If you request the same page again from the same client, your request will probably be sent to the same server again, but if you request the same page from different clients, you will see that the client requests get balanced.

To test the state replication, you can try accessing the servlet in the following directory:

UNIX:
`<ORACLE_HOME>/j2ee/home/servlet/SessionServlet`

NT:
`<ORACLE_HOME>\j2ee\home\servlet\SessionServlet`

Make the request once, and check which server becomes the primary server for the session. Stop that server and make the request again. The desired result is that the request is part of the same session as before but on a different node. And, the counter is updated correctly.

How OC4J Island Works (in OC4J standalone mode)

For all of the islands, there is a single load balancer OC4J instance that dispatches requests to the application clones.

- If a new request is made from an IP address that has not connected to the site before, and has no session associated with it, it is sent to a random OC4J instance. If more than one island in the cluster is capable of serving the same site, an island is chosen at random. Thereafter, a random node is picked within the selected island.

- All state replication occurs within the island of this selected node. If a request is made from an IP address that has connected to to the website before, the request will be sent to the same server as the previous request (unless the configuration specifies that requests not be routed based on IP address).

- By default, load balancing is based on client, not on request. In other words, statistically speaking, default load balancing is expected to send off an equal number of clients to each node in the island. Note that an equal number of clients to nodes in the island does not equate to an equal number of requests to the server, since each client makes a different number of requests.

- To make load balancing request based, you can use the "dontuseIP" switch, a powerful feature of OC4J islands.

- If a request is made within a keep-alive socket, the request will get sent to the same server as the previous request, unless you have specified that keep-alives should not be used (-dontUseKeepalives as command line option or use-keepalives="false" in load-balancer.xml

- If a request is made from a user in a session, the request is sent to the primary server for that session. If the primary server for the session does not respond, the request will be sent to another server in the same island. Since the state has been replicated, the other server has the same user state.

4

Migrating JSPs

This chapter describes how to migrate JavaServer Pages(JSPs) from WebSphere
Application Server Advanced Edition to Oracle9*i*AS OC4J. The JSP API, and the
details of the WebSphere extensions and different JSP engines it supports are
discussed. The process of migrating JSPs from WebSphere to Oracle9*i*AS is outlined
at the end of the chapter. The chapter is organized as follows:

- Overview of JSP Pages

- WebSphere Support for the JSP API

- Migrating from WebSphere JSP 0.91

- Migrating WebSphere Extensions to OC4J

Overview of JSP Pages

JavaServer Pages is a technology specified by Sun Microsystems as a method of
generating dynamic content from an application running on a web server. This
technology, which is closely coupled with Java servlet technology, allows you to
include Java code snippets and calls to external Java components within the HTML
code (or other markup code, such as XML) of your Web pages.

A JSP page is translated into a Java servlet before being executed (typically on
demand, but sometimes in advance). As a servlet, it processes HTTP requests and
generates responses. JSP technology offers a more convenient way to code the
servlet instead of embedding HTML tags in the servlets. Furthermore, JSP pages are
fully interoperable with servlets—that is, JSP pages can include output from a
servlet or forward output to a servlet, and servlets can include output from a JSP
page or forward to a JSP page.

Parts of a JSP Page

A JSP page typically consists of the following:

- Directives - imports and and interfaces
- Declarations - class-wide variables and methods
- Expressions - return value substitution
- Scriptlets - inline Java code

Each is described below.

Directives

Directives are compile-time control tags. They allow you to customize how your JSP pages are compiled to Java servlets. There are three types of directives:

Page A page directive is placed at the top of a JSP page. Its attributes apply to the entire page.

Example:

```
<%@ page language="java" import="com.mycom.*" buffer="16k" %>
```

Taglib A taglib directive extends the set of tags recognized by the JSP processor. It requires two attributes, uri and prefix.

Example:

```
<%@ taglib uri="tag-lib-uri" prefix="tag-prefix" %>
```

The uri attribute contains the location of the tag library TLD (Tag Library Descriptor) file.

The prefix attribute specifies the tag prefix you want to use for your custom tag.

Example:

```
<%@ taglib uri="/WEB-INF/tlds/myapp.tld" prefix="custom" %>
```

Include The include directive enables you to insert the content of another file into the JSP page at compilation time. Its syntax is as follows:

```
<%@ include file="localOrAbsoluteURL" %>
```

During compilation the content of the file specified in the file attribute will be added to the current JSP page.

What is a JSP container?

A JSP container is software that stores the JSP files and servlets, converts JSP files into servlets, compiles the servlets, and runs them (creating HTML). The exact make-up of a JSP container varies from implementation to implementation, but it will consist of a servlet or collection of servlets. The JSP container is executed by a servlet container.

The JSP container creates and compiles a servlet from each JSP file. The container produces two files for each JSP file:

- A `.java` file, which contains the Java language code for the servlet
- A `.class` file, which is the compiled servlet

The JSP container puts the `.java` and the `.class` file in a path specific to the container. The `.java` and the `.class` file have the same filename. Each container uses a naming convention for the generated `.java` and `.class` files. For example, WebSphere generates files named `_simple_xjsp.java` and `_simple_xjsp.class` from the JSP file `simple.jsp`.

Life Cycle of a JSP Page

1. The user requests the JSP page through a URL ending with a `.jsp` file name.

2. Upon noting the `.jsp` file name extension in the URL, the servlet container of the Web server invokes the JSP container.

3. The JSP container locates the JSP page and translates it if this is the first time it has been requested.

   Translation includes producing servlet code in a `.java` file and then compiling the `.java` file to produce a servlet `.class` file.

4. The servlet class generated by the JSP translator subclasses a class (provided by the JSP container) that implements the `javax.servlet.jsp.HttpJspPage` interface.

5. The servlet class is referred to as the page implementation class. This document will refer to instances of page implementation classes as JSP page instances.

WebSphere Support for the JSP API

WebSphere Advanced Edition 3.5.3 supports JSP 0.91, JSP 1.0, and, in its latest service pack, JSP 1.1. However, the support is not backward compatible.

WebSphere specifies two modes of operation for JSPs: Compatibility mode and Compliance mode. In Compatibility mode, you can choose compatibility with JSP 1.0 or JSP 0.91. For example, if you choose compatibility mode with JSP 0.91 you cannot use features of JSP 1.0 or JSP 1.1. In compliance mode, your applications are compliant with JSP 1.1.

These modes of JSP are necessary because WebSphere provides a JSP processor for each supported level of the JSP specification. Each of these JSP processors is a servlet that can be added to a web application to handle all JSP requests specific to the web application. Which JSP processor is used is dependent on the web application. If a web application includes JSPs of version 1.0, WebSphere loads the JSP processors for JSP 1.0. These is specified as a part of your web application.

Table 4–1   JSP Processors

JSP Processor	Processor Servlet Name	Class Name
JSP 1.0	`JSPServlet`	`com.sun.jsp.runtime.JspServlet in jsp10.jar`
JSP 0.91	`PageCompileServlet`	`com.ibm.servlet.jsp.http.pagecompile.` `        PageCompileServletinibmwebas.jar`

WebSphere-Specific Features

WebSphere provides several JSP features which are available in WebSphere only. These are:

Batch JSP Compiler

WebSphere provides a batch JSP compiler, enabling faster responses to requests for the JSP files. The process of batch compilation is different for JSP 0.91 and JSP 1.0.

HTML Template Extensions in JSP 0.91

WebSphere has built-in extensions for JSPs called HTML templates for variable data. These extensions are supported by the WebSphere JSP engine and are useful for generating tabular data.These template extensions consist of three additional tags:

- `<INSERT>` - This tag enables developers to insert a value based on a property name and an object specifier. This object can be a bean name or reference to a object in request object.

- `<REPEAT>` - This tag enables developers to write a "for" loop in a JSP page, as a HTML element, without embedding Java code. For example, a database query resulting in a variable result set can be iterated using a `<REPEAT>` tag instead of a embedded Java "for" loop. A `<REPEAT>` tag can contain a block of HTML tagging that in turn contains the `<INSERT>` tags, and the HTML tags for formatting content. The `<REPEAT>` tag iterates from the start value to the end value until either the end value is reached or an `ArrayIndexOutofBoundsException` is thrown. The output of a `<REPEAT>` block is buffered until the block completes. If an exception is thrown before a block completes, no output is written for that block.

- `<BEAN>` - This tag enables the developer to reference a bean in the JSP.

JSP also has tags for database connect, query, and modify: `<DBCONNECT>`, `<DBQUERY>` and `<DBMODIFY>`. The functionality of these tags has not changed in JSP 1.0 other than being moved to a new tag library, `tsx`, with names `<tsx:dbconnect>`, `<tsx:dbquery>`, and `<tsx:dbmodify>` respectively.

WebSphere Extensions to JSP 1.0

WebSphere Advanced Edition 3.5.3 provides several extensions to the base APIs. The extensions are categorized as tags for variable data and tags for database access.

Tags for variable data:

- `<tsx:repeat>` - This tag is similar to the `<REPEAT>` tag described above, useful in creating HTML tables.

- `<tsx:getProperty>` - This tag is an extension of the Sun JSP tag `<jsp:getProperty>`. It is similar to `<jsp:getProperty>`, and adds the ability to introspect a database bean that was created using the extension `<tsx:dbquery>` or `<tsx:dbmodify>`.

Tags for database access:

(These tags are useful for making database connections from a JSP and then use that connection to query or update the database. The user ID and password for the database connection can be provided by the user at request-time or hard coded within the JSP file.)

- `<tsx:dbconnect>` - This tag enables the JSP page to make a database connection to through JDBC. dbconnect tags are not used directly to establish a database connection. Instead, the `<tsx:dbquery>` and `<tsx:dbmodify>` tags are used to reference a `<tsx:dbconnect>` in the same JSP file and establish the connection to the database. Note that this is different from what is done at the application server level, where you setup a set of datasources.

- `<tsx:userid>` and `<tsx:passwd>` - These tags enable the JSP page to accept user input for the values and then add that data to the request object. The request object can be accessed by a JSP file, for example, `Account.jsp`, that requests the database connection.These two tags should be used in within a `<tsx:dbconnect>` tag.

- `<tsx:dbquery>` - This tag is used to establish a connection to a database using information specified in the `<tsx:dbconnect>` tag in the same JSP file, and query the database and return the result set. This caches the result set in a results object. At the end of the operation it closes the connection.

- `<tsx:dbmodify>`: `<tsx:dbconnect>` - This tag is used to open a new connection to a database and then update the database tables. This tag is also similar to `<tsx:dbquery>` in that it obtains database connnection information, and at the end of the operation closes the connection.

The OracleJSP container

Oracle's JavaServer Pages implementation is highly portable across server platforms and servlet environments. OracleJSP can run on any servlet environment that complies with version 2.0 or higher of the Sun Microsystems Java Servlet Specification.

The JSP Translator in Oracle9*i*AS OC4J

OC4J supports two JSP translators: the OC4J JSP translator, which is JSP 1.1 compliant, and the OJSP translator, which is JSP 1.1 compliant. The OC4J JSP translator is the default translator. Oracle tag libraries and data access beans work with both translators. You can use OJSP with OC4J if you need features such as NLS and SQLJ.

> **Note:** In Oracle9*i*AS Release 2, both the Oracle JSP and OC4J JSP translator implementations have been consolidated into one providing the best of both.

OracleJSP Extensions

OracleJSP provides some very useful functionality through custom tag libraries and custom JavaBeans, all of which are portable to other JSP environments. The extended functionality include:

- Extended data types implemented as JavaBeans that can have a specified scope integration with XML and XSL

- Data-access JavaBeans

- The Oracle JSP Markup Language (JML) custom tag library, which reduces the level of Java proficiency required for JSP development

- A custom tag library for SQL functionality

OracleJSP also provides the following Oracle-specific extensions:

Support for SQLJ, a standard syntax for embedding SQL statements directly into Java code

- Extended globalization support

- `JspScopeListener` for event handling

- The `globals.jsa` file for application support

Migrating from WebSphere JSP 0.91

This section explains how to migrate WebSphere JSP 0.91 files to OC4J.

The `<REPEATGROUP>` Tag

1. If you are migrating JSP 0.91 files that contain `<REPEATGROUP>` tags, you must change these tags. This tag is used for repeating a block of HTML, for data that is already logically grouped in the database.

2. Replace the `<SERVLET>` tag with the `<jsp:include>` tag.

   For example, change the following:

   ```
   <SERVLET CODE="com.samples.test.TestServlet"></SERVLET>
   ```

   to

   ```
   <jsp:include page="/servlet/com.samples.test.TestServlet" />
   ```

3. Replace the WebSphere `<BEAN>` tag with the `<jsp:useBean>` tag.

The example below shows the `<BEAN>` tag migrated to the JSP standard tag:

```
<BEAN NAME="AccountDBBean"
TYPE="com.test.AccountDBBean"
CREATE="YES"
INTROSPECT="YES"
SCOPE="request">
<PARAM NAME="userID" VALUE="wsdemo">
</BEAN>
```

Migrating to OC4J, the above is replaced by:

```
<jsp:useBean
id="AccountDBBean"
type="com.test.AccountDBBean"
class="com.test.AccountDBBea"
scope="request"/>

<jsp:setProperty
name="AccountDBBean"
property="userID"
value="wasdemo" />
```

- Note that the explicit attribute of `CREATE="YES"` is removed. This is because, if the bean with the name specified by the `id` attribute is not found within the specified scope, then an instance of bean will be created according to the `class` attribute. The JSP `NAME` attribute corresponds to the JSP 1.0 `id` attribute. It is no longer an `INTROSPECT` attribute. (The JSP 0.91 scope of requests and sessions carries over to JSP 1.0.)

- The `class` attribute is not necessary if the bean already exists within the specified scope . But if the `class` attribute is not specified and the bean is not in the specified scope an error will occur when creating a new instance of the bean.

4. Set the bean properties.

In JSP 0.91, the `<PARAM>` tag is used within the `<BEAN>` tag to specify properties for the bean. In JSP 1.0, you must use the `<jsp:setProperty>` tag outside of the `<jsp:useBean>` tag. You can link to the property settings of an existing bean using the `name` attribute within `<jsp:setProperty>` and specifying the bean identified by the `id` attribute in `<jsp:useBean>`. A similar way to obtain bean property values can be achieved using the tag `<jsp:getProperty>`.

Migrating WebSphere Extensions to OC4J

There are two ways to migrate JSPs that use any WebSphere-specific custom tags defined in the `tsx` tag library to OC4J.

- If there are many pages, and it is tedious to modify all the JSP files, you can use the WebSphere tag library and deploy it on on OC4J.

- Edit the JSP source files, using the OracleJSP tag library wherever possible.

Following are code examples showing how to migrate WebSphere JSP extensions to OC4J using the Oracle JSP Markup Language (JML) tag library.

`<REPEAT>` or `<tsx:repeat>` tag:

These tags are provided by WebSphere for looping over a HTML block a specified number of times, as an altenative to writing a Java "for" loop within a JSP page. The Oracle JML tag library has a `<jml:for>` tag with the same functionality. The syntax for this tag is:

```
<jml:for id = " loopVariable"
    from = "<%= jspExpression %>"
    to = "<%= jspExpression %>" >
    ... body of for tag (executed once at each value of range, inclusive)...
</jml:for>
```

which is similar to the WebSphere tsx:repeat:

```
<tsx:repeat index=name start=start_index end=end_index >
</tsx:repeat>
```

The differences are:

- The `id` variable in the `<jml:for>` tag holds the current value in the range and is local in scope to the tag, whereas the `index` variable is global in scope to the JSP page.

- `from` and `to` in `<jml:for>` are mandatory in OJSP. In WebSphere, `start` and `end` are optional.

5

Migrating Enterprise Java Beans

This chapter provides you with an overview of Sun Microsystems' Enterprise JavaBeans (EJB) architecture and its implementation in Oracle9*i*AS. In addition, the issues involved in migrating EJB components from WebSphere Advanced Edition to Oracle9*i*AS are presented.

This chapter contains these topics:

- Overview of Enterprise JavaBeans
- EJB Migration Considerations
- EJB Functionality and Components
- Differences between the EJB 1.0 and EJB 1.1 Specifications
- WebSphere 3.5.x Support for the EJB API
- Migrating EJB Applications from WebSphere to Oracle9iAS OC4J

Overview of Enterprise JavaBeans

Enterprise JavaBeans (EJB) is the standard server-side component architecture for developing and deploying object-oriented Java applications. It enables developers to quickly and easily build distributed applications.

A major goal of the EJB architecture is to provide component portability at both the source code level and the binary code level.

EJB components, called enterprise beans, are server-side components, written in Java, that typically contain the business logic of an application. The different types of enterprise beans are summarized in Table 5–1.

Table 5–1   Types of EJBs

Type of Enterprise Bean	Description
Session bean	A component created to provide a service on behalf of a single client; it lives only for the duration of a single client/server session.
Entity bean	A component representing data maintained in a data store; it can persist as long as the data it represents.

Although the EJB architecture does provide for component portability, certain implementation-specific aspects of an EJB component remain non-portable. These include:

- Deployment of enterprise beans

- Runtime support for deployed enterprise beans

- Container-managed persistence of entity beans

EJB Migration Considerations

One of the goals of the EJB initiative is to deliver component portability between different environments not only at source-code level, but also at a binary level, to ensure portability of compiled, packaged components. While it is true that EJB does offer an appreciable deal of portability, there are still a number of non-portable, implementation-specific aspects that need to be addressed when migrating components from one application server to another. Typically, an EJB component requires low-level interfaces with the container in the form of stubs and skeleton classes that will probably always need to be container implementation-specific. In effect, a clear partitioning between portable and non-portable elements of an EJB component can be drawn from the EJB 1.1 specification:

- Portable EJB elements include:

    - The actual component implementation classes and interfaces (bean class, and remote, and home interfaces).

    - The assembly and deployment descriptor that describes generic component properties such as JNDI names and transactional attributes.

    - Security attributes

- Implementation-specific elements include:

- Low-level helper implementation classes (stubs and skeletons) to interface with the host container.

- O/R mapping definitions for CMP entity beans, including search logic for custom finder methods that are declared in an implementation-specific format proprietary to each application server.

- Every component has a set of properties that require systematic configuration at deployment time. For example, mapping of security roles declared in an EJB component to actual users and groups is a task that is systematically performed at deployment-time, first, because mappings may not be known in advance, and secondly, because there are dependencies on the structure and population of the user directory on the target deployment server.

- There are issues specific to migration from WebSphere to Oracle9*i*AS OC4J that arise from different levels of EJB standards support, and WebSphere-specific extensions to APIs. WebSphere Advanced Edition 3.5.3 supports the EJB 1.0 specification, OC4J supports EJB 1.1 specifications, and some advanced features in the EJB 2.0 specification.

The following sections describe EJB specifications, session beans, and entity beans, transactions and concurrency, the WebSphere support for these APIs and WebSphere extensions, the difference between OC4J and WebSphere in the EJB containers and finally the migration path to OC4J.

EJB Functionality and Components

In brief, the goal of EJB technology surpasses the basic Java object model by integrating new functionality important for enterprise systems:

- Automatic management of object life cycle (instantiation, destruction, and activation)

- Object security: who can use which object and how

- Object persistence: how objects are stored on a long-term basis

- Transaction behavior of the objects

- Distribution: how can remote applications access objects

- Scalability: by implementing various technologies

From a developer's point of view, an EJB is presented as a group of files that brings together:

- Java classes
- Java interfaces,
- Deployment information,
- Metadata

The EJB Server

EJB servers manage low-level system resources, allocating resources to the containers as they are needed. The EJB Server hosts and provides a runtime environment for the EJB containers. Containers are transparent to the client—there is no client API to manipulate the container, and there is no way for a client to tell in which container an enterprise bean is deployed. However, the EJB container and the EJB servers are not clearly separated constructs. The EJB specification only defines a bean-container contract and does not define a container-server contract.

EJB container

The EJB container is a specialized service in which enterprise beans are deployed. EJB containers insulate the deployed enterprise beans from the underlying EJB server and provide a standard application programming interface (API) between the beans and the container. This specialized service manages their life cycle, transactions, security, naming, persistence, and so on, according to a specific contract and constrained model delineated by the EJB specification. To do this, the container uses the generic services provided by the server.

EJB Specification Roles

The Enterprise JavaBeans specification identifies the following roles that are associated with a specific task in the development of distributed applications.

Enterprise Bean Provider

Typically an expert in the application domain; for example, in the financial or telecommunications industry. The bean provider implements the business task without being concerned about the distribution, transaction, security, and other non-business aspects of the application.

Application Assembler

This is also a domain expert. The application assembler composes an application from various prefabricated building blocks (that is, enterprise beans) and adds

other components such as GUI clients, applets, and servlets to complete the application. While composing an application, an assembler is only concerned with the interfaces to enterprise beans, but not with their implementation.

Deployer

The deployer is specialized in the installation of applications. The deployer adapts an application, composed of a number of enterprise beans, to a target operation environment by modifying the properties of the enterprise beans. The deployer's tasks include, for example, the setting of transaction and security policies, specifying JNDI names by setting the appropriate properties in the deployment descriptor, and integration with enterprise management software.

EJB Server Provider

Typically a vendor with expertise in distributed infrastructures and services. The server provider implements a platform, which facilitates the development of distributed applications and provides a runtime environment for them. This role can also provide specialized containers that wrap a certain class of legacy applications or systems.

EJB Container Provider

An expert in distributed systems, transactions, and security. A container is a runtime system for one or multiple enterprise beans. It provides the glue between enterprise beans and the EJB server. A container can be both, prefabricated code as well as a tool that generates code specific for a particular enterprise bean. A container also provides tools for the deployment of an enterprise bean and hooks into the application for monitoring and management.

System Administrator

The system administrator is concerned with a deployed application. The administrator monitors the running application and takes appropriate action in the case of abnormal behavior of the application. The administrator ensures that the hardware and network hosting the application is maintained and serviceable for the duration of the application's availibility. Typically, an administrator uses enterprise management tools that are connected to the application by the deployer through the hooks provided by the container.

Session Beans

A session bean is an object that executes on behalf of a single client. The container creates the session bean instance in response to a remote task request from a client. A session bean has one client; in a sense, a session bean represents its client in the EJB server. Session beans can also be transaction-aware--they can update shared data in an underlying database but they do not directly represent the shared database data. The life of a session bean is transient and relatively short-lived. Typically, the session bean lives for as long as its client maintains the session "conversation." When the client terminates, the session bean is no longer associated to that client. A session bean is considered transient because the session bean instance is removed should the container crash, and the client must reestablish a new session object to continue.

There are two types of session beans: Stateful Session Beans (SFSB) and Stateless Session Beans (SLSB). Both of these beans must implement `javax.ejb.SessionBean`. However their life cycles are different within a EJB container.

Stateful Session Beans

A session bean typically maintains the state of the interaction or conversation with its client--that is, the session bean holds information about the client across method invocations and for the duration of the client session. A session bean that maintains its state is called a stateful session bean. When the client ends its interaction with the session bean, the session ends and the bean no longer maintains the state values.

The Life Cycle of Stateful Session Beans

A session bean's life cycle begins when a client invokes a `create()` method defined in the bean's home interface. In response to this method invocation, the container does the following:

1. Creates a new memory object for the session bean instance.

2. Invokes the session bean's `setSessionContext()` method. This method passes the session bean instance a reference to a session context interface that can be used by the instance to obtain container services and get information about the caller of a client-invoked method.

3. Invokes the session bean's `ejbCreate()` method corresponding to the `create()` method called by the EJB client.

Ready State

After a session bean instance is created, it moves to the ready state of its lifecycle. In this state, EJB clients can invoke the bean's business methods defined in the remote interface. The actions of the container in this state are determined by whether a method is invoked transactionally or non-transactionally:

Transactional Method Invocations

When a client invokes a transactional business method, the session bean instance is associated with a transaction. After a bean instance is associated with a transaction, it remains associated until that transaction completes. Furthermore, an error results if an EJB client attempts to invoke another method on the same bean instance and invoking that method causes the container to associate the bean instance with another transaction or with no transaction. The container then invokes the following methods:

1. The `afterBegin()` method if the session bean implements the `SessionSynchronization` interface.

2. The business method in the bean class that corresponds to the business method defined in the bean's remote interface and called by the EJB client.

3. The bean instance's `beforeCompletion()` method, if the session bean implements the `SessionSynchronization` interface.

The transaction service then attempts to commit the transaction, resulting either in a commit or a roll back. When the transaction completes, the container invokes the bean's `afterCompletion()` method (if the bean implements the `SessionSynchronization` interface), passing the completion status of the transaction (either commit or rollback) to the `afterCompletion()` method.

If a rollback occurs, a stateful session bean can roll back its conversational state to the values contained in the bean instance prior to beginning the transaction. Stateless session beans do not maintain a conversational state, so they do not need to be concerned about rollbacks.

Non-transactional Method Invocations

When a client invokes a nontransactional business method, the container simply invokes the corresponding method in the bean class.

Pooled State

The container has a sophisticated algorithm for managing which enterprise bean instances are retained in memory. When a container determines that a stateful session bean instance is no longer required in memory, it invokes the bean

instance's `ejbPassivate()` method and moves the bean instance into a reserve pool. A stateful session bean instance cannot be passivated (deactivated) when it is associated with a transaction.

If a client invokes a method on a passivated instance of a stateful session bean, the container activates the instance by restoring the instance's state and then invoking the bean instance's `ejbActivate()` method. When this method returns, the bean instance is again in the ready state.

Because every stateless session bean instance of a particular type is the same as every other instance of that type, stateless session bean instances are not passivated or activated. These instances exist in a ready state at all times until their removal.

Removal

The lifecycle of a stateful session bean ends when an enterprise bean client or the container calls a `remove()` method defined in the bean's home interface or remote interface. In response to this method invocation, the container calls the bean instance's `ejbRemove()` method. The container can end stateless session beans by this method, or it can pool them for later use.

A container can implicitly call a remove method on an instance after the lifetime of the EJB object has expired. The lifetime of a session EJB object is set in the deployment descriptor with the timeout attribute.

Stateless Session Beans

A session bean may also be a stateless session bean. A stateless session bean does not maintain information or state for its client. A client may invoke a method of a stateless session bean to accomplish some objective, but the bean will hold values in its instance variables only for the duration of the method call. The stateless session bean does not retain these values (or state) when the method completes. Thus, all instances of stateless session beans are identical except when they are in the midst of a method invocation. As a result, stateless session beans can support multiple clients. The container can maintain a pool of stateless bean instances, and it can assign any instance to any client.

The Life Cycle of a Stateless Session Bean

The stateless session bean's life cycle has two states:

- The does-not-exist state.

- The method-ready pool state.

When a bean instance is in the does-not-exist state, this means that it has not yet been instantiated. When a bean instance is instantiated by the container and is ready to serve client requests, it is in the method-ready pool state. The container moves a stateless session bean from the does-not-exist state to the method-ready pool state by performing the following three operations:

1. Invoke the `Class.newInstance()` method on the stateless bean class.

2. Invoke the `SessionBean.setSessionContext(SessionContext context)` method on the bean instance.

3. `ejbCreate()` method is invoked on the bean instance.

Entity Beans

An entity bean represents an object view of persistent data maintained in a domain model, as well as methods that act on that data. To be more specific, an entity bean maps to a record in your domain model. In a relational database context, one bean exists for each row in a table. A primary key identifies each entity bean. Entity beans are created by using an object factory `create()` method. Access to entity beans may be shared by more than one client--multiple clients can simultaneously access an entity bean. Entities access and update the underlying data within the context of a transaction so that data integrity is maintained. Entity beans are also implicitly persistent as an EJB object can manage its own persistence or delegate its persistence to its container. Based on the type of persistence in entity beans are divided into two types:

Container-managed Persistence (CMP) Entity Beans

Container Managed Persistence (CMP) allows developers to build EJB components without having to directly deal with persistence during development. For CMP Entity Beans, the EJB container is responsible for persisting the state of the entity beans and synchronization of instance fields within the persistence store (the database). This means that the container would, for example, manage both generating and executing SQL code to read and write to the database. Because it is container-managed, the implementation is independent of the data source. All container-managed fields need to be specified in the deployment descriptor for the persistence to be automatically handled by the container. CMP Entity Beans are wrappers for persistent data—commonly in the form of relational database tables—with additional support for transaction control and security.

Bean-managed Persistence (BMP) Entity Beans

For BMP Entity Beans, the entity bean is directly responsible for persisting its own state and the container does not need to generate any database calls. Each BMP EJB is responsible for storing and retrieving its own state from a backing store in response to specific "hook" messages (like `ejbLoad()` and `ejbStore()`) that are sent to it at appropriate times during its lifecycle. Consequently, this implementation is less adaptable than the previous one as the persistence needs to be hard-coded into the bean.

The Entity Beans Life Cycle

An entity bean is considered to be long-lived and its state is persistent. It lives as long as the data remains in the database, rather than for the life of the application or server process. An entity bean survives the crash of the EJB container. Once an enterprise bean is deployed into a container, clients can create and use instances of that bean as required. Within the container, instances of an enterprise bean go through a defined life cycle. The events in an enterprise bean's life cycle are derived from actions initiated by either the client or the container. The life cycle of entity beans has three states:

Does-not-exist State

At this stage, no instances of the bean exist. An entity bean instance's life cycle begins when the container creates that instance. After creating a new entity bean instance, the container invokes the instance's `setEntityContext()` method. This method passes to the bean instance a reference to an entity context interface that can be used by the instance to obtain container services and to retrieve information about the caller of the client-invoked method.

Pooled State

Once an entity bean instance is created, it is placed in a pool of available instances of the specified entity bean class. While the instance is in this pool, it is not associated with a specific `EJBObject`. Every instance of the same enterprise bean class in this pool is identical. While an instance is in this pooled state, the container can use it to invoke any of the bean's finder methods.

Ready State

When a client needs to work with a specific entity bean instance, the container picks an instance from the pool and associates it with the `EJBObject` initialized by the client. An entity bean instance is moved from the pooled to the ready state if there are no available instances in the ready state.

There are two events that cause an entity bean instance to be moved from the pooled state to the ready state:

- When a client invokes the `create()` method in the bean's home interface to create a new and unique entity of the entity bean class (and a new record in the data source). As a result of this method invocation, the container calls the bean instance's `ejbCreate()` and `ejbPostCreate()` methods. The new `EJBObject` is associated with the bean instance.

- When a client invokes a finder method to manipulate an existing instance of the entity bean class (associated with an existing record in the data source). In this case, the container calls the bean instance's `ejbActivate()` method to associate the bean instance with the existing `EJBObject`.

When an enterprise bean instance is in the ready state, the container can invoke the instance's `ejbLoad()` and `ejbStore()` methods to synchronize the data in the instance with the corresponding data in the data source. In addition, the client can invoke the bean instance's business methods when the instance is in this state. All interactions required to handle an entity bean instance's business methods in the appropriate transactional (or non-transactional) manner are handled by the container, unless the EJB developer wrote the bean to handle these interactions itself. When a container determines that an entity bean instance in the ready state is no longer required, it moves the instance to the pooled state. This transition to the pooled state results from either of the following events:

- When the container invokes the `ejbPassivate()` method.

- When the client invokes a `remove()` method on the `EJBObject` associated with the bean instance or on the EJB home object. When the `remove()` method is called, the underlying entity is removed permanently from the data source.

The state that an entity bean represents is shared and transactional. In contrast, if a session bean has state, it must be private and conversational.

Object-relational (O-R) Mapping and Persistence

The problem of persistence is complex, and many research projects are being carried out on this subject. One of the important points to remember is that at runtime, objects (as it happens, EJBs) are not isolated entities but referenced mutually. Therefore, the problem of persistence does not concern isolated objects, but complex object graphs. There are many questions to answer. How can an object graph in memory be projected on a disk and vice versa? How can synchronization problems between the graph on disk and the graph in memory be resolved? How do you go about loading in memory only the parts of the graph being used at a given

moment? How can the graph be saved in a relational database (object-relational mapping technologies)?

The EJB specification attempts to render container-managed persistence with a clean separation between an entity bean and its persistent representation. That is, a separation between the data logic methods (such as the logic in an entity bean to add two fields together) and JDBC. The reason this separation is valuable is that the persistent representation of an entity bean (such as changing from a relational database to an object database) can be modified without affecting the entity bean logic.

To achieve this clean separation, container-managed persistent entity bean classes must be written to be devoid of any JDBC or other persistence logic. The container then generates the JDBC by subclassing your entity bean class. The generated subclass inherits from your entity bean class. Thus, all container-managed persistent entity beans are each broken up into two classes: the superclass (which the user writes, and contains the entity bean data logic) and the subclass (which the container generates, and contains the persistence logic). With these two classes, a clean separation of entity bean logic and persistent representation is achieved. The actual entity bean is a combination of the superclass and the subclass.

EJB Transactions and Concurrency

A transaction is a set of statements that must be processed as a single unit. Transactions must have four properties recalled with the acronym ACID: Atomicity, Consistency, Isolation and Durability.

- Atomicity is "all or nothing" operation wherein, for example, a transaction consisting of a debit to one account and a credit to another is not committed unless both operations are successful. Typically, atomicity is provided by the database management system.

- Consistency reflects the state of the system. A system is always consistent based upon the state invariants and transactions give the opportunity to write code that checks the consistency of the state.

- Isolation keeps operations on shared data invisible across transactions.

- Durability guarantees the effect of completed transaction are permanent and are not lost.

The EJB specification describes the creation of applications that enforce transactional consistency on the data manipulated by the enterprise beans. However, unlike other specifications that support distributed transactions, the EJB specification does not require enterprise bean and EJB client developers to write any

special code to use transactions. Instead, the container manages transactions based on two deployment descriptor attributes associated with the EJB module and the enterprise bean. EJB application developers are freed to deal with the business logic of their applications.

A J2EE 1.2 compliant EJB container should support flat transactions, the most common kind of transctions. A flat transaction cannot have any child (nested) transactions. These are the only transaction types supported by EJBs.

The Java Transaction API(JTA)

The JTA APIs specifies start and end transactions.

```
interface javax.transaction.UserTransaction
{
  public abstract void begin();
  public abstract void commit();
  public abstract void rollback();
  public abstract void setRollbackOnly();
  public abstract int getStatus();
  public abstract void setTransactionTimeout(int);
}

interface javax.transaction.Status
{
  public static final int STATUS_ACTIVE;
  public static final int STATUS_MARKED_ROLLBACK;
  public static final int STATUS_PREPARED;
  public static final int STATUS_COMMITTED;
  public static final int STATUS_ROLLEDBACK;
  public static final int STATUS_UNKNOWN;
  public static final int STATUS_NO_TRANSACTION;
  public static final int STATUS_PREPARING;
  public static final int STATUS_COMMITTING;
  public static final int STATUS_ROLLING_BACK;
}
```

The JTA `UserTransaction` interface is actually an interface to the application server's transaction manager. It is the public API exposed by the transaction manager. To get a reference to this, you must look up the interface via JNDI, just like you use JNDI to lookup EJB homes, JDBC drivers, etc. The application server must publish the JTA under `"java:comp/UserTransaction"`.

```
Context ctx = new InitialContext(...);
  javax.transaction.UserTransaction userTran =
```

```
(javax.transaction.UserTransaction) PortableRemoteObject.narrow(
                    ctx.lookup("javax.transaction.UserTransaction"),
                    javax.transaction.UserTransaction.class);
```

Transaction Boundaries

Transaction boundaries mark the beginning and end of transactions. The application developer chooses the boundaries. The J2EE specification mentions three ways of controlling transactional boundaries: programmatically inside bean code (bean-managed transactions), programmatically from client code (client-managed transactions), and declaratively inside deployment descriptors (container-managed transactions).

Client-Managed Transactions

A Java client can use the `javax.transaction.UserTransaction` interface to explicitly demarcate transaction boundaries. The client program obtains the `javax.transaction.UserTransaction` interface using the JNDI API. The EJB specification does not imply that the `javax.transaction.UserTransaction` is available to all Java clients. The J2EE specification specifies the client environments in which the `javax.transaction.UserTransaction` interface is available.

Container-Managed Transactions (CMT)

Whenever a client invokes an enterprise bean, the container interposes on the method invocation. The interposition allows the container to control transaction demarcation declaratively through the transaction attributes set in the deployment descriptor.

For example, if an enterprise bean method is configured with the "Required" transaction attribute, the container behaves as follows: if the client request is not associated with a transaction context, the container automatically initiates a transaction whenever a client invokes an enterprise bean method that requires a transaction context. If the client request contains a transaction context, the container includes the enterprise bean method in the client transaction.

An entity bean must always be designed with container-managed transaction demarcation. For entity beans using container-managed persistence, transaction isolation is managed by the data access classes that are generated by the container provider's tools. The tools must ensure that the management of the isolation levels performed by the data access classes will not result in conflicting isolation level requests for a resource manager within a transaction.

Bean Managed Transactions (BMT)

The enterprise bean with bean-managed transaction demarcation must be a session bean. An instance that starts a transaction must complete the transaction before it starts another new transaction. A session bean can use the `EJBContext` and the `javax.transaction.UserTransaction` object to programmatically demarcate transactions. For session beans with bean-managed transaction demarcation, the bean code can specify the desirable isolation level programmatically in the enterprise bean's methods using the resource manager specific API. For example, the bean provider can use the `java.sql.Connection.setTransactionIsolation(...)` method to set the appropriate isolation level for database access.

For transactions, a session bean can either use container-managed transactions or bean-managed transactions. Entity beans must use container-managed transactions. Whether an enterprise bean uses bean-managed or container-managed transaction demarcation, the burden of implementing transaction management is on the EJB container and server provider.

Transaction Isolation and Concurrency

The transaction isolation attribute tells the container how to limit concurrent reads in a database. The EJB 1.1 specification removed the guidelines for managing transaction isolation levels for beans with container-managed transaction demarcation. But since bean deployers still require mechanisms to govern EJB concurrency, WebSphere continues to support it along with other mechanisms discussed in the next section.

Using CMP, different databases need different SQL statements while trying to acquire a read/write lock at the "database" level as opposed to optimistic/pessimistic concurrency or locking at the container/bean level. For example, MS-SQL Server needs a "SELECT ... AT ISLOLATION SERIALIZABLE", Oracle needs a "SELECT ... FOR UPDATE" as a method of acquiring 'locks' or in other words at a transaction isolation level 'Serializable' to prevent dirty/unrepeatable/phantom reads. Hence, it is difficult to use generic SQL clauses in conjunction with transactions and locks at the database level without resorting to vendor-specific clauses.

The need is for a simple time stamp/versioning mechanism in EJB 1.1 (even EJB 2.0 seems to imply that acquiring a read/write lock at the database level is up to the EJB vendor, which vendors may or may not provide). All that the time stamp and versioning do is compare versions when the client sends data over for modification. The reading could have been done by different clients in different transactions.

Hence, if another client tries to update the same data in the entity bean instance, the version numbers will not match if the data has been updated by another client, and an exception can be raised that effectively tells the client to 'refresh' the information, i.e. get the data again to see what might have changed since the client first requested it for modification. This is analogous to performing an "Update <table> set <fields> where <fields = fields_read_at_transaction_start>". The only difference is that the above technique works across transactions, i.e. it prevents a client from overwriting committed changes made by another client.

Session beans that use bean-managed transaction have transaction attributes associated with each method of the bean. The attribute value tells the container how it must manage the transactions that involve this bean. There are six different transaction attributes that can be associated with each method of a bean. This association is done at deployment time by the application assembler or deployer.

EJB supports distributed flat transactions. The distribution mechanism makes it possible to involve bean objects on multiple EJB servers or to update data in multiple databases in a single transaction. Every client method invocation on a bean is supervised by the bean's container, which makes it possible to manage the transactions according to the transaction attributes that are specified in the corresponding bean's deployment descriptor.

A particular transaction attribute can be associated with an entire bean and apply to all its methods or just to an individual method. The scope of a transaction is defined by the transaction context that is shared by the participating bean objects.

EJB Caching

EJB containers allow smart caching of entity beans, which allow some operations to occur in memory rather than at database level. Caching conserves system resources used in making a database connection by eliminating database accesses to unchanged data. There are three caching options available for the container in committing a transaction:

Option A : The container caches a readily available instance between transactions, which has explicit access to the state of the object in the persistent storage. That is, each instance of the EJB will be held in memory. This option is supported by WebSphere and OC4J but should only be used in a single node system. Neither WebSphere nor OC4J enforce this restriction, hence, it is the bean deployer's responsibility to ensure that this restriction is satisfied. This means that the beans using this option will only be used within a single container. It is thus the responsibility of all clients of that bean to always direct their requests to the one bean instance within that specific container.

Option B: The container caches the instance between transactions which does not have access to the persistent object state. This option is not supported by WebSphere nor OC4J.

Option C: The container does not cache the instance between transactions. An entity bean's state is read once per transaction at the beginning of each transaction- even if the value did not change from the last time it was read. The instance is returned to the pool after the transaction is completed. This is the default option supported by WebSphere and OC4J and should be used in multiple node configurations.

Differences between the EJB 1.0 and EJB 1.1 Specifications

This section summarazies differences between versions of the EJB specification.

Changes Specific to Entity Beans

The most obvious change in EJB 1.1 is its mandated support for entity beans which was optional in EJB 1.0 Specification. This support must be complete, embracing support for both container-managed and bean-managed persistent entities. The major changes to entity beans are listed below.

- Changes to Bean-managed Transactions (BMT) - The transaction attribute `TX_BEAN_MANAGED` has been removed from entity beans. This attribute is difficult to use because it requires explicit transactional control by the developer. Removing it from entity beans simplifies the EJB architecture. Stateful session beans, however, still retain the `TX_BEAN_MANAGED` option. In EJB 1.0, both stateful session beans and entity beans had the option of using BMT instead of CMT.

- Return types from the find methods for entity beans - In EJB 1.0, finder methods can return a single entity or a collection of entities. Finder methods that return a single entity return the entity's remote interface type; entities that return a collection use `java.util.Enumeration`. In EJB 1.1 a new return type has been added, `java.util.Collection`.

- Changes to `ejbCreate()` method signature - In the EJB 1.0 container, CMP and BMP have different values for `ejbCreate()` method: BMP entity beans return the unique identity of the bean, the primary key; CMP entity beans return void. The EJB 1.1 specification changes this so that both bean-managed and container-managed entities have to return the primary key type from the `ejbCreate()` methods.

- Changes to primary key classes - For BMP entity beans, EJB 1.1 states that the primary key class can be any valid Java RMI-IIOP type. In addition, the new specification requires the primary key class to implement the `Object.equals()` and `Object.hashCode()` methods to ensure that these methods evaluate properly when comparing keys and storing them in a `java.util.Hashtable`. The most significant change regarding primary keys is the option to defer their definition until deployment time. In other words, the primary key for an entity bean doesn't have to be defined by the developer, but can be left to the deployer. This is a significant departure from the previous specification, which required the bean developer to define the primary keys.

- The environment-naming context - Enterprise JavaBeans 1.0 provides one interface to the bean's environment, `EJBContext`. The `EJBContext` provides the bean class with an interface to the container, allowing the bean to discover and interact with aspects of its environment. The interface provides methods concerned with caller identity, transactions, and accessing environment properties.

Enterprise JavaBeans 1.1 introduces a new bean-container interface called the environment-naming context (ENC). The ENC is a JNDI name space that is specific to a bean type and its context at runtime. To simplify the bean-container interface, the ENC is made available, by default, when a JNDI context is created. The JNDI ENC enhances the bean-container contract by adding new functionality, but it doesn't completely replace the `EJBContext`. In EJB 1.1, the JNDI ENC and the `EJBContext` together represent the complete bean-container interface.

Example 5–1 is an example of an EJB 1.0 bean using the `EJBContext` to read an environment property used to validate a request (the comparison is applicable to both entity and session beans). Example 5–2 shows an EJB 1.1 bean using the new JNDI ENC to obtain an environment property to validate a request.

Example 5–1   Using the EJBContext

```
public class AccountBean implements EntityBean
{
  int id;
  double balance;
  EntityContext ejbContext;
  public void setEntityContext(EntityContext ctx)
  {
     ejbContext = ctx;
  }
  public void withdraw(Double withdraw)throws WithdrawLimitException
  {
```

```
      Properties props = ejbContext.getEnvironment();
      String value = props.getProperty("withdraw_limit");
      Double limit = new Double(value)
      if (withdraw.doubleValue() > limit.doubleValue())
         throw new WithdrawLimitException(limit);
      else
         balance = balance - withdraw.doubleValue();
  }
...
}
```

Example 5–2   Using the environment properties in EJB 1.0

```
public class AccountBean implements EntityBean {
  int id;
  double balance;
  EntityContext ejbContext;
  public void setEntityContext(EntityContext ctx)
  {
     ejbContext = ctx;
  }
  public void withdraw(Double withdraw) throws WithdrawLimitException
  {
     InitialContext jndiContext = new InitialContext();
     Double limit = (Double)
                    jndiContext.lookup("java:comp/env/withdraw_limit");
     if (withdraw.doubleValue() > limit.doubleValue())
                               throw new WithdrawLimitException(limit);
     else
        balance = balance - withdraw.doubleValue();
  }
...
}
```

In EJB 1.0, environment properties are limited to String types and are available through the EJBContext. In EJB 1.1, environment properties can be of type String or any one of the primitive numerical wrappers (integer, long, double, boolean, byte, and float); they are available through a default JNDI context. Why the change? EJB 1.1 wanted to extend the bean-container contract to address many of the issues that would have mandated complicated changes to the EJBContext interface. To avoid the limitations of EJBContext -- its definition is fixed and therefore limited -- the JNDI ENC was introduced, which provides a more dynamic and extensible bean-container interface. The EJBContext still exists with

some changes, but most of the new EJB 1.1 features are realized through the JNDI ENC.

WebSphere 3.5.x Support for the EJB API

WebSphere supports EJB 1.0 specification with some additional features. WebSphere supports:

- Container-managed persistence (CMP)
- Uses `javax.transaction.UserTransaction`
- Restricts bean-managed transactions to session beans (as required by EJB 1.1 specification)

WebSphere Advanced Edition 3.5.3 does not support these EJB 1.1 features:

- XML deployment descriptors
- Use of JNDI within EJB environment:
    - Lookup of home interfaces via EJB references and links defined in the EJB's environment
    - The new `HomeHandle` class and the associated API changes to `EJBHome`
- Use of the `javax.security.Principal` interface

WebSphere and VisualAge for Java extend the EJB specification with the following features:

- Access beans: Simplify client application using EJBs
- Association: Support relationship between CMP beans
- Inheritance: Support polymorphism and reuse

Read-only Methods

The EJB specification does not provide a standard mechanism to let the container check if the bean's state has changed within a unit of work. The specification assumes that all beans accessed during a transaction are "dirty," and must have their state written back to the persistent store at the end of a transaction. WebSphere provides an extension to the EJB specification with the const method flag in the deployment descriptor of entity beans. It lets the developer tell the container which methods are const or read-only—in other words, it doesn't change the state of the

bean. For these methods EJB container does not call `ejbLoad` at the end of the method call.

EJB Finder-Helper Interface

WebSphere uses a concept called a "FinderHelper" to define the finder logic for CMP entity beans.The following finder logic is required for each finder method (other than the `findByPrimaryKey` method) contained in the home interface of an entity bean with CMP:

- The logic must be defined in a public interface named `<Name>BeanFinderHelper`, where `<Name>` is the name of the enterprise bean (for example, `AccountBeanFinderHelper`).

- The logic must be contained in a `String` constant named `<findMethodName>QueryString`, where `<findMethodName>` is the name of the finder method. The `String` constant can contain zero or more question marks (?) that are replaced from left to right with the value of the finder method's arguments when that method is invoked.

Example 5–3   Finder Helper Interface

```
Public interface AccountBeanFinderHelper
{
   String findLargeAccountsQueryString ="select * from ejb.accountbeantblwhere balance > ?";
}
```

This file contains one static `java.lang.String` field for each finder method declared in the EJB home interface. The strings are initialized with SQL queries executed dynamically when bean instances are retrieved in a finder method. Note that this file is specific to the WebSphere application server.

CMP in WebSphere

The CMP model in WebSphere allows a set of entity EJBs to be read from a relational database in the `findXXX()` method with only single SQL SELECT call. This is much more efficient then BMP case, which requires N+1 SQL calls to accomplish the same task.

Transactions

WebSphere supports two-phase commits for distributed transactions for Oracle , DB2, and Sybase and MQ Series. Distributed transaction support is also provided for Oracle and Microsoft SQL Server using the Merant drivers in addition to the

existing support for DB2 and Sybase. WebSphere 3.5.3 also supports distributed transactions over EJBs and JMS.

WebSphere never passivates an active bean, i.e. a bean participating in a transaction. WebSphere will throw a ROLLBACK exception back to the client.

WebSphere also makes the `UserTransaction` interface available to Java clients including servlets, JSPs, and standalone programs.

EJB Inheritance

WebSphere provides EJB inheritance similar to Java class inheritance.This EJB inheritance model is specific to the IBM EJB development environment. This is an extension of the EJB specification. In EJB inheritance, an enterprise bean inherits properties, such as CMP fields and association roles, methods, and method-level control descriptor attributes from another enterprise bean that resides in the same EJB group.

Distributed exceptions

Support for chaining distributed exceptions is provided by the `com.ibm.websphere.exception` Java package. The following classes and interfaces make up this package

- `DistributedException`

- `DistributedExceptionEnabled`

- `DistributedExceptionInfo`

- `ExceptionInstantiationException`

Access beans

An access bean adapts an enterprise bean to the JavaBeans programming model by hiding the home and remote interfaces from the access bean user (that is, an EJB client developer). This is specific to IBM WebSphere environment. These access beans are packaged in `com.ibm.ivj.ejb.access`. There are three types of access beans:

- A Java bean wrapper

  This facilitates either a session or entity bean to be used like a standard Java bean. It hides the enterprise bean home and remote interfaces from you. Each

Java bean wrapper that you create extends the
`com.ibm.ivj.ejb.access.AccessBean` class.

- Copy helper

  A copy helper is similar to Java bean wrapper, but it also incorporates a single copy helper object that contains a local copy of attributes from a remote entity bean. A user program can retrieve the entity bean attributes from the local copy helper object that resides in the access bean, which eliminates the need to access the attributes from the remote entity bean.

- Rowset

  A rowset access bean has all of characteristics of both the Java bean wrapper and copy helper access beans. However, instead of a single copy helper object, it contains multiple copy helper objects. Each copy helper object corresponds to a single enterprise bean instance.

Associations between enterprise beans

WebSphere supports one-to-one and one-to-many associations for CMP beans. The generated code is specific to WebSphere and VisualAge for Java environment specific.

In the EJB 1.1 specification, the application would have been required to throw an `EJBException`, but this has not yet been implemented in WebSphere Advanced Edition 3.5.3.

Migrating EJB Applications from WebSphere to Oracle9*i*AS OC4J

There are four aspects of an EJB that differ between WebSphere and Oracle9*i*AS OC4J:

- EJB code - The actual bean code, which is mainly well defined in the EJB 1.0 specification and should be compatible across implementations, with some minor exceptions which we will discuss.

- EJB-Java archive (JAR) - The EJB-Java Archive (JAR) file, which is also defined in the standard. This is designed to be a "binary" standard in the sense that EJB-JARs developed to the EJB specification are meant to be deployable on any EJB platform. The EJB-JAR file consists of the implementation and interface classes, a manifest and the serialized deployment descriptor `.ser` file in WebSphere 3.5.

- Client code - The client code is mainly defined and compatible between implementations with some minor exceptions. The deployment process differs between EJB implementations.

- Deployment process - The deployment process takes EJB code, packages it as an EJB-JAR file, creates support code (e.g. stubs and ties) so that it can be hosted in a particular EJB container and installs it into the server. When the JAR file is finally deployed, it has non-standard code in it. Note that different vendors have varying views on exactly what the "deployment" process is.

The following features cannot be migrated to OC4J:

- Access beans

- Object-level Trace (OLT) package of WebSphere

- XA-transactions across EJB, JMS, and MQ Series.

EJB Code Changes

In general, enterprise beans written to version 1.0 of the EJB specification are mostly compatible with version 1.1. However, you need to modify or recompile enterprise bean code in the following cases:

Changes specific to transaction API:

- IBM-specific transaction packages

  Any EJB application code still using the following IBM specific transaction support packages for user transactions need to be migrated. This involves finding the import statements in the EJB source files containing

  ```
  import com.ibm.db2.jdbc.app.jta.javax.transaction.*
  ```

  and changing it to:

  ```
  import javax.transaction.*
  ```

- The bean uses the `javax.jts.UserTransaction` interface.

  The package name of the `javax.jts interface` has changed to `javax.transaction`, and there have been minor changes to the exceptions thrown by the methods of this interface. An enterprise bean that uses the `javax.jts.UserTransaction` interface needs to be modified to use the new name `javax.transaction.UserTransaction`.

- `UserTransaction` object lookup

The UserTransaction object is obtained differently for enterprise beans written to version 1.1 of the EJB specification. Under WebSphere version 1.0, it was obtained as:

```
initialContext.lookup("jta/UserTransaction")
```

This should be changed to the following in OC4J:

```
initialContext.lookup("java:comp/UserTransaction")
```

- The bean uses the getCallerIdentity() or isCallerInRole(Identity identity) or getEnvironment() methods of the javax.ejb.EJBContext interface.

  These methods of EJBContext class were deprecated in EJB 1.1 because the class java.security.Identity is deprecated in the Java 2 platform. While OC4J still provides backward compatibility for implementation of these methods, Oracle recommends moving to standard methods.

Table 5–2   Deprecated methods in EJB 1.1

Method	Replace with	Description
getCallerIdentity()	Principal getCallerPrincipal()	getCallerIdentity() is deprecated in EJB 1.1. The container is allowed to always return null from this method. The enterprise bean should use the getCallerPrincipal() method instead.
getEnvironment()	The JNDI naming context java:comp/env	If the enterprise bean has no environment properties, this method returns an empty java.util.Properties object. This method never returns null.
CallerInRole(java.security. Identityrole())	boolean isCallerInRole(String roleName)	

- An entity bean with container-managed persistence needs to be recompiled

  The return value of ejbCreate() is different in EJB 1.1 than in EJB 1.0. A CMP bean written to the EJB 1.0 specification needs to be recompiled to work with EJB 1.1 compliant OC4J. The ejbCreate() method has different return values

for BMP and CMP entity beans in EJB 1.0 specification. BMP entity beans return the unique identity of the bean, which is the primary key. The CMP Entity Bean returns `void`.

The following code shows the different method signatures used for container-managed and bean-managed `ejbCreate()` methods in an EJB 1.0. CMP entity bean:

```
public class EmployeeCMP implements javax.ejb.EntityBean
{
  public int Empid;
  public void ejbCreate(int empID)
  {
     Empid= empID;
  }
  // more bean code follows
  ....
}
```

The code below shows a BMP entity bean, EJB 1.0:

```
public class EmployeeBMP implements javax.ejb.EntityBean
{
  public int Empid;
  public EmployeePK ejbCreate(int myID)
  {
     Empid= empID;
     // do a database insert using JDBC
     EmployeePK pk = new AccountPK(empID);
     return pk;
  }
  // more bean code follows
  ...
}
```

The EJB 1.1 specification changes this so that both bean-managed and container-managed entities have to return the primary key type from the `ejbCreate()` methods. However, container-managed beans are required to return `null` instead of a primary key object.

■ CMP EJB finder methods

The OC4J server puts finder queries in XML deployment descriptors using proprietary query language formats, as compared to the WebSphere finder helper classes. When deploying CMP entity beans you can ignore finder helper classes and replace them by manually editing queries into the OC4J deployment

descriptor `orion-ejb-jar.xml`. For example, the following finder helper in WebSphere EJB can be migrated to OC4J as follows:

In WebSphere:

```
public interface AccountBeanFinderHelper
{
  String findLargeAccountsQueryString ="select * from ejb.accountbeantbl
                                              where balance > ?";
}
```

To migrate this functionality to OC4J, change the generated `orion-ejb-jar.xml` by modifying the `<finder-method/>` tag section with:

```
public static final String findByOwner_query="full: select "+
                  "*"+"from from ejb.accountbeantbl"+ "where balance >$1";
```

Note that this query uses "`full:`" at the beginning to specify that this is the full text of the query.

■ Read-only entity bean methods

As pointed out earlier, WebSphere provides an option in its Administration Console to indicate to the EJB container that a particular method is read-only. OC4J uses a runtime approach to provide the same feature. In order to migrate EJBs containing read-only methods you must:

1. Define a method "`public boolean isModified()`" in the bean code.

2. Ensure that the method returns a `boolean` flag that is set each time the bean is modified. This flag should be cleared in `ejbStore()`.

   If `isModified()` returns `false`, OC4J knows that it does not need to talk to the database.

Client Level Code Changes

The following code changes are necessary:

■ Client managed transactions - The EJB programming model allows client applications and enterprise beans to create transactions (through the `javax.jts.UserTransaction` interface) to bound related method invocations into a single atomic unit of work. The EJB specification does not, however, define how a client application can create a transaction; it does not specify how to obtain a reference to the `javax.jts.UserTransaction`

interface in a client application. In the absence of this specification, the application server provides a class that may be used to access the JTS CosTransactions . This is the current interface, which is a superset of the `javax.jts.UserTransaction` interface. This interface allows transactions to be created, suspended and resumed, and committed or aborted. To obtain a reference to the `org.omg.CosTransactions.Current` interface, the client application may simply call the static `getCurrent()` method in the `com.ibm.ejs.client.EJClient` class. For example:

```
org.omg.CosTransactions.Current current
                           =com.ibm.ejs.client.EJClient.getCurrent();

current.begin();
// do transactional work
current.commit(false);
```

OC4J does not support client-managed transactions. Any client-managed transactions can be deployed as container managed transactions by changing the transaction properties in the EJB deployment descriptors.

- JNDI lookup - To lookup an EJB from a client code, you need to create the JNDI initial naming context `InitialContext` object. This object will be hooked to the naming service running in the application server based on two parameters: host name and port of the naming service (`PROVIDER_URL`) and the name of the initial context factory (`INITIAL_CONTEXT_FACTORY`). This information is provided to the constructor of the `InitialContext` as a `java.util.Properties` object. When migrating from WebSphere to OC4J, these values must be changed. For example:

  In WebSphere, the initial naming context is obtained by the following code:

  ```
  java.util.Properties properties =new java.util.Properties();
  properties.put(javax.naming.Context.PROVIDER_URL,"iiop:///");
  properties.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
                        "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
  javax.naming.InitialContext initialContext =
                          new javax.naming.InitialContext(properties);
  initialContext.lookup("com/ibm/Hello");
  ```

  The string "`///`" for the `PROVIDER_URL` property indicates to the runtime environment to find the naming service at the standard port on the local machine. The general form of the `PROVIDER_URL` string is "`iiop://hostname:port/`". The value of the second property, `INITIAL_CONTEXT_FACTORY`, is the class name of the naming service factory.

  To migrate to OC4J, this code must be changed as follows:

■ If the client and the EJB are deployed on the on the same OC4J instance, the JNDI properties (initial context factory, location, security parameters) are not needed):

```
javax.naming.InitialContext initialContext =
                                    new javax.naming.InitialContext();
Object yourEJBHome = initialContext.lookup("java:comp/env/ejb/Hello")
```

If the client and EJBs are deployed on different OC4J instances:

```
java.util.Properties properties = new java.util.Properties();
properties.put(javax.naming.Context.PROVIDER_URL,
                        "ormi://<hostname>:<port>/<application>");
properties.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
                "com.evermind.server.rmi.initialContextFactory");
properties.put(javax.naming.Context.SECURITY_PRINCIPAL,"admin");
properties.put(javax.naming.Context.SECURITY_CRIDENTIALSL,"123");
javax.naming.InitialContext initialContext = new
                            javax.naming.InitialContext(properties);
Object yourEJBHome = initialContext.lookup("Java:comp/env/ejb/Hello")
```

The `SECURITY_PRINCIPAL` and `SECURITY_CREDENTIAL` properties are the administrator username and password specified during installation of OC4J.

Changes in Transactional Semantics

The following should be noted:

■ `TRANSACTION_SERIALISABLE` has different semantics in OC4J and WebSphere. In WebSphere, if two transactions both read the same row (entity), they acquire a read lock on that row.

■ Read-only methods - The EJB specification does not provide a standard mechanism to allow a container to check if the bean's state has changed within a unit of work. As an IBM extension to the EJB specification, VisualAge for Java allows the bean developer to define the method types (read-only or update). With the Const Method checkbox in the control descriptor of the bean's Properties panel (Method screen), a selected checkbox means read-only, and non-checked is an update method. When deployed on WebSphere, the container runtime only stores a bean's state if at least one non read-only (update) method has been invoked during a transaction. This eliminates unnecessary SQL UPDATE queries at commit time, for example, for get operations that do not change the entity state. When an update (non read-only)

method is invoked on a bean and the bean is not active, the container runtime will load the bean using the Persister `loadForUpdate` method. This method will acquire a lock on the bean's state in the underlying data store.

- OC4J provides a complete implementation of the JTA 1.0.1 specification. Oracle9*i*AS Release 2 integrates the JTA facilities with a middle-tier Distributed Commit Coordinator based on the Oracle Database's Commit Coordinator that provides highly scalable one-phase commit and two-phase commit facilities across Oracle and non-Oracle XA compliant Resource Managers.

Object-relational (O-R) Mapping

With EJBs, O-R mapping is done differently according to the EJB container used (in practice, according to the application server).This means that in order to import an EJB into a different container, it is necessary to go through this development phase again. This is the case with WebSphere and OC4J, since they have different ways of generating O-R mapping classes. WebSphere uses the WebSphere Studio and the VisualAge for Java development environment to generate code for O-R mapping. WebSphere supports one-to-one and one-to-many associations among EJBs.

OC4J supports the EJB 2.0 O-R mapping model. EJB 2.0 is not backward compatible with EJB 1.1, rather, it is a complete break with the old way of doing CMP persistence. OC4J supports one-to-one and one-to-many associations.

Deployment of EJBs

Since WebSphere 3.5.x is at the EJB 1.0 specification level, the EJB JAR files are deployed using serialized deployment descriptors. These cannot be directly deployed on OC4J, which supports the EJB 1.1 XML-based deployment descriptors. Consequently, the EJB JAR files must be re-archived using the EJB 1.1 deployment descriptors. Package individual or multiple EJB components in EJB JAR files and place assembly and deployment properties for EJBs in the standard deployment descriptor XML file (`ejb-jar.xml`) within the EJB archive file. The deployment descriptor contains attribute and environment settings that define how the container invokes enterprise bean functionality. Every enterprise bean (both session and entity) must have a deployment descriptor that contains settings for the following attributes:

- JNDI home name attribute - Specifies the Java Naming and Directory Interface (JNDI) home name that is used to locate instances of an EJB home object

- Transaction attribute - To define the transactional manner in which the container invokes a method

- Transaction isolation level attribute - To define the degree to which transactions are isolated from each other by the container

- Access control attribute - To define an access control entry that identifies users or roles that are permitted to access the methods in the enterprise bean. This value is not used by the WebSphere EJB servers.

- `RunAsMode` and `RunAsIdentity` attributes - the `RunAsMode` attribute defines the identity used to invoke the method. If a specific identity is required, the `RunAsIdentity` attribute is used to specify that identity. The `RunAsMode` attribute is used by the WebSphere EJB servers; the `RunAsIdentity` attribute is not.

The deployment descriptor for an entity bean must also contain settings for the following attributes. These attributes can be set on the bean only. They cannot be set on a per-method level.

- Primary key class attribute - identifies the primary key class for the bean.

- Container-managed fields attribute - lists those persistent variables in the bean class that the container must synchronize with fields in a corresponding data source to ensure that this data is persistent and consistent.

- Reentrant attribute - specifies whether an enterprise bean can invoke methods on itself or call another bean that invokes a method on the calling bean. Only entity beans can be reentrant. The deployment descriptor for a session bean must also contain settings for the following attributes. These attributes can be set on the bean only; they cannot be set on a per-method level.

- State management attribute - defines the conversational state of the session bean. This attribute must be set to either `STATEFUL` or `STATELESS`.

- Timeout attribute - defines the idle timeout value in seconds associated with this session bean.

These attributes can be set for the entire enterprise bean or for the individual methods in the bean. The container uses the definition of the bean-level attribute unless a method-level attribute is defined, in which case the latter is used.

```
<container-transaction>
  <method>
     <ejb-name>LogEntEJB</ejb-name>
     <method-name>*</method-name>
     <trans-attribute>Required</trans-attribute>
  </method>
  <method>
     <ejb-name>EntUtenteEJB</ejb-name>
```

```
            <method-name>*</method-name>
            <trans-attribute>Required</trans-attribute>
        </method>
    </container-transaction>
```

OC4J EJB Container Setting

To cache EJB instances, you specify maximum instance limits for each entity bean in `orion-ejb-jar.xml` with your application and place it in the `META_INF` directory.

```
<?xml version="1.0"?>
<orion-ejb-jar>
  <enterprise-beans>
     <entity-deployment name="BeanName" location="BeanName"
           max-instances="5" validity-timeout="3600000"/>
  </enterprise-beans>
</orion-ejb-jar>
```

6

Migrating JDBC Applications

This chapter introduces the JDBC (Java Database Connectivity) API and describes how to connect to, and access data from, a database with WebSphere Advanced Edition 3.5.3. It also discusses ways of migrating WebSphere applications to Oracle Containers for J2EE (OC4J). The sections in this chapter are:

- The JDBC API

- Database Drivers

- Connection Pooling

- IBM Extensions

The JDBC API

The JDBC API enables Java programs to create sessions, execute SQL statements, and retrieve results from relational databases, providing vendor-independent access to relational data. The JDBC specification delivers a call-level SQL interface for Java that is based on the X/Open SQL call level interface specification.

The JDBC API consists of four major components: JDBC drivers, connections, statements, and a result set. Database vendors deliver only the driver, which should comply with JDBC specifications (for a complete description, see "Database Drivers" on page 6-2). The connection, statement, and result set components are in the JDBC API package (that is, the `java.sql` package).

The JDBC API provides interface classes for working with these components:

- The `java.sql.Driver` and `java.sql.DriverManager` for managing JDBC drivers

- The `java.sql.Connection` for using connections

- The `java.sql.Statement`, for constructing and executing SQL statements
- The `java.sql.ResultSet` for processing the results

The JDBC 2.0 API includes many new features in the `java.sql` package as well as the new Standard Extension package, `javax.sql` Features in the `java.sql` package include support for SQL3 data types, scrollable result sets, programmatic updates, and batch updates.

The new JDBC standard extension APIs, an integral part of Enterprise JavaBeans (EJB) technology, allows you to write distributed transactions that use connection pooling and connect to virtually any tabular data source, including files and spreadsheets.

When you write a JDBC application, the only driver-specific information required is the database URL. You can build a JDBC application so that it derives the URL information at runtime. Using the database URL, a user name, and password, your application first requests a `java.sql.Connection` from the `DriverManager`.

A typical JDBC program follows this process:

1. Load the database driver, using the driver's class name
2. Obtain the connection, using the JDBC URL for connection
3. Create and execute statements
4. Use result sets to navigate through the results
5. Close the connection.

Database Drivers

JDBC defines standard API calls to a specified JDBC driver, a piece of software that performs the actual data interface commands. The driver is considered the lower level JDBC API. The interfaces to the driver are database client calls, or database network protocol commands that are serviced by a database server.

Depending on the interface type, there are four types of JDBC drivers that translate JDBC API calls:

- Type 1, JDBC-ODBC bridge: Translates calls into ODBC API calls.
- Type 2, Native API driver: Translates calls into database native API calls. As this driver uses native APIs, it is vendor dependent. The driver consists of two parts: a Java language part that performs the translation and a set of native API libraries.

- Type 3, Network Protocol: Translates calls into DBMS-independent network protocol calls. The database server interprets these network protocol calls into specific DBMS operations.

- Type 4, Native Protocol: Translates calls into DBMS native network protocol calls. The database server converts these calls into DBMS operations.

The `DriverManager` Class

Using different drivers, a Java program can create several connections to several different databases. To manage driver operations, JDBC provides a driver manager class, the `java.sql.DriverManager`, which loads drivers and creates new database connections.

Registering JDBC Drivers

The `DriverManager` registers any JDBC driver that is going to be used. If a Java program issues a JDBC operation on a non-registered driver, JDBC raises a "No Suitable Driver" exception.

There are several ways to register a driver:

- Register the driver explicitly by using

  ```
  DriverManager.registerDriver(driver-instance)
  ```

  where *driver-instance* is an instance of the JDBC driver class.

- Load the driver class by using

  ```
  Class.forName(driver-class)
  ```

  where *driver-class* is the JDBC driver class.This loads the driver into the Java Virtual Machine. When loaded, each driver must register itself implicitly by using the `DriverManager.registerDriver` method.

  For example, to register the DB2 JDBC Type 2 driver in the `COM.ibm.db2.jdbc.app` package, you can use either:

  ```
  DriverManager.registerDriver(new COM.ibm.db2.jdbc.app.DB2Driver());
  ```

  or

  ```
  Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
  ```

  For an Oracle database:

```
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver ());
```

or

```
Class.forName("oracle.jdbc.driver.OracleDriver")
```

A J2EE server implicitly loads the driver based on the JDBC driver configuration, so no client-specific code is needed to load the driver. The JNDI (Java Naming and Directory Interface) tree provides the datatsource object reference.

WebSphere Advanced Edition 3.5.3 supports DB2, Informix, Microsoft SQL Server, Oracle, Sybase, Versant, and others. However, WebSphere does not support the Oracle thick JDBC driver. To use the Oracle thick JDBC driver, configure the data sources in Oracle Containers for J2EE (OC4J), as described in "Configuring Data Sources". OC4J will automatically load the driver classes during server startup.

> **Note:** If you use the Type 3 JDBC driver
> (`COM.ibm.db2.jdbc.app.DB2Driver` - `jdcb:db2:DBNAME`),
> you must install the DB2 CAE (Client Application Enabler) and
> then catalog the remote database. OC4J will treat the cataloged
> database as a local database.

The `DataSource` Class

The JDBC 2.0 specification introduced the `java.sql.Datasource` class to make the JDBC program 100% portable. In this version, the vendor-specific connection URL and machine and port dependencies were removed. This version also discourages using `java.sql.DriverManager`, `Driver`, and `DriverPropertyInfo` classes. The data source facility provides a complete replacement for the previous JDBC `DriverManager` facility. Instead of explicitly loading the driver manager classes into the runtime of client applications, the centralized JNDI service lookup obtains the `java.sql.Datasource` object. The `Datasource` object can also be used to connect to the database.

According to the JDBC 2.0 API specification, a data source is registered under the JDBC subcontext or one of its child contexts. The JDBC context itself is registered under the root context. A `Datasource` object is a connection factory to a data source. WebSphere and OC4J both support the JDBC 2.0 `DataSource` API.

Configuring Data Sources

In WebSphere, you configure data sources using the Administrative Console to specify the data source name, database name, and JDBC URL string. This information is stored in a repository database.

OC4J uses flat files to configure data sources for all of its deployed applications. data sources are specified in the following descriptor file:

UNIX:
`<ORACLE_HOME>/j2ee/home/config/data-sources.xml`

NT:
`<ORACLE_HOME>\j2ee\home\config\data-sources.xml`

Following is a sample data source configuration for an Oracle database. Each data source in `data-sources.xml` (xa-location, ejb-location and pooled-location) must be unique.

```
<data-source
class="com.evermind.sql.DriverManagerDataSource"
name="Oracle"
url="jdbc:oracle:thin@dlsun2058.us.oracle.com:1521:orcl"
xa-location="jdbc/xa/OracleXADS"
ejb-location="jdbc/OracleDS"
pooled-location="jdbc/OraclePoolDS"
connection-driver="oracle.jdbc.driver.OracleDriver"
username="scott"
password="tiger"
schema="database-schemas/oracle.xml"
inactivity-timeout="30"
max-connections="20"
/>
```

Table 6–1 describes all of the configuration parameters in `data-sources.xml`. (Not all of the parameters are shown in the example above).

Table 6–1 `data-sources.xml` file

Parameter	Description
class	Class name of the data source.
connection-driver	Class name of the JDBC driver
connection-retry-interval	Number of seconds to wait before retrying a failed connection. The default is 1.

Table 6–1 `data-sources.xml` file

Parameter	Description
ejb-location	JNDI path for binding an EJB-aware, pooled version of this data source; this version will participate in container-managed transactions. This is the type of data source to use from within EJBs and similar objects.
	This parameter only applies to a ConnectionDataSource.
inactivity-timeout	Number of seconds unused connections will be cached before being closed.
location	JNDI path for binding this data source.
max-connect-attempts	Number of times to retry a failed connection. The default is 3.
max-connections	Maximum number of open connections for pooling data sources.
min-connections	Minimum number of open connections for pooling data sources. The default is zero.
name	Displayed name of the data source.
password	User password for accessing the data source (optional).
pooled-location	JNDI path for binding a pooled version of this data source. This parameter only applies to a ConnectionDataSource.
schema	Relative or absolute path to a database-schema file for the database connection.
source-location	Underlying data source of this specialized data source.
url	JDBC URL for this data source (used by some data sources that deal with java.sql.Connections)
username	User name for accessing the data source (optional).
wait-timeout	Number of seconds to wait for a free connection if all connections are used. Default is 60.
xa-location	JNDI path for binding a transactional version of this data source. This parameter only applies to a ConnectionDataSource.
xa-source-location	Underlying XADataSource of this specialized data source (used by OrionCMTDataSource)

Note that WebSphere does not support subcontexts. For example, you cannot specify xa/OracleXADS, where xa is subcontext under the JDBC context. Morever, in WebSphere, the JDBC context is implicit, and you don't specify it (as you specify

it explicitly for OC4J, in `data-sources.xml`). However, both WebSphere and OC4J automatically bind the data sources for you.

Configuring OC4J with DB2 Database

If you are using DB2 as your database, you need to create an additional file, `db2.xml`, in the following directory to define DB2 as a data source:

UNIX:
<ORACLE_HOME>/OC4J/j2ee/home/config/database-schema

NT:
<ORACLE_HOME>\OC4J\j2ee\home\config\database-schema

Below is an example of the schema file `db2.xml`:

```xml
<?xml version="1.0"?>
<!DOCTYPE database-schema PUBLIC "-//Evermind//- Database schema"
"http://www.orionserver.com/dtds/database-schemas.dtd">
<database-scheme name="DB2" not-null="not null" null="default null"
  primary-key="primary key">
  <type-mapping type="java.lang.String" name="varchar(255)" />
  <type-mapping type="int" name="integer" />
  <type-mapping type="long" name="bigint" />
  <type-mapping type="float" name="double" />
  <type-mapping type="double" name="double" />
  <type-mapping type="byte" name="smallint" />
  <type-mapping type="char" name="smallint" />
  <type-mapping type="short" name="smallint" />
  <type-mapping type="boolean" name="char(1)" />
  <type-mapping type="java.util.Date" name="timestamp" />
  <type-mapping type="java.io.Serializable" name="blob(1 M)" />
  <disallowed-field name="add" />
  <disallowed-field name="admin" />
  <disallowed-field name="wvarchar" />
</database-scheme>
```

The following is an example of a corresponding `data-sources.xml` file with the `db2.xml` file specified:

```xml
<data-source
name="Default data-source"
class="com.evermind.sql.ConnectionDataSource"
location="jdbc/DefaultDS"
pooled-location="jdbc/DefaultPooledDS"
xa-location="jdbc/xa/DefaultXADS"
```

```
ejb-location="jdbc/DefaultEJBDS"
url="jdbc:db2:dbTest"
connection-driver="COM.ibm.db2.jdbc.app.DB2Driver"
username="myUserName"
password="myPwd"
inactivity-timeout="30"
schema="database-schemas/db2.xml"
/>
```

Obtaining a Data Source Object

Obtaining a data source object involves binding to the JNDI initial context and doing a lookup for the subcontext jdbc/sampleDB. To do this, you have to get a handle to the intial context javax.naming.InitialContext. IntialContext is the root context of the JNDI namespace. InitialContext has two constructors:

- A default constructor that takes no parameters

- A constructor that takes one parameter, java.util.Properties or java.util.HashTable

For OC4J, you must change your code to use the constructor that takes a parameter. The following code example illustrates this:

```
//WebSphere Code
try
{
  java.util.Properties parms = new java.util.Properties();
  parms.setProperty(Context.INITIAL_CONTEXT_FACTORY,
                          "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
  javax.naming.Context ctx = new javax.naming.InitialContext(parms);
  javax.sql.DataSource ds = (javax.sql.DataSource)ctx.lookup("jdbc/SampleDB");
  java.sql.Connection conn = ds.getConnection();

  // process the results
  ...
}
```

To migrate from WebSphere, you must change the class that implements the initial context factory (Context.INITIAL_CONTEXT_FACTORY) of the JNDI tree:

from the WebSphere-specific class:

```
com.ibm.ejs.ns.jndi.CNInitialContextFactory
```

to the OC4J-specific class:

```
com.evermind.server.ApplicationClientInitialContextFactory
```

Connection Pooling

Most web-based resources, such as servlets and application servers, access information in a database. Each time a resource attempts to access a database, it must establish a connection to the database using system resources to create the connection, maintain it, and release it when it is no longer in use. The resource overhead is particularly high for web-based applications due to the frequency and volume of web users connecting and disconnecting. Often, more resources are consumed in connecting and disconnecting than in executing the business logic.

Connection pooling enables you to control connection resource usage by spreading the connection overhead across many user requests. A connection pool is is a cached set of connection objects that multiple clients can share when they need to access a database resource. The resources to create the connections in the pool are expended only once for a specified number of connections. The connections are left open and re-used by many client requests instead of each client request consuming resources to create and close its own connection. Connection pooling improves overall performance in the following ways:

- Reducing the load on the middle-tier server

- Minimizing resource usage by having session-wide create and close operations

- Eliminating bottlenecks caused by socket and file descriptor limitations and 'n' user license limitations

The JDBC 2.0 specification allows you to define a pool of JDBC database connections, with the following objectives:

- Maximize the availability of connections to resources

- Minimize the idle connections in the pool

- Return orphan connections to the pool and make them available for reuse by other servlets or application servers.

To meet these objectives, you should perform the following:

1. Set the maximum connection pool size property equal to the maximum number of concurrently active user requests expected

2. Set the minimum connection pool size property equal to the minimum number of concurrently active user requests expected.

The connection pooling properties ensure that as the number of user requests decreases, unused connections are gradually removed from the pool. Likewise, as the number of user requests begins to grow, new connections are created. The balance of connections is maintained so that connection re-use is maximized and connection creation overhead minimized. You can also use connection pooling to control the number of concurrent database connections.

Migrating WebSphere Connection Pooling to Oracle9*i* Application Server

WebSphere Advanced Edition 3.5.3 provides two options for accessing database connections:

- Connection pooling (model based on JDBC 2.0)

- Connection manager (model based on JDBC 1.0)

Migrating from WebSphere JDBC 2.0 connection pooling:

WebSphere implements JDBC 2.0 connection pooling and data source objects using the following packages.

```
import com.ibm.db2.jdbc.app.stdext.javax.sql.*;
import com.ibm.ejs.dbm.jdbcext.*;
```

To migrate from the WebSphere JDBC 2.0 connection to OC4J you must replace these import packages with `javax.sql.*`.

An application component that obtains two or more connections to the same database manager (using either the same data source or different data source) must use data sources with JTA-enabled drivers. For more information, refer to *Oracle9iAS Containers for J2EE User's Guide.*

IBM Extensions

WebSphere provides the following extension packages for data access. Applications using these packages require code level changes for migration.

Data Access Beans

WebSphere Advanced Edition 3.5.3 also provides data access beans (in addition to access beans for EJBs), which offer a set of features for working with relational database queries and result sets. The `com.ibm.db` package contains the data access JavaBean classes. The classes are in the `databeans.jar` file (found in the `lib`

directory under the application server root install directory). You will need this JAR file in your classpath in order to compile a servlet using the data access JavaBeans.

If you have lot of code using data access beans that need to be migrated to OC4J, then put `databeans.jar` in the classpath of OC4J. However, Oracle recommends that you migrate to JDBC 2.0 APIs.

Connection Pool Manager

As mentioned, IBM WebSphere 3.5.x still supports connection pooling with a proprietary connection pool manager. Oracle recommends that you develop connection pooling using IBM's standard extensions for JDBC 2.0.

A

J2EE Server Comparison

This chapter compares WebSphere Advanced Edition 3.5.3 to Oracle9*i*AS OC4J in the following areas:

- Installation and Configuration
- Code Portability
- Database Connections
- Security
- Performance and Scalability

Installation and Configuration

- Setting up WebSphere is complex and is different on different platforms. The most stable version of WebSphere is v3.5, but it requires service pack 3 or 4 in order to be J2EE 1.2 compliant. If there are problems during the installation of the service pack you cannot un-install or re-apply the service pack; you must re-install WebSphere.

  Oracle9*i*AS OC4J takes only a few minutes to install and start.

- WebSphere requires a database repository for installation, and to run applications. WebSphere stores server and application configuration data in the database, even for a single instance of WebSphere. Because of this, starting an application takes an unusually long time, even in a development environment. DB2 must be installed before installing WebSphere 3.5.x.

  OC4J uses standard, easy-to-configure XML files for server configuration and J2EE applications deployment. An administration repository (Oracle9*i*AS infrastructure) is not required unless a cluster with more than one Oracle9*i*AS instance is required.

Code Portability

- WebSphere requires developers to use proprietary packages for standard J2EE features. To use servlet chaining or filtering, developers must use WebSphere-specific packages. These are not WebSphere-specific features but are part of the J2EE servlet 2.3 API specifications. Hence, WebSphere 4.0 does not support the Servlet 2.3 API requirements.

  OC4J is fully complaint with the latest Servlet 2.3 API

- WebSphere v3.5.x CMP entity beans are not portable. The WebSphere EJB container requires every CMP entity bean to define an interface named *<ejbname>FinderHelper*. This is not defined in the J2EE standard and is not portable.

- WebSphere-specific access beans are not portable. WebSphere strongly influences developers to use access beans. These are client side beans to access EJBs and are not portable. The VisualAge tool generates access beans by default.

  OC4J is fully complaint with EJB 1.1 and also supports advanced features of EJB 2.0 (WebSphere 4.0 does not support the EJB 2.0 spcification).

Table A–1 shows the level of J2EE support provided by versions of WebSphere and OC4J.

Table A–1   J2EE component specification version support

Technology	WebSphere 3.5.3	WebSphere 4.0	Oracle9*i*AS OC4J
JDK	1.2.2	1.2.2 and 1.3	1.2.2 and 1.3
Servlets	2.1+	2.2	2.2 and 2.3
JSPs	1.0	1.1	1.1
EJBs	1.0+	1.1	1.1 and 2.0
JDBC	2.0	2.0	2.0
JNDI	1.2	1.2	1.2.1
JTA	1.0	1.0	1.01.
JMS	1.0	1.0	1.0.1

Database Connections

- WebSphere consumes more memory for each database connection it opens.Prepared SQL statements are cached across the pool of connections, by J2EE server, for faster execution. WebSphere connection pooling fails if you try to create more than one prepared SQL statement simultaneously per connection.

  OC4J supports more than one prepared SQL statement simultaneously. This significantly improves database access performance in OC4J as compared to WebSphere.

Security

- Enabling security on WebSphere degrades performance significantly for EJBs. Application response times degrade by a factor of two. For each EJB interaction triggered by a HTTP transaction, WebSphere generates a large log output when security is enabled. Also, WebSphere is not 100% complaint with J2EE security APIs. For example, calls such as is **CallerInRole** does not work (a **NotImplemented** exception is thrown).

  OC4J is fully compliant with J2EE 1.2 security APIs.

Performance and Scalability

- WebSphere uses RMI-IIOP to communicate between components. This is particularly troublesome in the Enterprise Edition in which the Component Broker performs the dual role of EJB container and Object Request Broker (ORB) server. WebSphere is known to have performance problems with its RMI-IIOP implementation due to its use of the CORBA cosNaming service for object lookup.

  OC4J uses a very efficient and optimized RMI (ORMI).

- Co-located servlets, JSPs, and EJBs have very short code paths in OC4J compared to WebSphere.

- Container-managed persistence with associations causes performance problems in WebSphere. WebSphere support for CMP association does not cache the InitialContext nor the EJBhome objects. As a framework, it uses lookups extensively through the homes, with each home lookup equal in performance cost to executing 4 DB "SELECTs". It is recommended that you build a caching "HomeFactory" for application use as well as make changes to the association

framework so that it also uses the caching "HomeFactory". These functionalities should be addressed by the EJB server itself (as they are by OC4J).

- EJB-to-EJB communication is very fast and efficient in OC4J as compared to WebSphere. OC4J uses local non-RMI calls within a given JVM.

- WebSphere requires tuning of too many parameters to realize any performance improvements. The following parameters must be tuned for performance:

    - JVM

    - Transport queues between HTTP listener plug-in- Open Sevlet Engine(OSE) and Servlet Container.

    - Servlet auto reload (this feature is part of hot deployment and causes performnce problems since the Administration Server has to update configuration information in the database).

    - Transport queues between sevlet container and EJB container.

    - ORB (for enterprise edition).

    OC4J requires very minimal tuning as compared to WebSphere. The latest release of WebSphere, version 4.0, replaces support for multiple transport protocols with a single HTTP-IIOP protocol to commnicate between the HTTP server and application server.

- WebSphere consumes more resources (CPU and memory) than OC4J. This not only true for starting the application server, it is also true for each request.

- WebSphere clustering performance degrades for more than 3 to 4 nodes in a cluster.

- The EJB container in WebSphere Enterprise Edition is based on that in the WebSphere Advance Edition.

B

Migrating from WebSphere 4.0

This appendix outlines the migration strategy from WebSphere Advanced Edition 4.0 to Oracle9*i*AS in the following aspects:

- Feature Differences Between WebSphere Advanced Edition 3.5.3 and 4.0

- J2EE Specification Differences Between WebSphere Advanced Edition 4.0 and Oracle9iAS

- Migrating WebSphere 4.0 Servlets to Oracle9iAS

- Migrating WebSphere 4.0 JSPs to Oracle9iAS

- Migrating WebSphere 4.0 EJBs to Oracle9iAS

- Other Considerations

Feature Differences Between WebSphere Advanced Edition 3.5.3 and 4.0

WebSphere Advanced Edition 4.0 is an evolution of the 3.5.x version in several areas including J2EE specification support up to J2EE 1.2 for 4.0. Table B–1 summarizes the evolved features. The information in the table provides a general reference for what you can reuse from the other chapters in this book for your migration tasks from WebSphere Advanced Edition 4.0 tp Oracle9*i*AS Release 2.

Table B–1 Summary of WebSphere 3.5.x and 4.0 feature differences

WebSphere Advanced Edition 3.5.x	WebSphere Advanced Edition 4.0
Policy-based authorization	J2EE roles-based authorization - method invocation permissions for enterprise beans can now be assigned to users through their respective roles. Roles are specified in an application's deployment descriptors.
JSP 1.0	JSP 1.1
Servlet 2.1	Servlet 2.2
EJB 1.0	EJB 1.1
Presentation layer separation - HTML and CGI requests are served by a separate product. No implementation of J2EE Web Application concept.	J2EE Web Application support (support for `.war` files)
`ServiceInitializer` interface	`CustomService` interface - Similar to the `ServiceInitializer` interface except that `CustomService` does not pass in the context for the services to use for registration binding.
Connection Manager V2.0	Connection pooling - JDBC 2.0 or data access beans is used for connection pooling.
JNDI context interface used: `com.ibm.ejs.ns.jndi.` `CNInitialContextFactory`	New JNDI context interface used: `com.ibm.websphere.naming.` `WsnInitialContextFactory`
XML4J V2.0.15 parser	XML4J V3.1.1 parser
`.servlet` file support	Servlet extensions are bundled in `.war` file.
Open Servlet Engine (OSE) remote and servlet redirector support for remote request invocations.	HTTP transport plug-ins to communicate between web server and application server.

J2EE Specification Differences Between WebSphere Advanced Edition 4.0 and Oracle9*i*AS

WebSphere Advanced Edition 4.0 advances on the J2EE specification support from WebSphere Advanced Edition 3.x. It has more comparable features with Oracle9*i*AS in the J2EE specification levels. Table B–2 compares the specification levels supported by Oracle9*i*AS Release 2 and WebSphere Advanced Edition 4.0.

Table B–2   Summary of Oracle9iAS and WebSphere Advanced Edition 4.0 feature differences

J2EE Specification	WebSphere Advanced Edition 4.0	Oracle9iAS
JDK	1.2.2 and 1.3	1.2.2 and 1.3
Servlet	2.2	2.2 and 2.3
JSP	1.1	1.1 and 1.2
EJB	1.1	1.1 and 2.0
JDBC	2.0	2.0
JNDI	1.2	1.2.1
JTA	1.0.1	1.0.1
JMS	1.0	1.0.1

Migrating WebSphere 4.0 Servlets to Oracle9*i*AS

WebSphere 4.0 is compliant with Servlet 2.2 specifications. It has proprietary mechanisms to enable servlet chaining and filtering and is not fully Servlet 2.3-compliant. The WebSphere 3.5.3 compatibility mode is not available. With those points in mind and not taking Servlet 2.3 features into consideration, migrating WebSphere Advanced Edition 4.0 servlets to Oracle9*i*AS OC4J servlets should be straightforward as both products support the same level of specifications (Servlets 2.2). However, the following possible incompatibilities should be noted and resolved if applicable to your servlets:

- WebSphere Specific Servlet Extensions

- WebSphere-Specific Deployment Descriptors

- Deprecated 3.5.3 API (Supported in WebSphere 4.0)

WebSphere Specific Servlet Extensions

The servlet extensions from WebSphere Advanced Edition 3.5.3 are still available in WebSphere Advanced Edition 4.0. Information in the section "WebSphere Extensions to the Servlet API" on page 3-13 is still valid when these extensions are used. If your servlets use any of these extensions, they need to be re-written to conform with the standard Servlet 2.2 or 2.3 API in order to run in OC4J.

WebSphere-Specific Deployment Descriptors

WebSphere Advanced Edition uses non-J2EE compliant deployment descriptors. WebSphere 4.0's Application Assembly Tool generates additional descriptor files in addition to the standard J2EE files. These are the DDL and XMI files used to store binding and WebSphere-specific extension information.

Since these files apply to WebSphere-specific extensions, they are redundant in Oracle9iAS and need not be migrated. Ensure that other WebSphere-specific extensions are not implemented in the migrated application before deploying in Oracle9iAS.

Deprecated 3.5.3 API (Supported in WebSphere 4.0)

If your servlets deployed in WebSphere 4.0 use the 3.5.3 deprecated API shown in the following table, you need to re-write them to use the equivalent Servlet 2.2 API as follows:

Table B–3   Deprecated 3.5.3 API and their Servlet 2.2 replacement

WebSphere 3.5.3 (supported in 4.0)	Servlet 2.2
getValue()	getAttribute()
getValueNames()	getAttributeNames()
removeValue()	removeAttribute()
putValue()	setAttribute()

Migrating WebSphere 4.0 JSPs to Oracle9iAS

WebSphere Advanced Edition 4.0 and Oracle9iAS both support JSP 1.1 (with Oracle9iAS supporting 1.2 as well). Hence, JSP migration between the two products should be straightforward. In general, the rules and processes that apply when migrating JSPs from WebSphere Advanced Edition 3.x to Oracle9iAS can also be applied here. This is especially true for variations deviating from the JSP 1.1 specifications. These are related to the "tsx" family of tags. In OC4J, these should be replaced with OracleJSP's JML tags. Refer to the section "Migrating WebSphere Extensions to OC4J" on page 4-9 for more information.

Migrating WebSphere 4.0 EJBs to Oracle9iAS

WebSphere Advanced Edition 4.0 complies with EJB 1.1. This is the same specification level which Oracle9iAS is compliant with (some EJB 2.0 features are

also supported by Oracle9*i*AS). Hence, migrating EJBs between the two application servers should not pose any significant challenges. We strongly recommend that you archive your EJBs in a EAR file with any web applications and deploy that file using OEM or `dcmctl` to ensure that appropriate stubs are generated by Oracle9*i*AS. Copying EJB classes and their WebSphere-compiled stubs manually is not recommended.

We are investigating the EJB migration process further and will update this document and information in the Oracle Technology Network website (http://otn.oracle.com) if we encounter any issues.

Other Considerations

In addition to the migration points above, the following should also be observed:

Dynamic Fragment Cache

Dynamic Fragment Cache is a performance enhancement feature in WebShpere that caches the output of servlets and JSPs. This feature intercepts calls to the `service` method of servlets and determines if the calls can be serviced by its cache. For servlets or JSPs to use this feature, they have to use the `com.ibm.websphere.servlet.cache` package. When migrating to Oracle9*i*AS, ensure that this package is removed and any related code modified. For caching functionality in Oracle9*i*AS, consider using Oracle9*i*AS Web Cache and its Edge Side Includes for Java (JESI) technology.

Data Access and Sources

WebSphere 4.0 provided the `com.ibm.db` package as a substitute for the standard JDBC package `java.sql`. When migrating to Oracle9*i*AS, replace usage of `com.ibm.db` with `java.sql` (standard JDBC 2.0 package is recommended). If your application and components also use IBM data access beans, follow the guidelines in "Data Access Beans" on page 6-10.

In WebSphere 4.0, all data sources must be created using `com.ibm.websphere.advanced.cm.factory.DataSourceFactory`. To migrate to Oracle9*i*AS, data sources can be obtained using the JDBC 2.0 `java.sql.Datasource`, which is a connection factory to a data source. A `Datasource` object can be obtained by looking it up in the JNDI namespace. Refer to "The DataSource Class" on page 6-4 for more information.

C

Partner Migration Tools

Enterprises migrating their applications to Oracle9*i*AS from other application servers can now do so easily and more effectively using our partner migration tools.

Cacheon

Cacheon Migrator helps automate and improve the productivity of migrating existing J2EE applications from BEA WebLogic and IBM WebSphere to Oracle9*i*AS.

With Cacheon Migrator, developers can automatically convert JSPs, EJB deployment descriptors, Java source code, JMS settings, WEBINF configuration files, and third party security features into Oracle9*i*AS.

Cacheon Migrator automates up to fifty percent of the migration effort. For the remaining portion, the software highlights areas that may require manual intervention and offers suggested solutions.

Features of Cacheon Migrator

- Fast J2EE migration - Accelerate the conversion of EJB deployment descriptors, Java source code, JSP and tag libraries, JMS settings and configuration files, and security features.

- Migration workbench - Manage the conversion of J2EE applications from one application server platform to another.

- Issue resolver - Identify and remove application server dependencies and build an overall schedule for a migration project.

- Migration reports - Identify all source application modifications and track problem areas during migration.

- Customize rules - Customize conversion rules to meet application specific requirements.

- Create new rules - Create new rules in a scripting language to extend migration capabilities.

- JDeveloper integration - Streamline the migration process by converting applications directly within Oracle9*i* JDeveloper.

For more information and resources, visit http://www.cacheon.com.

TogetherSoft

Together ControlCenter, the comprehensive Model-Build-Deploy Platform for end-to-end software development, enables organizations to not only deploy to leading application servers, but also lets them migrate their J2EE applications from other J2EE-compliant application servers to Oracle9*i*AS.

ControlCenter handles the underlying deployment details so that developers can focus on the business logic of the application instead of the application server specifics.

An online viewlet that demonstrates how Together ControlCenter can help migrate an application from WebSphere to Oracle9*i*AS can be found at: http://www.togethersoft.com/developers/integrations/oracle9ias.jsp and http://otn.oracle.com/products/ias/daily/mar28.html.

Index

E

F

G