Oracle9*i* Application Server

Migrating From WebLogic

Release 2 (9.0.2)

April 2002

Part No. A95109-01

ORACLE

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

Part No. A95109-01

Primary Authors: Anand Ramakrishnan, Kai Li

Contributing Authors: Joan Gregoire

Contents

3 Migrating Java Servlets

4 Migrating JSP Pages

5 Migrating Enterprise JavaBean Components

6 Migrating JDBC

Send Us Your Comments

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

Part No. A95109-01

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 WebLogic provides you with the information required for a successful migration from BEA Systems' application server, WebLogic Server 6.0, to Oracle Corporation's application server, Oracle9*i* Application Server (Oracle9*i*AS).

This preface contains these topics:

Audience

Oracle9i Application Server Migrating From WebLogic is intended for those Java development managers, application developers, and system administrators responsible for planning and migrating J2EE web applications from BEA System's WebLogic Server 6.0 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 WebLogic Server environment, as well as the Oracle9*i*AS environment.

Organization

This document contains:

Chapter 1, "Overview"

This chapter provides an overview of the issues involved in migrating J2EE web applications from WebLogic Server 6.0 to Oracle9*i* Application Server (Oracle9*i*AS), and the effort required.

Chapter 2, "Comparison of Oracle9iAS and WebLogic Server 6.0"

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

Chapter 3, "Migrating Java Servlets"

This chapter provides the information you need to migrate Java servlets from WebLogic Server 6.0 to Oracle9*i*AS. It addresses the migration of simple servlets, WAR files, and exploded web applications.

Chapter 4, "Migrating JSP Pages"

This chapter provides the information you need to migrate JavaServer pages from WebLogic Server 6.0 to Oracle9*i*AS. It addresses the migration of simple JSP pages, custom JSP tag libraries, and WebLogic custom tags.

Chapter 5, "Migrating Enterprise JavaBean Components"

This chapter provides the information you need to migrate Enterprise JavaBeans from WebLogic Server 6.0 to Oracle9*i*AS. It addresses the migration of stateful and

stateless session beans and container-managed persistence and bean-managed persistence entity beans.

Chapter 6, "Migrating JDBC"

This chapter provides the information you need to migrate database access code from WebLogic Server 6.0 to Oracle9*i*AS. It addresses the migration of JDBC drivers, data sources, and connection pooling.

Appendix A, "Oracle9iAS 1.0.2.x and WebLogic Server 6.0 Comparison"

This appendix provides a comparison of performance and features between Oracle9*i*AS version 1.0.2.2.x and BEA Systems' WebLogic Server 6.0. (Oracle9*i*AS 1.0.2.2 is the predecessor Oracle9*i*AS Release 2.)

Appendix B, "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://bea.com/` for more information on WebLogic 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**.
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.

Convention	Meaning	Example
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.

1

Overview

This chapter provides an overview of the issues involved in migrating J2EE web applications from WebLogic Server 6.0 to Oracle9*i* Application Server (Oracle9*i*AS) Release 2, and the effort required.

The chapter contains these topics:sv

Note: Unless explicitly stated, any mention of "Oracle9*i*AS" in this guide refers to Oracle9*i*AS Release 2.

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 environment	
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 and JSP pages	Servlet: A Java program, used to generate dynamic content, that executes on a web server
	JSP page: 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 (Cont.)

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. It helps separate system complexity from business logic, enabling developers to focus on solving business problems. An application server helps 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. Oracle9*i*AS enables developers to develop web applications in any language including Java, Perl, PL/SQL, XML, and 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 Oracle9*i*AS Containers for J2EE (OC4J). OC4J runs on the standard JDK and is extremely lightweight, provides high performance and scalability, and is simple to deploy and manage. With Oracle9*i*AS Release 2, the OC4J supports J2EE 1.2 with support for some J2EE 1.3 features.

This migration guide seeks to help you understand the migration challenges you may face when migrating your J2EE applications from Weblogic Server 6.0 to Oracle9*i*AS.

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
- 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 WebLogic Server 6.0 to Oracle9*i*AS, providing an approach to migration with solutions based on the J2EE version 1.2 specification.

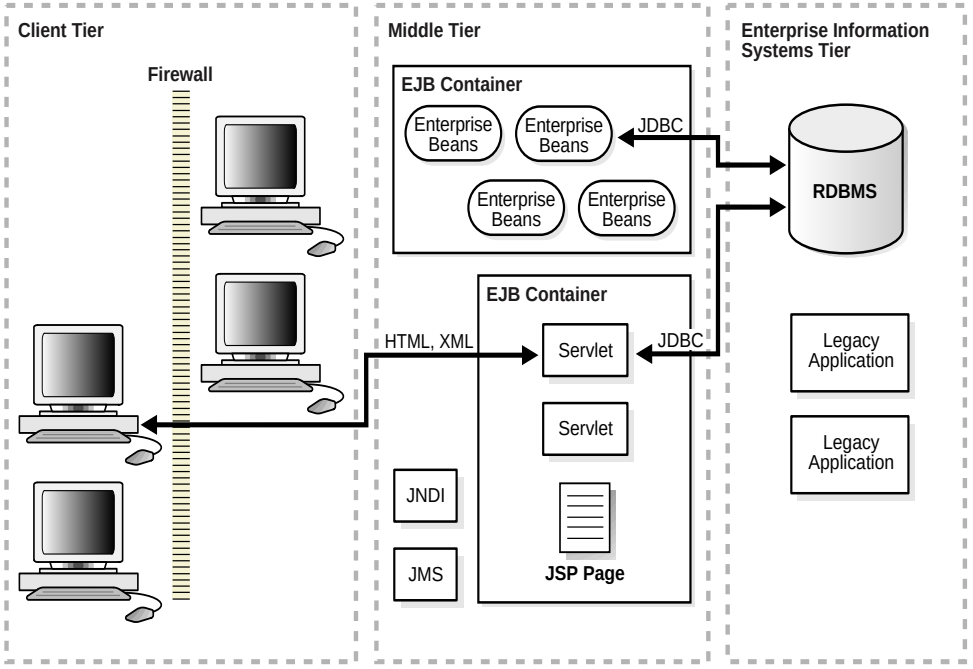
J2EE Application Architecture

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 JavaBeans (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

In quantifying the migration effort, it is helpful to examine the application components to be migrated with the following issues in mind:

- Portability

  Code may not be portable because it contains embedded references to vendor-specific extensions to the J2EE specification. Evaluating and planning for code modifications may be a significant part of the migration effort.

- Proprietary extensions

  If vendor-specific extensions are in use, migration of those components becomes difficult or unfeasible. Complete redesign toward J2EE specifications is not addressed in this document. If vendor-specific extensions are in use, they may need to be redesigned and reimplemented, rather than being identified as migration candidates.

- Deviations from J2EE Specification, v1.2

  If a component is largely non-compliant with the J2EE specification, this guide will not be helpful in determining the migration path to Oracle9*i*AS. If the J2EE specification version of the component is not v1.2 (the version on which this guide is based), then the specification implementation differences will need to be addressed.

Migration Approach

The approach in developing this migration guide was to document our experience migrating web application components from WebLogic Server 6.0 to Oracle9*i*AS. Examples shipped with WebLogic Server 6.0 were selected, tested on WebLogic Server 6.0, and migrated to Oracle9*i*AS. Issues encountered in the migration of these examples are the basis for this document.

Migration Effort

Moving from WebLogic Server 6.0 to Oracle9*i*AS is a relatively simple process. Standard J2EE applications, using no proprietary APIs, can be deployed with no required code changes. The only actions required are configuration and deployment. Those applications using proprietary utilities or APIs can be ported easily.

Using This Guide

This guide details the migration of components from WebLogic Server 6.0 to Oracle9*i*AS. 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 WebLogic Server 6.0 applications and plan and execute their migration to Oracle9*i*AS. 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

2

Comparison of Oracle9*i*AS and WebLogic Server 6.0

Although WebLogic Server 6.0 and Oracle9*i*AS are both J2EE servers that support J2EE 1.2 and some J2EE 1.3 features, both application servers have intrinsic differences ranging from product packaging to runtime architecture. This chapter seeks to discuss these differences and is organized as follows:

Application Server Product Offerings

WebLogic Server 6.0 is sold in several configurations. This section outlines these configurations and Oracle9*i*AS Release 2.

WebLogic Server 6.0

WebLogic Server 6.0 is available in three configurations: WebLogic Server, WebLogic Enterprise, and WebLogic Express.

WebLogic Server

WebLogic Server provides the core services and infrastructure for J2EE applications. It supports J2EE 1.2 with additional support for some J2EE 1.3 features. These J2EE 1.3 features include JSP 1.2, Servlets 2.3, EJB 2.0, and JCA 1.0. However, these are

implemented from non-final J2EE 1.3 specifications. Hence, compatibility problems will surface if a J2EE 1.3 compliant application is deployed on WebLogic Server 6.0.

WebLogic Server allows Java applications and components to be deployed as web services through SOAP, UDDI, and WSDL. It does not, however, support CORBA applications. CORBA support is available in WebLogic Enterprise (see next section).

Each WebLogic Server can be configured as a web server utilizing its own HTTP listener, which supports HTTP 1.1. Alternatively, Apache, Microsoft IIS, and Netscape web servers can also be used. This web server configuration allows WebLogic Server to service requests for static HTML content in addition to dynamic content generated by servlets or JSPs.

A WebLogic Server node can be deployed as an administration server. This node provides administrative services to other WebLogic Servers, called managed servers, in the WebLogic domain. A WebLogic domain is a set of WebLogic Servers and clusters of WebLogic Servers managed by an administration server, inclusive of the latter. The administration server provides a web-based GUI for management of the entire domain. In each domain, WebLogic Servers can be clusted together or are standalone. Refer to "Oracle9iAS Support for Clustering and Load Balancing" for more clustering information.

Note: For a list of J2EE 1.2 APIs supported by WebLogic Server, refer to "J2EE Support Comparison" in this chapter.

WebLogic Enterprise

WebLogic Enterprise consists of WebLogic Server and BEA Tuxedo. Tuxedo is a distributed transaction management platform that enables distributed transactions across multiple databases. Tuxedo integrates with WebLogic Server through the latter's connector architecture.

WebLogic Enterprise supports multiple application environments including Java, C++, C, and COBOL. WebLogic Enterprise also supports CORBA applications and allows single sign-on to disparate application environments. Additionally, through Tuxedo, WebLogic Enterprise supports industry standard SNMP MIBS allowing WebLogic Server to be monitored by third-party tools.

WebLogic Express

WebLogic Express is a "lightweight"version of WebLogic Server. It is not J2EE compliant as it does not have support for EJBs and JMS. It does support JSPs, servlets, JDBC, and RMI, and it also includes a web server. Hence, WebLogic

Express can be used to build rudimentary web applications with simple database access using JDBC (no support for two-phase transactions).

Oracle9*i* Application Server

Like WebLogic Server, 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.

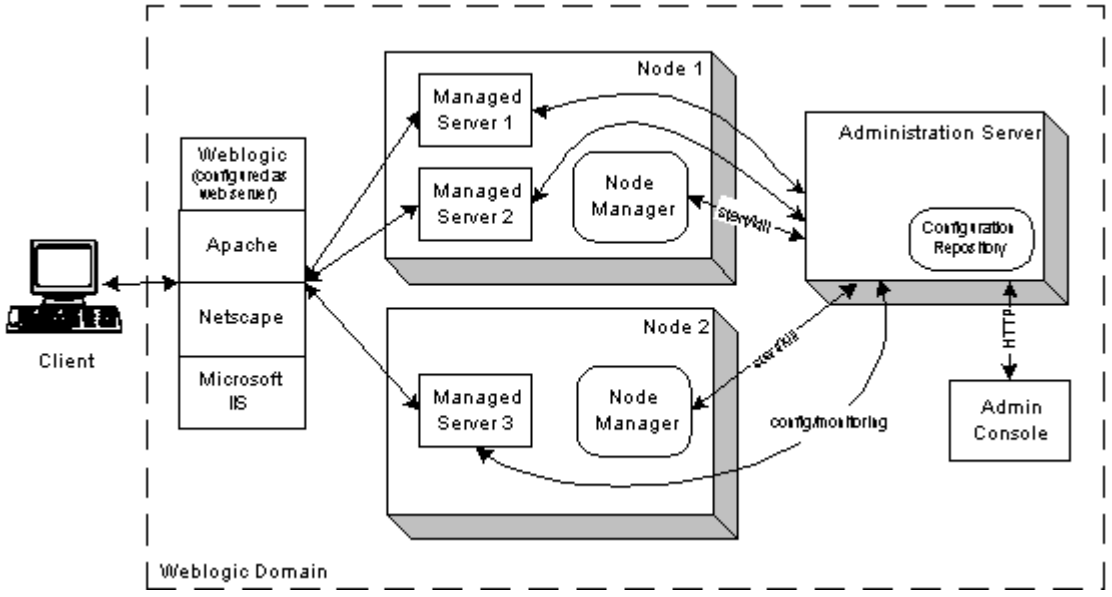
Architecture Comparison

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

WebLogic Server 6.0 Components and Concepts

WebLogic Server has several components and concepts peculiar to it. Each WebLogic Server can be configured and deployed either as a Managed Server or an Administration Server. A Managed Server hosts and executes the application logic deployed in it when requests are received from clients. An Administration Server configures and monitors Managed Servers. Figure 2–1 depicts the components in WebLogic Server and their interactions.

Figure 2–1 WebLogic Server components



In any node, more than one Managed Server can exist. Each Managed Server is a Java process (JVM) executing J2EE containers (Web and EJB). An Administration Server, which is also a Java process, is required to propagate configuration information to Managed Servers when they start-up. The configuration information is stored in the filesystem on the Administration Server node.

The Administration Server is also used to monitor and log information about individual Managed Servers and the entire WebLogic domain. A WebLogic domain can consist of standalone Managed Servers, clusters of Managed Servers, and one Administration Server. If the Administration Server goes offline, client requests can still be serviced by the Managed Servers. However, configuration information is not available for new Managed Servers to start-up, and monitoring services are not available for server clusters. The Administration Server does not have automatic failover or replication. Configuration data for the WebLogic domain has to be manually backed. The Administration Server functions can be accessed through a console GUI (remotely over HTTP) or a command line utility.

In order for the Administration Server to start Managed Servers remotely, a Node Manager must be running on each node where there are Managed Servers. This Node Manager is a Java program executing in the background as a Unix daemon or Windows NT service. With the Node Manager, the Administration Server can also

kill a Managed Server if the latter hangs or does not respond to commands from the former.

WebLogic Server 6.0 can also be set up to run as a web server. In this mode, it supports HTTP 1.1 and resolves client requests to Managed Servers based on the settings in the XML configuration files. Instead of WebLogic Server 6.0, third-party proxy plug-ins can also be used. Supported plug-ins are Apache, Netscape, and Microsoft IIS.

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.

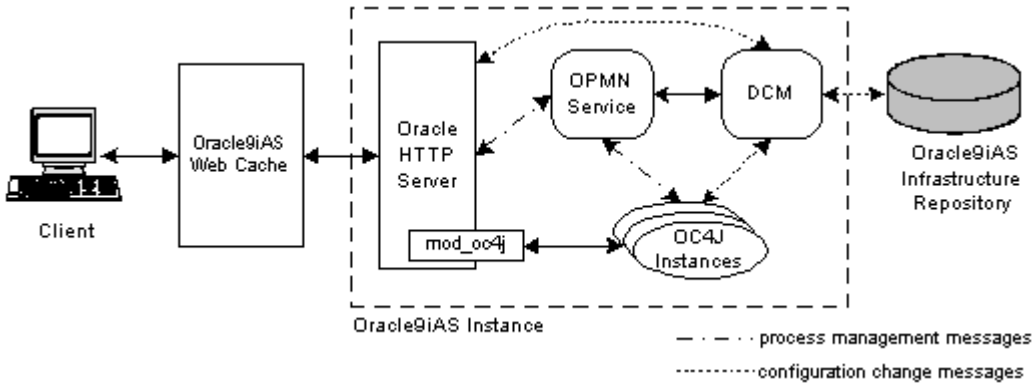
The OC4J listener listens to requests coming from the `mod_oc4j` module of the Oracle HTTP Server and forwards them to the appropriate OC4J instance. From a functional viewpoint, the Oracle HTTP Server acts as a proxy server to OC4J, wherein all servlet or JSP requests are redirected to OC4J instances.

`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 Oracle9iAS 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 WebLogic Server 6.0 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 LoadBalancing?

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.

WebLogic Server 6.0 Suppport for Clustering and Load Balancing

One or more WebLogic Servers can be grouped together as a cluster. Applications can be deployed commonly in all servers in a cluster, through cluster-wide deployment, to allow client requests to be load balanced across the cluster and the applications to have failover capabilities. In a WebLogic cluster, the entities that benefit from clustering are HTTP session states, and EJB and RMI objects. Several load balancing algorithms are used by WebLogic. These are round-robin, weight-based, and parameter-based.

HTTP Session State Load Balancing and Failover (Servlet Clustering)

Clients making requests to a WebLogic cluster can have their requests load balanced across the servers in the cluster. For this to work, a web server installed with the WebLogic proxy plug-in or a hardware load balancer must be used. The WebLogic proxy plug-in uses a round-robin load balancing mechanism to distribute the request load. If a hardware load balancer is used, the cluster can be load balanced using the hardware's mechanism.

WebLogic Server achieves failover for servlets and JSPs by replicating the HTTP session states of clients. When a WebLogic Server receives the very first request for a servlet or JSP, it replicates the servlet's session state to another server. The replicated session state is always kept up-to-date with the original. The WebLogic proxy plug-in returns the names of the two servers to the client through a cookie or by rewriting the URL. If the server hosting the original session state fails, the WebLogic proxy plug-in uses the information in the cookie or URL to redirect the client to the server with the replicated session state. At any one time, the cluster maintains an original and replica of each active session state. In this scenario, the session state is replicated in memory. WebLogic Server also supports replication to the file system or a database through JDBC, however, the failover is not automatic for these replication methods.

EJB and RMI Object Load Balancing and Failover

WebLogic Server provides load balancing and failover for EJB and RMI objects by using a JNDI service and client stubs which are both cluster-aware.

Each WebLogic Server in a cluster maintains a local JNDI tree. This tree contains information on objects deployed on the local server and around the cluster (for objects that are clusterable). If a clusterable object is deployed on more than one

server, each JNDI tree reflects the existence of that object on those servers. When a clusterable object is deployed on a server, that server, through multicast, notifies the other servers in the cluster of the new deployment. The other servers' update their JNDI trees accordingly. Note that the server with the deployed object also sends the object's stub to the other servers.

When a client looks up a clusterable object in the JNDI service, the server servicing the request returns a stub of the object to the client. This stub contains information about which server(s) the object is actually deployed in. The stub also has load balancing logic to balance method calls to the object. The load balancing algorithms available are round-robin, weight-based, random, and parameter-based. From the client's point-of-view, the cluster is transparent. The JNDI look ups and load balancing are done without the client knowing that it is working with a clustered object at the server end.

In the case where a clustered object is stateful, for example, a stateful session EJB, the object's state is replicated to a second server. The replication is achieved in a similar manner as for HTTP session state. The server that is chosen to service a client's very first request replicates the object's state to another server. The client stub is updated to reflect this. If the first server fails, the stub receives an exception when it tries to invoke a method. The stub then redirects the invocation to the server with the replicated object state. This server instantiates the object with the replicated state and executes the method invocation. The server also selects another server to replicate the state to since the original server is down. Failover of stateful objects is achieved this way.

Failover of stateless objects is more straightforward to achieve as state need not be replicated. Upon receiving an exception indicating that a server has failed, the client stub simply selects another server which is hosting another instance of the called object and redirects the method invocation there.

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

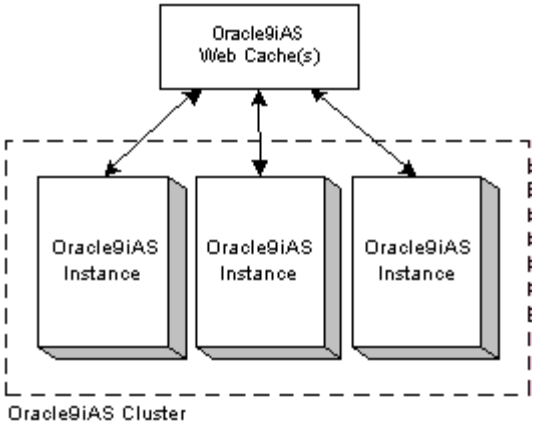
An Oracle9*i*AS cluster is made up of one or more Oracle9*i*AS instances (see Figure 2–3). All Oracle9*i*AS instances in the cluster have the same configuration. The first Oracle9*i*AS instance to join a cluster has its configuration replicated to the

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



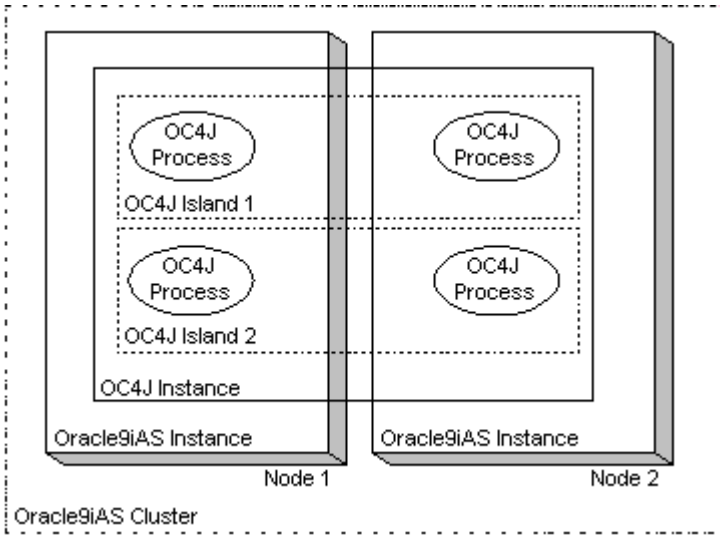
OC4J Islands

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 WebLogic Server 6.0 and Oracle9*i*AS.

Oracle9*i*AS 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.

WebLogic Server 6.0 supports J2EE 1.2. It also has some early J2EE 1.3 features which were implemented before the J2EE 1.3 specifications were finalized.

Table 2–1 lists the J2EE technologies and the level of support provided by Oracle9*i*AS and WebLogic Server 6.0:

Table 2–1 J2EE Technology Support

J2EE Technology	Version Supported by WebLogic Server 6.0	Version Supported by Oracle9*i*AS
JDK	1.3	1.2.2 and 1.3
Servlets	2.2 (early 2.3.2)	2.2 and 2.3
JSPs	1.2.1	1.2
EJBs	1.1 (early 2.0)	1.1 and 2.0
JDBC	2.0	2.0
JNDI	1.2	1.2.1
JTA	1.0.1	1.0.1
JMS	1.0.2	1.0.1
JavaMail	1.1	1.2
JAF	None	1.0.1
JAXP	1.0.1	1.1
JCA	1.0	1.0
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 WebLogic Server and Oracle9*i*AS.

WebLogic Server Development and Deployment Tools

The WebLogic Server development environment, tools, and Administration Console are described below.

WebLogic Server Development Tools

WebLogic has partnered with WebGain to provide a suite of tools. WebLogic itself does not provide the tools as an integrated package with its WebLogic Server. The tools available from WebGain are:

- Application Composer - a rapid application development environment
- Business Designer - a tool for managing business process requirements
- TopLink - an object-relational mapping tool
- WebGain Studio - a Java IDE

WebLogic Server Administration Console

The WebLogic Server administrative console provides a GUI for managing the WebLogic Server domain. A WebLogic Server domain consists of one or more WebLogic Server instances (where each instance runs one or more applications) or clusters of instances. The administrative console connects to the designated administrative server running in the domain and can be used to change the configuration or run-time state on any machine in a domain. The administrative console is used to define clusters, add servers, deploy applications, configure applications, and manage web servers, services, and resources in the domain.

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 Interent 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.xml` file, and deployment descriptors.

You can build applications with Oracle JDeveloper and deploy them manually, using Oracle Enterprise Manager, or with the OC4J Administration Console. 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 Java Servlets

This chapter provides the information you need to migrate Java servlets from WebLogic Server 6.0 to Oracle9*i*AS. It covers the migration of simple servlets, WAR files, and exploded web applications.

This chapter contains these topics:

Introduction

Migrating Java servlets from WebLogic Server 6.0 to Oracle9*i*AS is typically straight forward, requiring little or no code changes to the servlets migrated.

Both application servers are fully compliant with Sun Microsystem's Java 2 Servlet Specification, version 2.2. All servlets written to the standard specification will work correctly and require minimal migration effort.

The primary tasks involved in migrating servlets to a new environment are configuration and deployment. The use of proprietary extensions, such as htmlKona, will require additional tasks and complicate the migration effort.

The tasks involved in migrating servlets also depend on how the servlets have been packaged and deployed. Servlets can be deployed as a simple servlet, as a web application packaged with other resources in a standard directory structure, or as a web archive (WAR) file.

Differences Between WebLogic Server and Oracle9*i*AS Servlet Implementations

Oracle9*i*AS and WebLogic Server both support the Servlet 2.2 specification. Additionally, Oracle9*i*AS fully supports the finalized Servlet 2.3 specification. WebLogic Server has Servlet 2.3 support, but its implementation is based on non-finalized Servlet 2.3 specification. Hence, migrating a servlet that uses the WebLogic Server Servlet 2.3 API and features to Oracle9*i*AS may require some code upgrade to use the finalized Servlet 2.3 API.

OC4J Key Servlet Container Features

One of the key distinguishing features of OC4J is the seamless integration with Single Sign-On (SSO) and Oracle Internet Directory (OID). This is achieved through Oracle's implementation of the Java Authentication and Authorization Service (JAAS) standard - JAAS provider is integrated with OC4J.

Migrating a Simple Servlet

Simple servlets are easily configured and deployed in OC4J. The manual process used to deploy a servlet is the same in both WebLogic Server and OC4J.

Note: The recommended and preferred way to deploy a servlet is by packaging it in a WAR or EAR file and using OEM or the `dcmctl` command line utility. The manual processes described in this chapter of editing XML files and starting OC4J at the command line using the `java` command should be used for development purposes and for discussion in this chapter only.

A servlet must be registered and configured as part of a web application. To register and configure a servlet, several entries must be added to the web application deployment descriptor.

The overall steps to deploy a simple servlet are as follows (detailed steps are in Table 3–1):

1. Update the web application deployment descriptor (`web.xml`) with the name of the servlet class and the URL pattern used to resolve requests for the servlet.
2. Copy the servlet class file to the `WEB-INF/classes/` directory. If the servlet class file contains a package statement, create additional subdirectories for each level of the package statement. The servlet class file must then be placed in the lowest subdirectory created for that package.
3. Invoke the servlet from your browser by entering its URL.

To determine the effort involved in migrating servlets, we selected and migrated example servlets provided with WebLogic Server 6.0. We chose examples that did not use proprietary extensions.

Table 3–1 presents the manual process for migrating a simple servlet, HelloWorld, from WebLogic Server 6.0 to OC4J.

Table 3–1 Migrating a Simple Servlet

Step	Description	Process

1	Modify the web application deployment descriptor	Add the following to the `web.xml` file located in the `j2ee/home/default-web-app/WEB-INF/` directory of your OC4J installation:
		`<servlet>`
		`  <servlet-name>`
		`  HelloWorldServlet`
		`  </servlet-name>`
		`  <servlet-class>`
		`  examples.servlets.HelloWorldServlet`
		`  </servlet-class>`
		`</servlet>`
		`<servlet-mapping>`
		`  <servlet-name>`
		`  HelloWorldServlet`
		`  </servlet-name>`
		`  <url-pattern>`
		`  /HelloWorld/*`
		`  </url-pattern>`
		`</servlet-mapping>`
		
		Save the changes to the `web.xml` file

2	Copy the servlet class file to the appropriate directory	Copy `HelloWorldServlet.class` from the `wlserver6.0/config/examples/applications/examplesWebApp/WEB-INF/classes/examples/servlets/` directory of your WebLogic Server installation to the `j2ee/home/default-web-app/WEB-INF/classes/examples/servlets/` directory of your OC4J installation NOTE: This servlet provided with the WebLogic Server installation belongs to a package called `examples.servlets`.

Table 3–1 Migrating a Simple Servlet (Cont.)

Step	Description	Process
3	Start the OC4J J2EE application server, if not currently running	Use the OEM administration web pages or the following `dcmctl` command: `dcmctl start -i` *`9ias_instance_name`* `-ct oc4j -co` *`oc4j_instance_name`* where *`9ias_instance_name`* is the name of your Oracle9*i*AS instance and *`oc4j_instance_name`* is the name of the OC4J instance you want to start
4	Run the servlet from your web browser	Access the servlet from your web browser using the URL `http://localhost:7777/HelloWorld` (Substitute "`localhost`" with your OC4J's host name if using the browser from another machine.)

See Also: *Oracle9iAS Containers for J2EE Servlet Developer's Guide* for detailed information on configuring and deploying servlets.

Migrating a WAR File

WAR files are also easily migrated to OC4J.

A web application can be configured and deployed as a WAR file. This is easily accomplished in OC4J by using the Oracle Enterprise Manager administration GUI or manually copying the WAR file to the appropriate directory. This is also true for WebLogic Server. We will illustrate here the manual process for deploying WAR files. This manual process should only be used in development environments where OC4J is in standalone mode (not a component of an Oracle9*i*AS instance).

See Also: *Oracle9i Application Server Administrator's Guide* and *Oracle Enterprise Manager Administrator's Guide* for detailed information on using the Oracle Enterprise Manager administration screens.

Production web applications are typically deployed using WAR or EAR files through OEM or the `dcmctl` utility. Also, during the development of a web application, it may be faster to deploy and test edited code using an exploded directory format.

To deploy a WAR file in WebLogic Server, copy the WAR file into the `config/<domain_name>/applications` directory of your WebLogic Server installation. Once the WAR file is in this directory, WebLogic Server automatically deploys the web application (auto-deployment must be enabled for the domain and the Administration Server must be running).

Deploying a WAR file in OC4J is slightly different. Copy the WAR file into the `<ORACLE_HOME>/j2ee/home/applications` directory of your OC4J installation. Then, modify the application deployment descriptor found in `<ORACLE_HOME>/config/application.xml` to include the WAR file. Bind the web application to your web site by adding an entry in `<ORACLE_HOME>/config/default-web-site.xml`.

Instead of manually editing these XML files, you can also use the `admin.jar` utility. Refer to the *Oracle9iAS Containers for J2EE User's Guide* for more details.

However, when either of these methods are used, the other Oracle9*i*AS components, Distributed Configuration Manager and Oracle Process Management Notification service, are not aware of the changes. Hence, the best way would be to use Oracle Enterprise Manager or `dcmctl` utility to deploy the WAR file.

For OC4J to automatically deploy a web application, the web application must be packaged in an EAR file. Create a WAR file for the web application and package the WAR file inside an EAR file. When you modify `server.xml` (one of the descriptor files you need to modify) and save it, OC4J detects the timestamp change of the EAR file and deploys the application automatically. OC4J need not be restarted.

Note: When you package the WAR file in a EAR file, the J2EE specified manifest file (for the enterprise archive) `META-INF/application.xml` must be added to the EAR file. This manifest file may have to be created manually. See instructions below for an example of this file.

To determine the effort involved in migrating web applications packaged as WAR files, we selected and migrated example web applications provided with WebLogic Server 6.0. We chose examples that did not use proprietary extensions.

Table 3–2 presents the typical process for migrating a WAR file from WebLogic Server 6.0 to OC4J.

Table 3–2 Migrating a WAR File

Step	Description	Process
1	Modify the appropriate application deployment descriptor and save the changes	Add the following to the `application.xml` file located in the `j2ee/home/config/` directory of your OC4J installation: `<web-module id="cookie" path="..home/applications/cookie/ cookie.war" />` Save the changes to the `application.xml` file
2	Modify the appropriate `*-web-site.xml` file and save the changes	Add the following to the `http-web-site.xml` file located in the `j2ee/home/config` directory of your OC4J installation: `<web-app application="cookie" name="cookie" root="/cookie" />`
3	Modify `server.xml`	Add the following to `j2ee/home/config/server.xml`: `<application name="cookie" path="../applications/cookie.ear" />` There is no need to restart OC4J as it should pick up the new timestamp on this file automatically.

Table 3–2 Migrating a WAR File (Cont.)

Step	Description	Process

4	Prepare application files for archival to EAR file	**1.** Create a staging directory to perform the EAR file archival.
		2. Assuming you deployed the cookie example in the "examples" domain of WebLogic Server, copy `cookie.war` from the *<WL_HOME>*`/wlserver6.0/config/examples/ applications/` directory of your WebLogic Server installation into this staging directory.
		3. Create a subdirectory `META-INF` in this staging directory and create a file called `application.xml` in `META-INF`. This acts as the manifest file for the EAR and should look like the following:
		`<?xml version="1.0"?>`
		`<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application 1.2//EN" "http://java.sun.com/j2ee/dtds/application_1_2.dtd">`
		`<application>`
		`  <module>`
		`   <web>`
		`    <web-uri>cookie.war</web-uri>`
		`    <context-root>/cookie</context-root>`
		`   </web>`
		`  </module>`
		`</application>`
		4. In the same directory level as `META-INF`, run the `jar` archive utility:
		`jar cvfM cookie.ear *`

Table 3–2 Migrating a WAR File (Cont.)

Step	Description	Process
5	Copy the EAR file to the appropriate directory	Copy `cookie.ear` from the staging directory to `j2ee/home/applications/` directory of your Oracle9*i*AS installation. If OC4J is running, it will detect the new EAR file and automatically deploy it. If it is not running, use the OEM administration web pages or the following `dcmctl` command: `dcmctl start -i` *`9ias_instance_name`* `-ct oc4j -co` *`oc4j_instance_name`* where *`9ias_instance_name`* is the name of your Oracle9*i*AS instance and *`oc4j_instance_name`* is the name of the OC4J instance you want to start.
6	Invoke the web application from your web browser	Invoke the web application from your web browser using the URL `http://localhost:7777/cookie` (Substitute "`localhost`" with your OC4J's host name if using the browser from another machine.)

See Also: *Oracle9iAS Containers for J2EE Servlet Developer's Guide* and *Oracle9iAS Containers for J2EE User's Guide* for detailed information on deploying WAR files.

Migrating an Exploded Web Application

Web applications can also be configured and deployed as a collection of files stored in a standard directory structure or exploded directory format. This can be accomplished in OC4J by manually copying the contents of the standard directory structure to the appropriate directory in the OC4J installation. The same method can also be used for WebLogic Server. In this section, we will describe the manual process for deploying an exploded web application.

See Also: *Oracle9i Application Server Administrator's Guide* for detailed information on using the Oracle Enterprise Manager administration GUI.

Deploying a web application in exploded directory format is used primarily during the development of a web application. It provides a fast and easy way to deploy

and test changes. When deploying a production web application, package the web application in a WAR file and deploy the WAR file using OEM.

To manually deploy an exploded web application in WebLogic Server, copy the top-level directory containing the exploded web application files into the *<WL_HOME>*/config/*<domain_name>*/applications directory of your WebLogic Server installation. Once the top-level directory is copied to the appropriate directory, create an empty file with the name "REDEPLOY" within the top-level directory. WebLogic Server detects this file and deploys the web application. (WebLogic Server reads the timestamp of this file every few minutes to determine if the application needs redeploying. Hence, whenever an application file is updated, the REDEPLOY file's timestamp has to be updated to redeploy the file. In UNIX, this can be done by using the touch command.)

Manually deploying an exploded web application in OC4J varies slightly. Copy the top-level directory containing the exploded web application into the *<ORACLE_HOME>*/j2ee/home/applications directory of your OC4J installation. Then, modify the application deployment descriptor found in *<ORACLE_HOME>*/config/application.xml to include the web application. Bind the web application to your web site by adding an entry in *<ORACLE_HOME>*/config/default-web-site.xml (or the appropriate website XML file). Finally, register the new application in *<ORACLE_HOME>*/config/server.xml by adding a new <application> tag entry. When you modify server.xml and save it, OC4J detects the timestamp change of this file and deploys the application automatically. OC4J need not be restarted.

Migrating Configuration and Deployment Descriptors

Since WebLogic Server and Oracle9*i*AS fully support J2EE 1.2, there is a standard set of XML configuration files supported by both application servers. These are:

- **web.xml** (found in the WEB-INF directory of a web application's WAR file)
- **application.xml** (found in the META-INF directory of a web application's WAR file)
- **ejb-jar.xml** (found in the META-INF directory of an EJB module's exploded directory hierarchy)

In addition to the standard files, each application server has specific files used only by their respective environments. These are:

Oracle9*i*AS

- **`server.xml`**
  Found in `j2ee/home/config`. This is the overall OC4J runtime configuration file. It defines attributes such as the deployed applications directory, the server log file path and name, path and names of other XML files, names of applications and their EAR files, paths to runtime libraries, etc.
- **`application.xml`**
  Found in `j2ee/home/config`. This is the global configuration file common settings for all applications deployed on a particular OC4J installation. Note that this is different from the `application.xml` in a J2EE WAR file.
- ***`<website_name>`*`-web-site.xml`**
  Found in `j2ee/home/config`. This file defines a website and specifies attributes such as host name, HTTP listener port number, web applications it services and their URL contexts, and HTTP access log file and path. Note that the name and path of each `*-web-site.xml` file has to be specified in the `server.xml` file for OC4J to configure the defined website at runtime.
- **`data-sources.xml`**
  Found in `j2ee/home/config`. This file contains configuration information for data sources used by the OC4J runtime. Information in this file include: JDBC drivers used, JNDI binding for each data source, username and password for each data source, database schemas to use, maximum connections to each database, and time out values.
- **`principals.xml`**
  Found in `j2ee/home/config`. This file contains the user repository for the default `XMLUserManager` class. Groups, users belonging to them, and group permissions are defined in this file. The mapping of groups to roles is defined in the global `application.xml` file.
- **`orion-application.xml`**
  Found in `j2ee/home/application-deployments/`*`<app_name>`*. This file contains OC4J-specific information for an application (*`<app_name>`*) deployed on an OC4J installation. Web and EJB module names and security information for the application are included in the file. This file is generated by OC4J at deploy time.
- **`global-web-application.xml`**
  Found in `j2ee/home/config/`. This file contains servlet configuration information used internally by the OC4J runtime. An example is the JSP translator servlet.

- **orion-web.xml**
  Found in j2ee/home/application-deployments/*<app_name>*/*<web_app_name>*/. OC4J internal JSP and servlet information for *<web_app_name>* is specified in this file. This file is generated by OC4J at deploy time.
- **orion-ejb-jar.xml**
  Found in j2ee/home/application-deployments/*<app_name>*/*<ejb_jarfile_name>*/. This file contains OC4J internal deployment information for EJBs in the JAR file specified by *<ejb_jarfile_name>* belonging to the application *<app_name>*. This file is generated by OC4J at deploy time.
- **oc4j-connectors.xml**
  Found in j2ee/home/config/. This file contains connector information for the OC4J installation.

WebLogic Server 6.0

- **config.xml**
  Found in *<WL_HOME>*/config/*<domain_name>*/config.xml. This file contains configuration information for an entire WebLogic Server domain. Information specified in this file include the domain administration server's host name and admin port number, JNDI mappings to data sources, JDBC connection pool information, applications deployed to all nodes in the domain, SSL certificate information,
- **weblogic.xml**
  Found in *<WL_HOME>*/config/*<domain_name>*/applications/*<web_app_name>*/WEB-INF/. This file defines JSP properties, JNDI mappings, resource references, security role mappings, and HTTP session and cookie parameters for a Web application. This file is WebLogic Server-specific but is created manually.
- **weblogic-ejb-jar.xml**
  Found in an EJB module's META-INF subdirectory. This file maps WebLogic Server resources to EJBs. These resources include security role names, data sources, JMS connections, and other EJBs. This file also has performance attributes for caching and clustering for the EJBs defined in the corresponding ejb-jar.xml file.

Note: The files mentioned above are not an exhaustive list of all XML configuration file used by each application server. They are files which are relevant to the configuration and deployment of servlet applications. Other XML files also exist to configure components such as HTTP listeners, RMI, security.

Migrating Cluster Aware Applications

OC4J provides clustering features that are superior to WebLogic Server in both performance and ease of use.

WebLogic Server provides two primary cluster services, HTTP session state clustering and object clustering. The focus of this section is on HTTP session state clustering or web application clustering.

WebLogic Server supports clustering for servlets and JSP pages by replicating the HTTP session state of clients accessing clustered servlets and JSP pages. To benefit from HTTP session state clustering, you must ensure that the HTTP session state is persistent by configuring either in-memory replication, filesystem persistence, or JDBC persistence.

Oracle9*i*AS provides clustering support similar to that of WebLogic Server. In addition, Oracle9*i*AS provides:

- **Servlet Clustering**—OC4J provides facilities to cluster servlets without requiring any changes to the web application. The changes necessary are deployment configuration modifications that are transparent to the web application.
- **Clustering Architecture and Simplicity**—An important differentiator for Oracle9*i*AS is the ease with which different instances can be clustered and the robustness of the architecture used for clustering.
- **Clustering Simplicity**—Oracle Enterprise Manager (OEM) provides a GUI to configure various Oracle9*i*AS instances to belong to a single cluster, whether they are multiple servers with load balancing on a single machine or on different machines. Alternatively, you can also edit a single XML file. In contrast, it is more complex to configure WebLogic Server clusters with load balancing either with multiple instances on one machine or on multiple machines.
- **Superior Clustering Architecture—**OC4J uses dynamic IP addresses to register instances as part of a cluster. Its built-in load balancer, or any standard load

balancer such as Cisco Local Director or BigIP, has the ability to use a variety of load balancing algorithms to route requests to different instances. In contrast, WebLogic Server uses static IP addresses to configure clustering. Static IP addresses preclude the use of a load balancer to distribute requests across instances. As a result, you get either clustering or load balancing with WebLogic Server but not both.

Each Oracle9*i*AS farm consists of multiple OC4J islands and each island can consist of multiple applications. The sharing of session state for failover is within a particular island.

Use the following steps to configure Oracle9*i*AS clustering:

Note: These steps illustrate the manual method of editing XML files to configure Oracle9*i*AS. OEM provides a graphical interface to manage Oracle9*i*AS and provides a better view of the entire system. Refer to the *Oracle9i Application Server Administrator's Guide* for instructions on how to use OEM.

---

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

   Verify that the web application is installed on all nodes in the cluster. To avoid duplication of the web application, place the web application on a drive shared by the appropriate servers. Start all nodes and verify that the web application works correctly on all nodes.

2. Set up the web application to replicate its state

   Edit the OC4J specific deployment descriptor for the web application located at `j2ee/home/application-deployments/<application-name>/<web-app-name>/orion-web.xml`. If clustering is to be configured for all web applications on a web site, edit the OC4J-specific deployment descriptor for the global web application located at `j2ee/home/config/global-web-application.xml`. Edit either file by adding the `<cluster-config/>` tag after the main body of the `<orion-web-app>` tag.

   Optionally you may:

   - Specify the multicast host/IP address to transmit and receive cluster data
   - Specify the port to transmit and receive cluster data (default port is 9127)
   - Specify the id (number) of the node to identify itself with in the cluster (default is based on the IP address of the local machine)

Repeat step 2 for all the nodes in the cluster. As a result, the following will be replicated:

- The `HttpSession` data (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

It is important to understand that load balancing is providing load balancing for the web component, not the EJB. 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.

3. Configure the OC4J islands

   OC4J islands are connected to a certain web site rather than to a web application. To configure an island, edit the deployment descriptor of the web site that the web application is deployed on. For example, if the web application is deployed on the default web site, edit the deployment descriptor located at `j2ee/home/config/default-web-site.xml`. Edit the file by adding `cluster-island="1"` to the `<web-site>` tag. If the cluster has more than one island, specify different island values for servers that belong to different islands and similar values for those in the same island. Remember, HTTP session state is shared only within an island.

   Obtaining the IP address of the local host is not reliable on all platforms. The back-end needs to tell the front-end about its IP address in some other way. This is accomplished by specifying the host using the `host="host/ip"` attribute in the `<web-site>` tag of the same deployment descriptor modified earlier in this step.

4. Tell the back-end about the load balancer

   To specify where the load balancer for the web site is located, edit the same deployment descriptor modified in Step 3 by adding the following tag in the main body of the `<web-site>` tag:

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

   `balancer-host` is the hostname of the server where the load balancer is running and *`balancer-port`* is the port number of the load balancer. As this host and port make up the public hostname for the site, port 80 is suggested.

5. Make the web application distributable

To indicate that a web application is distributable, edit the web application deployment descriptor located in `WEB-INF/web.xml`. Add the `</distributable>` tag to the deployment descriptor. For example:

```
<web-app>
...
   <distributable/>
...
</web-app>
```

6. Test the Oracle9*i*AS configuration

   Access the load balancer's host and port using a web browser. Notice how the request is sent off to one of the back-end servers. Now, request the same page from the same client. More than likely, you will be sent to the same back-end node, but if the same page is requested from a different client, you will see that this client request gets balanced across to another node.

   To test state replication, request `servlet/SessionServlet`. Determine which server has become the primary server for the session by looking at the access logs. Shut that server down and request the `SessionServlet` again. If the Oracle9*i*AS configuration is correct, you will get to the same session as before but on a different node and the counter will be updated appropriately.

   See Also: "Oracle9iAS Support for Clustering and Load Balancing" on page 2-11 of this book and the *Oracle9iAS Containers for J2EE User's Guide*.

4

Migrating JSP Pages

This chapter provides the information you need to migrate JavaServer pages from WebLogic Server 6.0 to Oracle9*i*AS 9.0.2. It covers the migration of simple JSP pages, custom JSP tag libraries, and WebLogic custom tags.

This chapter contains these topics:

Introduction

Migrating JSP pages from WebLogic Server 6.0 to Oracle9*i*AS is straight forward and requires little or no code changes.

Both application servers are fully compliant with Sun Microsystem's JavaServer Page specifications, version 1.1 and 1.2. All JSP pages written to the standard specification will work correctly and require minimal migration effort.

The primary tasks involved in migrating JSP pages to a new environment are configuration and deployment. The use of proprietary extensions and tag libraries will require additional tasks and complicate the migration effort.

The tasks involved in migrating JSP pages also depend on how the JSP pages have been packaged and deployed. JSP pages can be deployed as a simple JSP page, as a web application packaged with other resources in a standard directory structure, or as a enterprise application archive (EAR) file. The migration of web applications in exploded directory format and EAR file format is addressed in Chapter 3, "Migrating Java Servlets".

Differences Between WebLogic Server and Oracle9iAS JSP Implementations

Since both WebLogic Server and Oracle9*i*AS Containers for J2EE (OC4J) have implemented the same versions of the Java ServerPages specifications, there are no differences between the two in the core areas. There are a few differences in the JSP 1.2 features each server supports. Each vendor also provides their own JSP custom tags. WebLogic Server provides three specialized JSP tags - `cache`, `repeat`, and `process` - that you can use in your JSP pages. OC4J also provides various JSP tags - Oracle JSP Markup Language (JML) Custom Tag Library, tags for XML and XSL integration, and several JSP utility tags. A comprehensive discussion of these tags can be found in *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

OC4J JSP Features

Oracle9*i*AS 9.0.2 provides one of the fastest JSP engines on the market. Further, it also provides several value-added features and enhancements such as support for globalization and SQLJ. For those of you who are familiar with Oracle9*i*AS 1.0.2.2, the first release of Oracle9*i*AS to include OC4J, there were two JSP containers: a container developed by Oracle and formerly known as OracleJSP and a container licensed from Ironflare AB and formerly known as the "Orion JSP container".

In Oracle9*i*AS 9.0.2, these have been integrated into a single JSP container, referred to as the "OC4J JSP container". This new container offers the best features of both previous versions, runs efficiently as a servlet in the OC4J servlet container, and is

well integrated with other OC4J containers. The integrated container primarily consists of the OracleJSP translator and the Orion container runtime running with a new simplified dispatcher and the OC4J 1.0.2.2 core runtime classes. The result is one of the fastest JSP engines on the market with additional functionality over the standard JSP specifications.

OC4J JSP provides extended functionality through custom tag libraries and custom JavaBeans and classes that are generally portable to other JSP environments:

- Extended types implemented as JavaBeans that can have a specified scope
- `JspScopeListener` for event handling
- Integration with XML and XSL through custom tags
- 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
- Additional utility tags for functionality such as uploading or downloading files or sending e-mail
- JESI (Edge Side Includes for Java) tags and Web Object Cache tags and API that work with content delivery network edge servers to provide an intelligent caching solution for web content.

> **See Also:** *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference* for detailed information on custom JSP tag libraries.

The OC4J JSP container also offers several important features such as the ability to switch modes for automatic page recompilation and class reloading, JSP instance pooling, and tag handler instance pooling.

Oracle JDeveloper and OC4J JSP Container

Oracle JDeveloper is integrated with the OC4J JSP container to support the full JSP application development cycle - editing, debugging, and running JSP pages. It also provides an extensive set of data-enabled and web-enabled JavaBeans, known as JDeveloper web beans and a JSP element wizard which offers a convenient way to add predefined web beans to a page. JDeveloper also provides a distinct feature that is very popular with developers. It allows you to set breakpoints within JSP page source and can follow calls from JSP pages into JavaBeans. This is much more

convenient than manual debugging techniques, such as adding print statements within the JSP page to output state into the response stream for display on browser or to the server log.

Migrating a Simple JSP Page

Simple JSP pages are easily configured and deployed in OC4J. The process used to deploy a JSP page is similar in both WebLogic Server and OC4J.

JSP pages do not require specific mappings as do HTTP servlets. To deploy a simple JSP page, you can copy the JSP page and any files required by the JSP page to the appropriate directories. No additional registrations are required.

> **Note:** OEM should be used to deploy any type of applications including JSPs. But for the purpose of this discussion here, we are copying JSP files manually without using OEM.

The deployment process has been simplified in OC4J by providing a J2EE web application and various configuration files by default.

To determine the effort involved in migrating JSP pages, we selected and migrated example JSP pages provided with WebLogic Server 6.0. We chose examples that did not use proprietary extensions.

Table 4–1 presents the typical process for migrating a simple JSP page from WebLogic Server 6.0 to OC4J.

Table 4–1 Migrating a Simple JSP Page

Step	Description	Process
1	Start an instance of OC4J, if none are currently running.	Go to `http://localhost:1810` and select the OC4J instance you want to start. Or, use the following `dcmctl` command: `dcmctl start -i` *`9ias_instance_name`* `-ct oc4j -co` *`oc4j_instance_name`* where *`9ias_instance_name`* is the name of your Oracle9*i*AS instance and *`oc4j_instance_name`* is the name of the OC4J instance you want to start.

Table 4–1 Migrating a Simple JSP Page (Cont.)

Step	Description	Process
2	Copy the JSP page to the appropriate directory	Copy `HelloWorld.jsp` from `wlserver6.0/samples/examples/jsp/` of your WebLogic Server installation to `j2ee/home/default-web-app/` of your OC4J installation
3	Copy any files required by the JSP page	Copy `BEA_Button_Final_web.gif` from `wlserver6.0/config/examples/applications/examplesWebApp/images/` of your WebLogic Server installation to `j2ee/home/default-web-app/images/` of your OC4J installation. Note that you may have to create the `images` directory
4	Request the JSP page from your web browser	From your web browser, request the JSP page through the URL `http://localhost:7777/j2ee/HelloWorld.jsp`

See Also: *Oracle9iAS Containers for J2EE Support for JavaServer Pages Reference* and *Oracle9iAS Containers for J2EE User's Guide* for detailed information on configuring and deploying JSP pages.

Migrating a Custom JSP Tag Library

WebLogic Server and OC4J provide the ability to create and use custom JSP tags. The process used to deploy a custom JSP tag library is similar for both WebLogic Server and OC4J.

Tag libraries can be packaged and deployed as part of a web application, and are declared in a specific section of the web application deployment descriptor.

To determine the effort involved in migrating custom JSP tag libraries, we selected and migrated example JSP pages provided with WebLogic Server 6.0. We chose examples that did not use proprietary extensions.

Table 4–2 presents the typical process for migrating a JSP page that utilizes a custom JSP tag library from WebLogic Server 6.0 to OC4J.

Table 4–2 Migrating a Custom JSP Tag Library

Step	Description	Process

1	Copy the tag library file to the appropriate directory	Copy `counter.tld` from
		`wlserver6.0/samples/examples/jsp/tagext/counter/`
		of the WebLogic Server installation to
		`j2ee/home/default-web-app/WEB-INF/`
		of your OC4J installation

2	Copy the JSP page to the appropriate directory	Copy `pagehits.jsp` from
		`wlserver6.0/samples/examples/jsp/tagext/counter/`
		of the WebLogic Server installation to
		`j2ee/home/default-web-app/`
		of your OC4J installation

3	Copy any class files required by the tag library and used by the JSP file to the appropriate directory	Copy `Count.class`, `Display.class`, and `Increment.class` from
		`wlserver6.0/config/examples/applications/examplesWebApp/WEB-INF/classes/examples/jsp/tagext/counter/`
		of the WebLogic Server installation to
		`j2ee/home/default-web-app/WEB-INF/classes/examples/jsp/tagext/counter/`
		of your OC4J installation
		Note that these `.class` files provided with the WebLogic server installation belong to a package called `examples.jsp.tagext.counter`. You may need to create the `examples/jsp/tagext/counter/` directory.

Table 4–2 Migrating a Custom JSP Tag Library (Cont.)

Step	Description	Process

4	Copy image files used by the JSP file	Copy the directory containing the image files from
		`wlserver6.0/samples/examples/jsp/tagext/counter/images/numbers/`
		of the WebLogic Server installation to
		`j2ee/home/default-web-app/images/numbers/`
		of your OC4J installation.
		Note that you may have to create the `images/numbers` directory

5	Modify the appropriate web application deployment descriptor and save the changes	Add the following to the `web.xml` file located in the
		`j2ee/home/default-web-app/WEB-INF/`
		directory of your OC4J installation (`<taglib>` is a child element of `<web-app>`):
		`<taglib>`
		`  <taglib-uri>`
		`  counter`
		`  </taglib-uri>`
		`  <taglib-location>`
		`  /WEB-INF/counter.tld`
		`  </taglib-location>`
		`</taglib>`

6	Start the OC4J instance, if it is not currently running.	Go to `http://localhost:1810` and select the OC4J instance you want to start. Or, use the following `dcmctl` command: `dcmctl start -i` *`9ias_instance_name`* `-ct oc4j -co` *`oc4j_instance_name`* where *`9ias_instance_name`* is the name of your Oracle9*i*AS instance and *`oc4j_instance_name`* is the name of the OC4J instance you want to start.
7	Request the JSP file from your web browser	From your web browser, use the URL `http://localhost:7777/j2ee/pagehits.jsp`

See Also:

- *Oracle9iAS Containers for J2EE Support for JavaServer Pages Reference* for detailed information on configuring and deploying JSP pages.
- *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference* for detailed information on custom JSP tag libraries.

Migrating from WebLogic Custom Tags

If WebLogic custom tags are used extensively throughout your web application, then the best option is to use the WebLogic tag library by deploying it on OC4J. This option is discussed in the previous section, "Migrating a Custom JSP Tag Library". You can then migrate to the Oracle JSP tags if required. In the future, tools or automated scripts may be available to make the code conversion easier.

If WebLogic custom tags are used sparingly throughout your web application, then the best option is to modify the JSP pages to use the Oracle JSP tag library. This option is discussed below.

WebLogic Server provides three specialized JSP tags for use in JSP pages. They are `cache`, `process`, and `repeat`.

WebLogic Server `cache` Tag

OC4J provides a superset of the WebLogic Server `cache` tag in the form of Web Object Cache Tags. These tags provide additional functionality over the WebLogic `cache` tag. Further, the Web Object Cache Tags of OC4J are well integrated with other tag libraries such as the XML tag library. For example, the `cacheXMLObj` tag is well integrated with OC4J's XML tags.

One feature which does not have direct functionality mapping is "async". However, Edge Side Includes (ESI) and Edge Side Includes for Java (JESI) can provide similar functionality to it.

See Also: Chapter 6 and Chapter 7 of *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference* for detailed information on Web Object Cache tags and JESI tags.

WebLogic Server `process` Tag

OC4J does not have an exact equivalent for the `process` tag. The closest option is to use the `jml:useForm` and `jml:if` tags. from Oracle's JSP Markup Language (JML).

> **See Also:** Bean Binding Tag Descriptions and Logic and Flow Control Tag Descriptions subsections in the JSP Markup Language (JML) Tag Descriptions section of Chapter 3 of *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference* for detailed information on these JML tags.

Alternatively, you could write Java code to implement the tag.

WebLogic Server `repeat` Tag

The OC4J equivalent for this tag is the `jml:foreach` tag. This tag provides the ability to iterate over a homogeneous set of values. The body of the tag is executed once per element in the set. This tag currently supports iterations over the following types of data structures:

- Java array
- `java.util.Enumeration`
- `java.util.Vector`

However, these tags do not cover data structures such as Iterators, Collections, and the keys of a hashtable.

> **See Also:** The Logic and Flow Control Tag Descriptions subsection in the JSP Markup Language (JML) Tag Descriptions section of Chapter 3 of *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference* for detailed information on this JML tag.

For `ResultSets` and `ResultSetMetaData`, OC4J provides tags called the SQL Tags for Data Access. These tags provide functionality very similar to that provided by the WebLogic Server `repeat` tag. The `dbNextRow` tag is the tag that you are likely to be most interested in. This tag can be used to process each row of a result set obtained in a `dbQuery` tag and associated with the specified `queryId`. Place the processing code in the tag body, between the `dbNextRow` start and end tags. The code in the body is executed for each row of the result set.

See Also:

- The Custom Data-Access Tag Library subsection in the SQL Tags for Data Access section of Chapter 4 of *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference* for detailed information on these JML tags.
- *Oracle9iAS Containers for J2EE Support for JavaServer Pages Reference* for detailed information about the standard JSP tag library framework and tag-extra-info classes.

Precompiling JSP Pages

JSP pages are compiled automatically by the JSP compiler. However, when testing and debugging JSP pages, you may want to access the JSP compiler directly.

The JSP compiler parses a `.jsp` file into a `.java` file. The standard Java compiler is then used to compile the `.java` file into a `.class` file.

Using the WebLogic Server JSP Compiler

To start the WebLogic Server JSP compiler, type the following command:

```
java weblogic.jspc -options fileName
```

The *`fileName`* parameter refers to the name of the JSP page to be compiled. Options may be specified before or after the JSP page name. The following example demonstrates the use of the `-d` option to compile `myFile.jsp` into the destination directory `weblogic/classes`:

```
java weblogic.jspc -d /weblogic/classes myFile.jsp
```

Using the OC4J JSP Pre-translator

In addition to the standard `jsp_precompile` mechanism, OC4J provides a command-line utility called `ojspc` for pretranslating JSP pages.

Consider the example where the JSP page, `HelloWorld.jsp`, is located in the OC4J default web application directory `j2ee/home/default-web-app/examples/jsp/`. (Copy the HelloWorld.jsp file from `j2ee/home/default-web-app/` to this subdirectory.)

To pre-translate this JSP page, set your current directory to the application root directory, then, in `ojspc`, set the `_pages` directory as the output base directory

using the `-d` option. This results in the appropriate package name and file hierarchy. To illustrate (assume % is a UNIX prompt):

```
% cd j2ee/home/default-web-app
% ojspc -d ../application-deployments/default/defaultWebApp/temp/_pages
   examples/jsp/HelloWorld.jsp
```

The directory structure above specifies an application-relative path of `examples/jsp/HelloWorld.jsp`. The translated JSP can be found in `j2ee/home/application-deployments/default/defaultWebApp/temp/_pages/_examples/_jsp/`.

At execution time, the JSP container looks for compiled JSP files in the `_pages` subdirectory. The `_examples/_jsp/` subdirectory would be created automatically by `ojspc` if run as in the above example.

Invoke the JSP page through the URL `http://localhost:7777/j2ee/examples/jsp/HelloWorld.jsp`. Notice that response time is faster than without pre-translating.

Standard JSP Pre-translation Without Execution (based on the JSP 1.1 specification)

You can specify JSP pre-translation, without execution, by enabling the standard `jsp_precompile` request parameter when invoking a JSP page from the browser. For instance, `http://hostname:port/foo.jsp?jsp_precompile=true`

Using the `j2ee/home/default-web-app/HelloWorld.jsp` file as an example, erase all the "`_HelloWorld*`" files in `j2ee/home/application-deployments/default/defaultWebApp/temp/_pages/`. Then, invoke the URL `http://localhost:7777/j2ee/HelloWorld.jsp?jsp_precompile=true`. The pre-translation is performed but the page does not appear on your browser. Check the `_pages` subdirectory for the translated files.

Configure the JSP Container for Execution with Binary Files Only

You can avoid exposing your JSP page source, for proprietary or security reasons, by pre-translating the pages and deploying only the translated and compiled binary files. JSP pages that are pre-translated, either from previous execution in an on-demand translation scenario or by using `ojspc`, can be deployed to any standard J2EE environment.

For further details, refer to the *Oracle9iAS Containers for J2EE Support for JavaServer Pages Reference*.

5

Migrating Enterprise JavaBean Components

This chapter provides the information you need to migrate Enterprise JavaBean components from WebLogic Server 6.0 to Oracle9*i*AS 9.0.2. It addresses the migration of simple EJB JARs, as well as J2EE web applications in the form of EAR files or in an exploded directory format.

This chapter contains these topics:

Introduction

Migrating Enterprise JavaBeans (EJB) from WebLogic Server 6.0 to Oracle9*i*AS 9.0.2 is straightforward, requiring little or no code changes to the EJBs migrated. Both application servers support the EJB 1.1 specification. For EJB 2.0 however, WebLogic Server's implementation is based on a non-final version of the EJB 2.0 specifications while Oracle9*i*AS's supports the final EJB 2.0 specifications.

All EJBs written to the EJB 1.1 standard specifications will work correctly and require minimal migration effort. The primary effort goes into configuring and deploying the applications in the new environment. Only in cases where proprietary extensions are used will the migration effort get complex. Additionally, if your WebLogic Server EJBs use 2.0 APIs and features, you may need to upgrade your EJB code to the final EJB 2.0 specifications.

In this chapter we cover the migration of EJBs deployed in the form of EAR files or in an exploded directory format.

Differences Between WebLogic Server and Oracle9*i*AS EJB Implementations

Since both WebLogic Server and Oracle9*i*AS Containers for J2EE (OC4J) have implemented the same versions of the Enterprise JavaBeans specifications, there are no differences between the two in the core areas. There are a few differences in the EJB 2.0 features each server supports. OC4J also supports more features than WebLogic Server 6.0.

EJB Container Facilities

Oracle9*i*AS has a complete EJB 1.1 implementation including session beans (both stateful and stateless) and entity beans (supporting both container managed persistence and bean managed persistence). Further, Oracle9*i*AS has support for EJB 2.0 including:

- XML deployment descriptors
- Message driven beans
- EJB 2.0 object-relational (O-R) mapping
- Support for several of the elements of the new container-managed persistence (CMP) architecture

More Efficient Container Managed Persistence

There are two specific facts that reflect the significant performance advantages in using Oracle9*i*AS' container-managed persistence (CMP) implementation compared to WebLogic Server's implementation:

- **Automatic Detection of Modified EJBs**—When using CMP, Oracle9*i*AS' J2EE container can automatically detect whether you have modified an EJB and writes the EJB's state to the database; it does an `ejbStore` only when necessary. WebLogic Server does not provide such automatic detection requiring a user to code is-modified methods which the WebLogic Server container uses to know whether or not to do the `ejbStore` operation.
- **Simple and Complex DB mapping for CMP**—When using CMP, Oracle9*i*AS' J2EE container supports both simple (1:1, 1:many) and complex (many:many) database field mappings very efficiently. In contrast, WebLogic Server provides rudimentary support for simple CMP database field mapping (1:many) in its CMP EJB 1.1 XML files. For instance, it is difficult to qualify a `where` clause string in WebLogic Server and this results in doing unnecessary full table scans.

Clustering Support

Application server clustering essentially means the use of a group of application servers that coordinate their actions in order to provide scalable, highly available services in a transparent manner.

From a comparative point of view, Oracle9*i*AS' J2EE container provides the following facilities:

- **Servlet Clustering**—Oracle9*i*AS provides facilities to cluster servlets without requiring any changes to the user's application. The changes are deployment configuration modifications which are transparent to the J2EE application.
- **Clustering Architecture and Simplicity**—An important differentiator for Oracle9*i*AS' J2EE container is the ease with which different instances can be clustered and the robustness of the architecture used for clustering. Specifically, Oracle9*i*AS requires a user to edit a single XML file to configure various Oracle9*i*AS instances to belong to a single cluster / island whether they are multiple servers with load balancing on a single machine or multiple servers with load balancing on different machines. In contrast, it is much more complex to configure WebLogic Server clusters with load balancing either with multiple instances on one machine or on multiple machines. For instance, if you indicate that your EJBs will be used in a cluster, then you need to specify it during the time the EJB stubs are created using `ejbc`, which then results in the creation of special cluster-aware classes that will be used for deployment. Overall,

Oracle9*i*AS' J2EE container provides a more robust clustering architecture with better ease-of-use.

- **Stateless Session Bean Clustering**—Oracle9*i*AS supports clustering of stateless session beans.
- **Stateful Session Bean and Entity Bean Clustering**—Oracle9*i*AS supports clustering of stateful session beans and entity beans. Two aspects of design are focused upon:
  - Clustered Performance—Existing clustering facilities such as those in WebLogic Server 6.0 impose a severe performance penalty when running the instances in a stateful fashion with clustering. As a result, most users choose to keep their middle tier completely stateless and write their state to a persistent store, for example, a database. In delivering clustered EJBs, Oracle is working on optimizing the EJB clustering implementation to avoid introducing performance penalties.
  - Programmatic Simplicity—Additionally, unlike servlets which have a natural session boundary at which to failover their state, EJBs do not have such a clear boundary. As a result, we are working on simple programmatic facilities to allow developers to use EJB clustering without any changes to their applications.

Security and LDAP Integration

One of the key distinguishing features of OC4J is the seamless integration with Single Sign On (SSO) and Oracle Internet Directory (OID). This is achieved through Oracle's implementation of the Java Authentication and Authorization Service (JAAS) standard. See *Oracle9iAS Containers for J2EE User's Guide* and *Oracle9i Application Server Security Guide*.

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 EJBs do offer portability, there are still a number of nonportable, implementation-specific aspects that need to be addressed when migrating components from one platform to another. Typically, an EJB component requires low level interfaces with the container in the form of stub and skeleton classes that will need to stay implementation-specific. In effect, a clear partitioning between portable and nonportable 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 platform.
- 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 because mappings may not be known in advance. Also, they may have dependencies on the structure and population of the user directory on the target deployment server.

Migration Steps

The tasks involved in migrating EJBs are best analyzed by looking at the steps required for deploying EJBs to an EJB container:

- Setting the EJB deployment descriptors, particularly the vendor-specific deployment descriptors
- Generating EJB container classes
- Loading EJB classes in the server
- Deploying the EJBs in the form of an EAR file or in an exploded directory format
- Configuring the EJBs for deployment at startup

We can address the migration tasks along the same lines.

Setting Deployment Properties

The deployment process starts with a JAR file or a J2EE standard deployment directory that contains the compiled EJB interfaces and implementation classes created by the EJB provider. There should also be an EJB-compliant `ejb-jar.xml` file that describes the bundled EJB(s). The `ejb-jar.xml` file and other required XML deployment files, typically the vendor-specific deployment descriptors, must reside in a top level `META-INF` directory of the JAR file or deployment directory as follows:

WebLogic EJB JAR Structure

```
<EJB Module Name>
   |
   |-- *.class
   |   <remote>.class
   |   <home>.class
   |
   |-- META-INF
   |      |
   |      |-- ejb-jar.xml
   |          weblogic-ejb-jar.xml
   |          weblogic-cmp-rdbms-jar.xml
```

Oracle9iAS EJB JAR Structure

```
<EJB Module Name>
   |
   |-- *.class
   |   <remote>.class
   |   <home>.class
   |
   |-- META-INF
   |      |
   |      |-- ejb-jar.xml
   |          orion-ejb-jar.xml
```

Vendor-specific Deployment Descriptors

WebLogic Server You would have first created and configured the WebLogic Server-specific and mandatory deployment descriptor, `weblogic-ejb-jar.xml`, and then added the file to the deployment file or directory. The `weblogic-ejb-jar.xml` file is used for specifying caching, clustering, and performance behavior.

If you were deploying an entity EJB that used container managed persistence, you would have also included an additional deployment file for specifying the O-R mapping details, or, in other words, the RDBMS-based persistence services in a file called `weblogic-cmp-rdbms-jar.xml`. A separate file would have been required for each bean that used RDBMS persistence.

OC4J In the case of OC4J, only one file is required. You first create and configure the OC4J-specific and mandatory deployment descriptor, `orion-ejb-jar.xml`, and then add the file to the deployment file or directory. The `orion-ejb-jar.xml` file is used for defining caching, clustering, and performance behavior. The details on

O-R mapping or the RDBMS-based persistence services are also specified in the `orion-ejb-jar.xml` file. This is different from WebLogic Server where two separate files were required.

Generating and Deploying EJB Container Classes

The next step after compiling the EJB classes and adding the required XML deployment descriptors (the J2EE deployment descriptor as well as the vendor-specific deployment descriptors) is generation of the container classes that are used to access the EJB. The container classes include implementation of the external interfaces (home and remote) that clients use, as well as the classes that the application server uses, for the internal representation of the EJBs.

WebLogic Server In WebLogic Server, you would have used the `ejbc` compiler to generate container classes according to the deployment properties specified in the WebLogic Server-specific XML deployment files. For example, if you indicate that your EJBs will be used in a cluster, `ejbc` creates special cluster-aware classes that will be used for deployment. You can also use `ejbc` directly from the command line by supplying the required options and arguments.

Once the container classes have been generated, you need to package the classes into a JAR or EAR file and deploy the classes using the console GUI.

OC4J For OC4J, explicit compilation is not required. The EJB JAR file is packaged into a EAR file (together with a WAR file, if any). Then, you can use the Oracle Enterprise Manager (OEM) GUI to specify the EAR file for deployment. The container classes are generated for OC4J and any J2EE Web application in the EAR file is bound to the OC4J container.

Alternatively, you can also use the `admin.jar` utility to deploy the EAR file if you are using OC4J in standalone mode. You would then need to update the OEM environment using the `dcmctl` utility. Refer to the *Oracle9iAS Containers for J2EE User's Guide* for more information.

Loading EJB Classes in the Server

WebLogic Server The final step in deploying an EJB involves loading the generated container classes into WebLogic Server. However, you can prompt WebLogic Server to automatically load EJB classes by starting WebLogic Server. This places the EJB in the deployment directory where it is automatically deployed when the server is started.

OC4J Similarly, you can specify classes belonging to an application to be loaded when OC4J starts by specifying the `auto-start="true"` parameter in the `<application>` tag in `server.xml`.

Migrating EJBs in a EAR or JAR File

EAR and JAR files containing EJBs which are deployed in WebLogic Server can be migrated to Oracle9*i*AS. However, you should un-archive and re-archive the EAR file to ensure its contents are complete and that the XML descriptors have the correct entries. Use the following points as a guideline:

- Ensure that the EJB client XML descriptors specify the JNDI names of the EJB stubs. If the client is a Web application, the JNDI names should be specified in `web.xml`. If the client is standalone, the names should be specified in `application-client.xml`.
- For the case where the EJB client is standalone, the client classes and XML descriptor file, `application-client.jar`, should be archived into a JAR file, which in turn should be archived into the EAR file where the EJBs are.
- If the EJB(s) to be migrated from WebLogic are in a JAR file, you need to repackage them in a EAR file with the EAR's `application.xml`.
- Deploy the EAR file on Oracle9*i*AS using OEM or `dcmctl`.
- You do not need to pre-compile EJB stubs using `ejbc`, `rmic`, or other such facilities into the client application. The OC4J EJB container generates EJB stubs on demand as it needs them. This makes application and system maintenance significantly more straightforward than WebLogic Server.

Migrating an Exploded EJB Application

EJB applications can also be deployed as a collection of files that use a standard directory structure defined in the J2EE specification. This type of deployment deploys applications in an exploded directory format. Deploying an EJB application in exploded directory format is done most often whilst developing your application and only for standalone OC4J instances. This is because the exploded directory format is more suitable for developers to modify source files and test the application quickly. In Oracle9*i*AS production environments, however, the application should be packaged in a EAR file and deployed using OEM or `dcmctl`.

When deploying an exploded directory structure to WebLogic Server, you would have copied the top level directory containing an EJB application in exploded directory format into the `mydomain/config/applications/` directory of your

WebLogic Server distribution (where `mydomain` is the name of your domain). Once copied, WebLogic Server automatically deploys the EJB application.

For OC4J, copy the top level directory containing the EJB application in exploded directory format into the `j2ee/home/applications/` directory of the OC4J installation. Then, modify the default J2EE application deployment descriptor, `server.xml`, located in the `j2ee/home/config/` directory to include your EJB module.

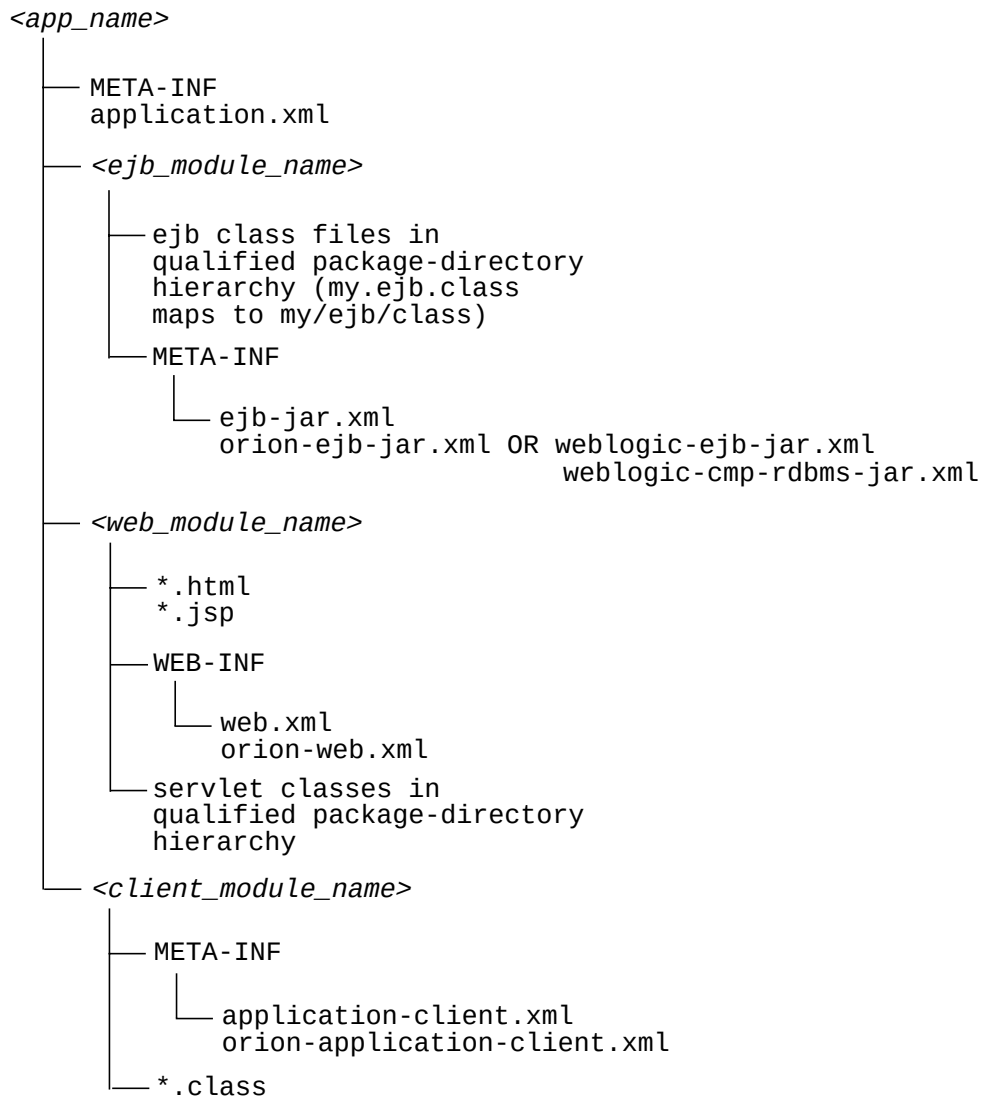
In WebLogic Server, if a file is modified using the administration console, or otherwise, it requires a server restart before the updated configuration is picked up. In the case of OC4J, the timestamp change for `server.xml` will cause OC4J to effect the changes in the XML file.

Configuring EJBs using Deployment Descriptors

There are typically two deployment descriptors that are used to configure and deploy EJBs. The first deployment descriptor, `ejb-jar.xml`, is defined in the EJB 1.1 specification and provides a standardized format that describes the EJB application. The second deployment descriptor is a vendor-specific deployment descriptor that maps resources defined in the `ejb-jar.xml` file to resources in the vendor's application server. It is also used to define other aspects of the EJB container such as EJB behavior, caching, and some vendor-specific features.

The WebLogic Server-specific deployment descriptors are `weblogic-ejb-jar.xml` and `weblogic-cmp-rdbms-jar.xml`, and the OC4J-specific deployment descriptor is `orion-ejb-jar.xml`.

A typical J2EE application directory structure would look like this:



The WebLogic Server-specific deployment descriptor, `weblogic-ejb-jar.xml`, defines EJB deployment descriptor DTDs which are unique to WebLogic Server. The EJB 2.0 container uses a version of `weblogic-ejb-jar.xml` that is different from the one shipped with WebLogic Server 5.1. The revised DTD for `weblogic-ejb-jar.xml` includes new elements for enabling stateful session EJB replication, configuring entity EJB locking behavior, and assigning JMS Queue and

Topic names for message-driven beans. The new DTD also reorganizes the major stanzas into more logical sections. The earlier `weblogic-ejb-jar.xml` DTD can be used for EJB 1.1-compliant EJBs that you deploy into the EJB 1.1 container.

Elements configured in the EJB `weblogic-ejb-jar.xml` include:

- `weblogic-enterprise-bean`
  - `ejb-name`
  - `entity-descriptor`
  - `stateless-session-descriptor`
  - `stateful-session-descriptor`
  - `message-driven-descriptor`
  - `transaction-descriptor`
  - `reference-descriptor`
  - `enable-call-by-reference`
  - `jndi-name`
- `Security-role-assignment`
- `transaction-isolation`

The WebLogic Server-specific deployment descriptor, `weblogic-cmp-rdbms-jar.xml`, defines deployment properties for an entity EJB that uses WebLogic Server RDBMS-based persistence services. The EJB 2.0 container uses a version of `weblogic-cmp-rdbms-jar.xml` that is different from the one shipped with WebLogic Server 5.1. The earlier `weblogic-cmp-rdbms-jar.xml` DTD can be used for EJB 1.1 beans that you deploy on WebLogic Server 6.0. However, if you want to use any of the new CMP 2.0 features, you must use the DTD described in the later version.

Each `weblogic-cmp-rdbms-jar.xml` defines the following persistence options:

- EJB connection pools or data source for 2.0 CMP
- EJB field-to-database-element mappings
- Finder method definitions (CMP 1.1)
- Foreign key mappings for relationships
- WebLogic Server-specific deployment descriptors for queries

The OC4J-specific deployment descriptor, `orion-ejb-jar.xml`, contains extended deployment information for session beans, entity beans, message driven beans, and security.

An entity EJB can save its state in any transactional or nontransactional persistent storage (bean-managed persistence), or it can ask the container to save its non-transient instance variables automatically (container-managed persistence). WebLogic Server and OC4J allow both choices and a mixture of the two.

In the case of an EJB that uses container-managed persistence, the `weblogic-ejb-jar.xml` or the `orion-ejb-jar.xml` deployment file specifies the type of persistence services that an EJB uses. In the case of WebLogic Server, the automatic persistence services requires the use of additional deployment files to specify their deployment descriptors, and to define entity EJB finder methods. WebLogic Server RDBMS-based persistence services obtain deployment descriptors and finder definitions from a particular bean using the bean's `weblogic-cmp-rdbms-jar.xml` file. This configuration file must be referenced in the `weblogic-ejb-jar.xml` file. In the case of OC4J, the type of persistence service as well as the details regarding the RDBMS-based persistence services are configured and obtained from the same deployment descriptor - `orion-ejb-jar.xml`.

Some of the attributes such as Development Mode are unique to OC4J.

> **See Also:** *Oracle9iAS Containers for J2EE Enterprise JavaBeans Developer's Guide and Reference* for more information on the attributes.

Writing Finders for RDBMS Persistence

For EJBs that use RDBMS persistence, WebLogic Server 6.0 provides a way to write dynamic finders. The EJB provider writes the method signature of a finder in the `EJBHome` interface, and defines the finder's query expressions in the `ejb-jar.xml` deployment file. The `ejbc` compiler creates implementations of the finder methods at deployment time, using the queries in `ejb-jar.xml`.

The key components of a finder for RDBMS persistence are:

- The finder method signature in `EJBHome`
- A query stanza defined within `ejb-jar.xml`
- An optional WebLogic Server query stanza within `weblogic-cmp-rdbms-jar.xml`

OC4J simplifies the whole process by automatically generating the finder methods.

Specifying the `findByPrimaryKey` method is easy to do in OC4J. All the fields for defining a simple or complex primary key are specified within the `ejb-jar.xml` deployment descriptor. To define other finder methods in a CMP entity bean, do the following:

1. Add the finder method to the home interface
2. Add the finder method definition to the OC4J-specific deployment descriptor—the `orion-ejb-jar.xml` file

WebLogic Query Language (WLQL)

In WebLogic Server 5.1 and 6.0, each finder query stanza in the `weblogic-cmp-rdbms-jar.xml` file had to include a WLQL string that defines the query used to return EJBs.

With the emergence of EJB Query Language (EQL), which is a standard based on the EJB 2.0 specification, use of WLQL is deprecated, and is not supported by WebLogic Server in later releases.

The first version of Oracle9*i*AS 9.0.2 does not provide support for EQL. Subsequent versions will provide it.

Message Driven Beans

In WebLogic Server, in addition to the new `ejb-jar.xml` elements, the `weblogic-ejb-jar.xml` file includes only one new message-driven-descriptor stanza to associate the message-driven bean with an actual destination in WebLogic Server. The XML element is `destination-jndi-name`.

In OC4J, to create a message-driven bean, you perform the following steps:

1. Implement a message-driven bean as defined in the EJB specification
2. Create the message-driven bean deployment descriptors
3. Configure the JMS `Destination` type (queue or topic) in the OC4J JMS XML file, `jms.xml`.
4. Map the JMS `Destination` type to the message-driven bean in the OC4J-specific deployment descriptor, `orion-ejb-jar.xml`
5. If a database is involved in your message-driven bean application, configure the data source that represents your database in `data-sources.xml`.

6. Create an EJB JAR file containing the bean and the deployment descriptor; once created, configure the `application.xml` file, create an EAR file, and deploy the EJB in OC4J.

Configuring Security

Security can be handled by the application server, or it can be incorporated programmatically into your EJB classes. Both WebLogic Server and OC4J provide similar support for security such as authentication, authorization, and digital certificates.

> **See Also:** Configuring Security in Chapter 6 of *Oracle9iAS Containers for J2EE Enterprise JavaBeans Developer's Guide and Reference* for more information.

Migrating Cluster-Aware Applications to OC4J

Oracle9*i*AS provides clustering features that are superior to WebLogic Server in performance as well as ease of use. Further, migration cluster-aware applications from WebLogic Server to OC4J is straightforward.

EJB Clustering in WebLogic Server

In-Memory Replication for Stateful Session EJBs

The WebLogic Server EJB container introduces new clustering support for stateful session EJBs. Whereas in WebLogic Server 5.1 only the `EJBHome` is clustered for stateful session EJBs, the EJB container can also replicate the state of the EJB across clustered WebLogic Server instances.

Replication support for stateful session EJBs is transparent to clients of the EJB. When a stateful session EJB is deployed, WebLogic Server creates a cluster-aware `EJBHome` stub and a replica-aware `EJBObject` stub for the stateful session EJB. The `EJBObject` stub maintains a list of the primary WebLogic Server instance on which the EJB instance runs and the name of a secondary WebLogic Server to use for replicating the bean's state.

Each time a client of the EJB commits a transaction that modifies the EJB's state, WebLogic Server replicates the bean's state to the secondary server instance. Replication of the bean's state occurs directly in memory, for best performance in a clustered environment.

Should the primary server instance fail, the client's next method invocation is automatically transferred to the EJB instance on the secondary server. The secondary server becomes the primary WebLogic Server for the EJB instance, and a new secondary server is used to account for the possibility of additional failovers. Should the EJB's secondary server fail, WebLogic Server enlists a new secondary server instance from the cluster.

By replicating the state of a stateful session EJB, clients are generally guaranteed to have the last committed state of the EJB, even if the primary WebLogic Server instance fails. However, in certain rare failover scenarios, the last committed state may not be available. This can happen when:

- A client commits a transaction involving a stateful EJB, but the primary WebLogic Server fails before the EJB's state is replicated. In this scenario, the client's next method invocation will work against the previous committed state, if available.
- A client creates an instance of a stateful session EJB and commits an initial transaction, but the primary WebLogic Server fails before the EJB's initial state can be replicated. In this scenario the client's next method invocation will fail to locate the bean instance, because the initial state could not be replicated. The client would need to recreate the EJB instance using the clustered `EJBHome` stub and restart the transaction.
- Both the primary and secondary servers fail. In this scenario the client would need to recreate the EJB instance and restart the transaction.

Requirements and Configuration

To replicate the state of a stateful session EJB in a WebLogic Server cluster, ensure that the cluster is homogeneous for the EJB class. In other words, deploy the same EJB class to every WebLogic Server instance in the cluster, using the same deployment descriptors. In-memory replication is not supported for heterogeneous clusters.

By default, WebLogic Server does not replicate the state of stateful session EJB instances in a cluster. To enable replication, set the replication type deployment parameter to `InMemory` in the `weblogic-ejb-jar.xml` deployment file. For example:

```
<stateful-session-clustering>
...
...
...
<replication-type>InMemory</replication-type>
```

```
</stateful-session-clustering>
```

EJB Clustering in Oracle9*i*AS

EJB clustering in Oracle9*i*AS provides EJB load balancing and failover. For 9.0.2, the mechanisms used to achieve these are different from HTTP session load balancing and failover. For EJBs, load balancing redirection is performed by the EJB client stubs and state replication for failover is done without using cluster islands (a future release of Oracle9*i*AS will implement cluster islands for EJBs).

To create an EJB cluster, you need to specify which OC4J nodes are part of the cluster and configure each of them with the same multicast address, username, and password. The EJBs to be clustered can then be deployed to each of these nodes. Configuring all nodes in the cluster with the same multicast username and password allows authentication to all nodes with a single username / password combination. If you use a different username / password combination with the same multicast address, another cluster is actually defined.

Load Balancing

Load balancing for EJBs is performed at the EJB client end. The client stubs obtain the addresses of nodes in the cluster in one of two ways: static discovery or dynamic discovery. Once all nodes in the same cluster are known, the client stubs select one at random. Hence, load balancing is performed using a random methodology.

Static and dynamic discovery is performed as follows:

Static Discovery At lookup time, the JNDI addresses of all nodes in the cluster are provided in the lookup URL property. This requires knowledge of the node name and `ormi` port for each node. For example:

```
java.naming.provider.url = ormi://serverA:23791/ejb, ormi://serverB:23792/ejb,
                           ormi://serverC:23791/ejb;
```

Dynamic Discovery For dynamic discovery, at the first lookup made, the first node that is contacted communicates with the other nodes with the same multicast address and username / password. The `ormi` addresses of these nodes are retrieved and returned to the client stubs, which select one of the addresses at random. To enable dynamic discovery, "`lookup:`" is inserted before the `ormi` URL:

```
ic.lookup("lookup:ormi://serverA:23791/ejb");
```

Failover

Depending on the type of EJB that is clustered, failover in an EJB cluster is achieved by request redirection and state replication.

Stateless Session EJBs Load balancing and failover for stateless session EJBs is performed by EJB client stubs by redirecting requests to randomly picked nodes after the nodes have been discovered statically or dynamically. Because of the stateless nature of the EJBs, replication of bean state is not required.

Stateful Session EJBs Load balancing for stateful session EJBs is the same as for stateless session EJBs. For failover, state replication is required, and by default, is replicated to all nodes in the cluster at the end of every method call to each EJB instance. Though reliable, this obviously incurs a significant amount of CPU overhead in all the nodes and degrades performance. Hence, two more replication modes are provided to allow replication without compromising performance significantly: JVM termination and stateful session context replication modes.

The JVM termination mode replicates the state of all stateful session EJBs to one other node when the JVM executing these EJBs terminates gracefully. The replication logic uses JDK 1.3 termination hooks (hence, JDK 1.3 or later is required). This mode is the most performant among all because replication is done only once. However, reliability is not the best as it is dependent on the JVM's ability to shutdown properly.

Stateful session context mode replicates state programatically. An OC4J-proprietary class, `com.evermind.server.ejb.statefulSessionContext`, is provided to allow you to specify the information to be replicated. By setting this information as parameters for the `setAttribute` method, this information can be replicated to all nodes in the EJB cluster. Hence, EJB providers have more control on when and what to replicate.

Entity EJBs Replication for entity EJBs allows EJB state to be stored in a database. Each time the state of an entity EJB changes, it is updated in the database. The entity EJB that changes the state notifies the other nodes in the cluster that their equivalent entity EJBs are out-of-date. If the node hosting the "up-to-date" EJB fails, the client stub redirects to another node and the out-of-date entity EJB in that node resynchronizes its state with the information in the database.

> **See Also:** *Oracle9iAS Containers for J2EE Enterprise JavaBeans Developer's Guide and Reference* for information on how to configure EJB clustering.

6

Migrating JDBC

This chapter provides the information you need to migrate database access code from WebLogic Server 6.0 to Oracle9*i*AS. It addresses the migration of JDBC drivers, data sources, and connection pooling.

This chapter contains these topics:

Introduction

Migrating applications deployed on WebLogic Server that use JDBC, specifically WebLogic JDBC drivers, to OC4J and Oracle JDBC drivers is can be straightforward, requiring little or no code changes to the applications migrated. Both application servers support the same API levels for the JDBC API - full support for version 2.0 of the specification. All applications written to the standard JDBC specifications will work correctly and require minimal migration effort. The primary effort goes into configuring and deploying the applications in the new environment. Only in cases where proprietary extensions are used will the migration effort get complex.

Differences between WebLogic and Oracle9iAS Database Access Implementations

Both WebLogic Server and OC4J are fully J2EE 1.2 compliant containers that permit the usage of all types of JDBC drivers to access several different databases. Further, the JDBC drivers from BEA as well as Oracle support the same version of the JDBC standard - version 2.0 specifications. Therefore, the differences between the two servers are minimal, often differing primarily in the area of proprietary extensions. Before analyzing any differences, an overview of JDBC Drivers is apt.

Overview of JDBC 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, Net-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.

BEA provides a variety of options for database access using the JDBC API specification. These options include WebLogic jDrivers for the Oracle, Microsoft SQL Server, and Informix database management systems (DBMS). In addition to the Type 2 WebLogic jDriver for Oracle, WebLogic provides a Type 2 driver for Oracle XA and three Type 3 drivers - RMI Driver, Pool Driver and JTS.

Similarly, Oracle9*i*AS provides a variety of options for database access, particularly the best JDBC drivers for the Oracle database, and JDBC drivers from partner Merant for accessing several other databases including DB2.

- **WebLogic jDriver for Oracle**—The WebLogic jDriver for Oracle provides connectivity to the Oracle database and requires an Oracle client installation since it is based on OCI (Oracle Call Interface API). The WebLogic jDriver for Oracle XA driver extends the WebLogic jDriver for Oracle for distributed transactions.

  The Oracle thick or JDBC OCI driver is the equivalent of WebLogic jDriver for Oracle as well as WebLogic jDriver for Oracle XA since the JDBC OCI driver provides XA functionality.

- **WebLogic Pool Driver**—The WebLogic Pool driver enables utilization of connection pools from server-side applications such as HTTP servlets or EJBs.

- **Oracle JDBC-OCI Driver**—The Oracle JDBC-OCI driver allows J2EE applications to use connection pools. This driver supports JDBC 2.0 connection pool features fully.

- **WebLogic RMI Driver**—The WebLogic RMI driver is a multitier, Type 3, Java Data Base Connectivity (JDBC) driver that runs in WebLogic Server and can be used with any two-tier JDBC driver to provide database access. Additionally, when configured in a cluster of WebLogic Servers, the WebLogic RMI driver can be used for clustered JDBC, allowing JDBC clients the benefits of load balancing and fail-over provided by WebLogic Clusters.

- **WebLogic JTS Driver**—The WebLogic JTS driver is a multitier, Type 3, JDBC driver used in distributed transactions across multiple servers with one database instance. The JTS driver is more efficient than the WebLogic jDriver for Oracle XA driver when working with only one database instance because it avoids two-phase commit.

- **Oracle Thin Driver**—The two-tier Oracle Thin Type 4 driver provides connectivity from WebLogic Server to Oracle DBMS.

  If you are already using the Oracle OCI or Oracle thin JDBC drivers from your WebLogic Server, your code will not require any changes and you can move to the section on configuring data-sources in OC4J.

Migrating Data Sources

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 explictly loading the driver manager classes into the client applications runtime, 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.

WebLogic and OC4J both support the JDBC 2.0 data source API. 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 `DataSource` object reference.

Data Source Import Statements

`DataSource` objects, along with JNDI, provide access to connection pools for database connectivity. Each data source requires a separate `DataSource` object, which may be implemented as a `DataSource` class that supports either connection pooling or distributed transactions.

To use the `DataSource` objects, import the following classes in your client code:

```
import java.sql.*;
import java.util.*;
import javax.naming.*;
```

In the case of WebLogic Server, you would use the `weblogic.jdbc.*` packages and in the case of OC4J, you would use `oracle.jdbc.*` packages.

Configuring Data Sources in the Application Server

In WebLogic, you configure data sources using the Oracle Enterprise Manager (OEM) web pages to specify the data source name, database name and JDBC URL string. You can also define multiple data sources to use a single connection pool, thereby allowing you to define both transaction and non-transaction-enabled `DataSource` objects that share the same database.

The best way to configure and define data sources is through OEM. However, in this document we will examine the underlying infrastructure and focus on direct manipulation of the configuration files. OC4J uses flat files to configure data sources for all of its deployed applications. Data sources are specified in the *<ORACLE_HOME>*`/j2ee/home/config/data-sources.xml` file. Following is an sample data source configuration for an Oracle database. Each data source specified 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@<database host name><database listener port
number>:<database SID>"
pooled-location="jdbc/OraclePoolDS"
xa-location="jdbc/xa/OracleXADS"
ejb-location="jdbc/OracleDS"
connection-driver="oracle.jdbc.driver.OracleDriver"
username="scott"
password="tiger"
url="jdbc:oracle:thin@<database host name><database listener port
number>:<database SID>"
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 Configuration Parameters in `data-sources.xml` File

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

Table 6–1 Configuration Parameters in `data-sources.xml` File

Parameter	Description
`inactivity-timeout`	Number of seconds unused connections should be cached before being closed.
`location`	JNDI path for binding this data source.
`max-connect-attempts`	Number of times to retry a failed connection. Default is 3 times.
`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`. 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 the specialized data source (used by `OrionCMTDataSource`).

Obtaining a Client Connection Using a Data Source Object

To obtain a connection from a JDBC client, you would use JNDI to look up and locate the `DataSource` object. This is illustrated in the following code fragment where you obtain a connection in WebLogic Server:

```
try
{
    java.util.Properties parms = new java.util.Properties();
    parms.setProperty(Context.INITIAL_CONTEXT_FACTORY,
    "weblogic.jndi.WLInitialContextFactory");

    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 the above code from WebLogic Server to OC4J, you need to change the class that implements the initial context factory (`Context.INITIAL_CONTEXT_FACTORY`) of the JNDI tree from `weblogic.jndi.WLInitialContextFactory`, which is the WebLogic-specific class, to `com.evermind.server.ApplicationClientInitialContextFactory`, which is the OC4J specific class.

With this change, your code is ready for deployment on OC4J and to use the Oracle JDBC drivers.

Migrating Connection Pools

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, consume system resources to create the connection, maintain it, and then release it when it is no longer in use. The resource overhead is particularly high for web-based applications, because of the frequency and volume of web users connecting and disconnecting. Often, more resources are consumed in connecting and disconnecting than in the interactions themselves.

Connection pooling enables you to control connection resource usage by spreading the connection overhead across many user requests. A connection pool 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, which are left open and re-used by many client requests, instead of each client using resources to create its own connection and closing it after its database operation is complete. Connection pooling improves overall performance in the following ways:

- Reducing the load on the middle tier and server
- Minimizing resource usage by session create and session 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:

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, 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.

Overview of Connection Pools

Connection pools provide ready-to-use pools of connections to your DBMS. Since these database connections are already established when the connection pool starts up, the overhead of establishing database connections is eliminated. You can utilize connection pools from server-side applications such as HTTP servlets or EJBs using the pool driver or from stand-alone Java client applications.

One of the greatest advantages of connection pooling is that it saves valuable program execution time and has almost no or very low overhead. Making a DMBS

connection is very slow. With connection pools, connections are established and available to users before they are needed. The alternative is for application code to make its own JDBC connections when needed. A DBMS runs faster with dedicated connections than if it has to handle incoming connection attempts at runtime.

How Connection Pools Enhance Performance

Establishing a JDBC connection with a DBMS can be very slow. If your application requires database connections that are repeatedly opened and closed, this can become a significant performance issue. WebLogic and Oracle9*i*AS connection pools offer an solution to this problem.

When WebLogic Server or Oracle9*i*AS starts, connections from the connection pools are opened and are available to all clients. When a client closes a connection from a connection pool, the connection is returned to the pool and becomes available for other clients; the connection itself is not closed. There is little cost to "open" and "close" pool connections.

How many connections should you create in the pool? A connection pool can grow and shrink according to configured parameters, between a minimum and a maximum number of connections. The best performance will always be when the connection pool has as many connections as there are concurrent users.

Overview of Clustered JDBC

Relevant only in multitier configurations, clustered JDBC allows external JDBC clients to reconnect and restart their JDBC connection without changing the connection parameters, in case a serving cluster member fails. For WebLogic, clustered JDBC requires data source objects and the WebLogic RMI driver to connect to the DBMS. Data source objects are defined for each WebLogic Server using the WebLogic Administration Console.

Oracle provides functionality that is similar to and more advanced than that provided by the clustered JDBC by leveraging the TAF capabilities of OCI.

Performance Tuning JDBC

Performance tuning your JDBC application in OC4J is similar to that for WebLogic Server. Connection pooling helps improve performance by avoiding the expensive operation of creating new database connections. The guidelines on writing efficient code hold true for Oracle9*i*AS and WebLogic Server.

A

Oracle9*i*AS 1.0.2.x and WebLogic Server 6.0 Comparison

This appendix provides a comparison of performance and features between Oracle9*i*AS version 1.0.2.2.x and BEA Systems' WebLogic Server 6.0. (Oracle9*i*AS 1.0.2.2 is the predecessor Oracle9*i*AS 9.0.2.)

This chapter is organized as follows:

> **Note:** Unless explicitly stated, any mention of "Oracle9*i*AS" in this chapter refers to Oracle9*i*AS 1.0.2.x.

Introduction

In response to Oracle's delivery of a 100% J2EE 1.2 standards compliant J2EE container in Oracle9*i*AS 1.0.2.2, which is both feature-rich and highly optimized for Java execution, the primary competitors in the application server market have responded with several claims to both feature and performance differentiation to Oracle9*i*AS.

To provide Oracle's customers with a fair and accurate representation of the actual facts when comparing Oracle9*i*AS 1.0.2.2 against the most recent release of WebLogic Server 6.0, this chapter provides the specific details comparing Oracle9*i*AS' J2EE facilities with this product.

This chapter is structured in two parts:

- **Performance Results**—This chapter first explains in specific detail: (i) the objective performance comparison's that Oracle conducted against WebLogic Server 6.0; and also (ii) explains the specific reasons for the comparative performance differences between the two products.
- **Feature/Function Comparison—**This chapter also explains in specific detail the relative capabilities of Oracle9*i*AS against WebLogic Server.

The objectives of this chapter are to provide Oracle's customers and potential application server customers with a clear and objective analysis of the capabilities of Oracle9*i*AS' J2EE facilities.

Performance Results and Analysis

In order to compare the relative performance of Oracle9*i*AS 1.0.2.2's J2EE container vs WebLogic Server 6.0, Oracle ran a formalized benchmark test comparing various aspects of the performance of the Oracle9*i*AS J2EE container against WebLogic Server 6.0 (the production version available at the time of our benchmarking effort). The goals were to measure the comparative performance of the J2EE containers of the two application servers along the following objective set of guidelines:

- **Real World Benchmark**—First, the goal of our benchmarking exercise was to run a J2EE application that represented a real-world application and would execute all aspects of the J2EE container rather than one that either did not represent a real-world application or one that only tested aspects of the J2EE container that were favorable to Oracle.
- **Identical Hardware Configurations**—Second, the benchmarking exercise was conducted on identical hardware configurations, for example, the same operating systems and hardware platforms, to ensure that the results represented an objective comparison of the two products.
- **Identical Software Configurations**—Third, to ensure fairness, the benchmarking exercise was also conducted on identical software configurations. There were four aspects to such identical software configurations:
  - Web Caching—Web caching was not used to enhance the performance of Oracle9*i*AS results. While the web cache is a specific capability of Oracle9*i*AS, and when used by customers could provide further performance acceleration compared to BEA, the initial focus of the performance tests were to provide specific comparisons on the J2EE containers in these products alone. As a result, to ensure a fair comparison of the results, no web caching was used with Oracle9*i*AS.

  - JDK, JRE, and Operating System Versions—The same JDK and JRE versions and the same operating system were used in both cases. Since the more recent JDK versions have much faster JIT compilers and other optimizations of the Java runtime, these in turn would impact overall J2EE performance measurements. Again, in the interest of a fair comparison of the results, the same versions were used to measure the performance of both containers.
  - Web Server Configurations—Finally, the same web server configuration was used in both cases to ensure that measured performance differences reflected J2EE container performance alone and also reflected real-world customer deployments. Specifically, Oracle9*i*AS provides, as part of its J2EE container, an extremely fast and optimized web server which has a number of optimizations for extremely efficient HTTP streaming and input/output. Further, this web server can be run in process with the servlet engine and EJB container and, as a result, for benchmarking purposes, could provide a big performance difference when compared to BEA which embeds a much slower web server. Since the performance tests were focused on measuring J2EE container performance, we did not use the Oracle9*i*AS embedded web server but chose to use Apache, forwarding requests via a proxy architecture to both J2EE containers. This not only isolates the differences in J2EE container performance but also reflects typical customer situations where customers already have existing web servers deployed on a separate tier from their J2EE containers.

- **Performance and Scalability Comparisons**—Fourth, the benchmarking exercise was also designed to measure both performance and scalability of the vendors' J2EE containers.
  - Performance results were measured on the basis of average response time under specific user or transaction loads.
  - Scalability was measured on the basis of total CPU utilization to support a specific user or transaction load on identical hardware. The system that uses the smallest amount of CPU to support a specific number of users can support more users on the same hardware configuration and, as a result, scales best.

The results of the performance and scalability comparisons are as documented below.

Performance and Scalability Results

Although the characteristics of the individual tests varied, Oracle9*i*AS demonstrated superior performance to WebLogic Server in all tests. We stopped the

tests at the point where the application server's average response time was increasing exponentially, but all indications suggest that Oracle9*i*AS can be scaled to much higher user loads and configurations. On average, Oracle9*i*AS performance (based on response times) was 2-5 times better than WebLogic Server. On average, Oracle9*i*AS demonstrated much superior scalability (based on CPU utilization) compared to WebLogic Server. CPU utilization was about 2 times less than WebLogic Server. Even at lower user loads, Oracle9*i*AS used significantly less hardware resources than WebLogic Server.

Feature Comparison

The section above clearly documents the relative performance and scalability differences of Oracle9*i*AS' J2EE container with WebLogic Server. The results showed that the Oracle9*i*AS' J2EE container yielded better performance and used less CPU and memory than WebLogic Server on all the tests. In response to these benchmark results, BEA falsely claims that Oracle9*i*AS' J2EE container lacks several features. To address these concerns, this section specifically examines the J2EE features in Oracle9*i*AS against WebLogic Server.

Installation and Configuration

BEA has claimed that Oracle9*i*AS has a very heavyweight disk and memory footprint and is, as a result, difficult to install and configure. In doing so, they falsely compare the relative disk and memory requirements for their J2EE containers with the requirements for all of Oracle9*i*AS, which includes a directory server, an enterprise portal, a cache, business intelligence, and several other capabilities. Their comparisons are not accurate. Accurate comparisons are shown in Table 6–2.

Table 6–2 Disk and Minimum Memory Configurations for J2EE Containers

Issue	Oracle	BEA
Download Size	10 MB	32 MB
Disk Space	15 MB	45 MB
Minimum Memory	20 MB	256 MB

To address these claims, we compare the disk and memory requirements for Oracle9*i*AS J2EE container with those of WebLogic Server - the requirements for WebLogic Server are taken from BEA's product documentation. The facts highlight these important results:

- **1/3 Disk Footprint**—Oracle9*i*AS has the smallest download footprint and requires just 1/3 the disk space of WebLogic Server 6.0.
- **1/12 Minimum Memory Footprint**—The memory footprint that was measured was the recommended memory configuration required to start-up the JDK, start-up an instance of the J2EE container, and run the standard Java Pet Store Demo application. Table 6–2 indicates that Oracle9*i*AS requires less than 1/12 the memory footprint of WebLogic Server 6.0.
- **No Oracle Database Requirement**—Additionally, BEA claims that it is necessary to install and configure an Oracle database in order to run J2EE applications. This is also false. Specifically, Oracle9*i*AS' J2EE container can be installed and configured with no Oracle database dependency. All the configuration information is captured in simple XML files and the JNDI namespace is captured in the file system.
- **No Oracle Java VM Requirement**—Additionally, Oracle9*i*AS' J2EE container has no dependency on Oracle's Java Virtual Machine. It has been certified on both JDK 1.2 and JDK 1.3, versions available on all six major operating systems including Windows, Linux, Solaris, HP-UX, AIX, and Compaq TRU-64.
- **20 Minutes Installation and Configuration Time**—To specifically address these competitive considerations and to make it easy for J2EE application developers, Oracle provides a downloadable version of its web cache, Apache, and J2EE container as a single zip file. The entire product can be downloaded as a single zip file, requires one command to unzip, and a single command to start the J2EE container. The total time to install and configure all three components is 20 minutes. This is substantially faster and easier than WebLogic Server 6.0.

Performance and Scalability

Having considered the relative ease of installation and configuration, let us examine the fundamental reasons for the performance and scalability differences between Oracle9*i*AS and WebLogic Server. There are a number of reasons that drive the measured performance differences that were documented in the previous section.

- **More Efficient Code Path and Optimizations**—There are three points to consider:
  - 2-3X More Optimized Code Path—First, Oracle9*i*AS has a highly optimized J2EE container which has a roughly 2-3X more efficient code path with a much smaller instruction count than WebLogic Server 6.0.
  - Optimizations for In-Process Deployments—Additionally, Oracle9*i*AS' J2EE container has even better optimization for the typical deployment

configurations when JSPs, servlets, and EJBs are co-located in the same Java Virtual Machine. For instance, unlike WebLogic Server, Oracle9*i*AS uses very optimized in-process calling mechanisms which avoid doing costly RMI calls when calling between co-located JSPs, servlets, and EJBs.

  - Very Efficient Byte Array Operations—Finally, Java developers know that Java String operations are extremely expensive and slow to execute on even the latest versions of the JDK Virtual Machine. To avoid using such costly operations, Oracle9*i*AS' J2EE container was designed from the ground-up to use byte-array operations without using Java Strings. This is a significant performance advantage which would require significant re-architecture by BEA to duplicate.

- **More Efficient Network Protocol**—Additionally, Oracle9*i*AS uses a much more efficient network protocol than WebLogic Server to call between Java applications.

  - Very Efficient RMI Dispatch—Oracle9*i*AS' J2EE container uses a very highly optimized RMI dispatcher that has been optimized for calling between J2EE applications. Oracle9*i*AS does not use a CORBA infrastructure to route calls between J2EE applications which introduces protocol overhead when tunneling RMI through IIOP for instance. Oracle will continue to offer a highly optimized RMI protocol for performance reasons even when we comply with EJB 2.0 / J2EE 1.3 which requires RMI-over-IIOP.

- **More Efficient Database Access**—Oracle9*i*AS also uses a very efficient mechanism to schedule transactions between the middle tier application server and the database. For instance, Oracle9*i*AS can prepare and cache both a single SQL statement and multiple SQL statements across a JDBC connection pool. This ensures that Oracle9*i*AS is highly optimized when it dispatches transactions to both Oracle and non-Oracle databases and significantly improves database access performance.

- **More Efficient Container Managed Persistence**—There are two specific facts that reflect the significant performance advantages in using Oracle9*i*AS' Container Managed Persistence (CMP) implementation compared to WebLogic Server's implementation:

  - Auto-Detection of Modified EJBs—When using CMP, Oracle9*i*AS' J2EE container can automatically detect whether you have modified an EJB and writes the EJB's state to the database, for example, does an `ejbStore` only when necessary. WebLogic Server does not provide such auto-detection requiring a user to code is-modified methods which the WebLogic Server container uses to know whether or not to do the `ejbStore` operation. Note

that this problem may be addressed with the coming WebLogic Server EJB 2.0 implementation.

- Simple and Complex DB mapping for CMP—When using CMP, Oracle9*i*AS' J2EE container supports both simple (1:1, 1:many) and complex (many:many) database field mappings very efficiently. In contrast, WebLogic Server provides very poor support for even some simple CMP database field mapping (1:many) in its CMP EJB 1.1 xml files. For instance, it is difficult to qualify a `where` clause string in WebLogic Server and this results in doing unnecessary full table scans.

- **Fast and Predictable Steady State Performance**—Further, when comparing the performance of J2EE applications, the time to execute the first request / transaction with WebLogic Server when several JSPs, servlets, or EJBs are created, takes several minutes because the container takes a long time to load the first J2EE objects and does so in an as necessary fashion. Oracle9*i*AS does not have a similar problem. Note that in the interests of fairness in carrying out the performance comparisons, we increased the test ramp-up time to get WebLogic Server's performance to predictable steady-state.

- **Simpler to Configure for Scalability**—Finally, to configure Oracle9*i*AS for scalability requires very minimal tuning when compared to WebLogic Server. Oracle9*i*AS does not require the developer to configure operating system, Java Virtual Machine, network, or thread parameters for scalability. In contrast, WebLogic Server, for instance, has a "number of threads" parameter which is difficult to tune, and can limit scalability. The recommended default is 15 threads for a WebLogic Server. This parameter limits the number of concurrent users. You can tune the parameter to increase it, but it will have different effects on performance at different user loads. For example, for one of our test workloads, increasing this to 30 significantly improved performance for the 1000 user test, but made the 500 user test worse. Since in a real production environment you can't really adjust this thread count as the load changes, it is difficult to use this parameter for tuning. If left at 15, as the BEA documentation generally suggests, it can limit concurrency and scalability to fewer than 100 concurrent users. Note that Apache JServ has a "number of threads" parameter, but setting it higher than the required concurrency didn't cause the same type of degradation we saw in WebLogic Server.

J2EE Container Features

Table 6–2 lists the J2EE features of a J2EE container and the implementation currently supported by Oracle and BEA.

Table 6–3 J2EE Features of J2EE Containers

J2EE Compliance	Oracle	BEA
EJB	1.1	1.1 (early 2.0)
Servlets	2.2	2.2 (early 2.3)
JSP	1.1	1.1 (early 1.2)
JDBC	2.0	2.0
JNDI	1.2	1.2
JMS	1.0.2	1.0.2

- **JSP Facilities**—Oracle9*i*AS has a complete JSP 1.1 implementation and most of the features of JSP 1.2 including:
  - Uses Servlet 2.3 as its runtime environment
  - XML syntax for JSP pages
  - Translate time validation
  - Tag library runtime support
- **Servlet Facilities**—Oracle9*i*AS has a complete Servlet 2.2 implementation and virtually all of the features of Servlet 2.3 including:
  - Servlet filters
  - Servlet chaining
  - Application life cycle events
  - Formalized support for inter-JAR dependencies
  - Support for new class loading rules

  WebLogic Server 6.0 does not yet have JSP 1.2 and Servlet 2.3 implementation.
- **EJB Container Facilities**—Oracle9*i*AS has a complete EJB 1.1 implementation including session beans (both stateful and stateless) and entity beans (supporting both container managed persistence and bean managed persistence). Further, Oracle9*i*AS already has several of the features of EJB 2.0 including:

  - XML deployment descriptors
  - Message-driven beans
  - EJB 2.0 object-relational mapping
  - Support for several of the elements of the new CMP architecture

- **Simpler Deployment for OC4J**—As a further differentiator, application development with Oracle9*i*AS is much simpler than with WebLogic Server. For example, EJB deployment with WebLogic Server requires manual coding of the WebLogic specific EJB xml deployment files. It is done manually from scratch using a crude text editor. Oracle9*i*AS in contrast auto-generates the Oracle9*i*AS-specific EJB xml deployment files for EJBs with some defaults already filled-in. It is then much quicker to only modify the fields you need to, instead of creating the whole thing yourself.
- **JDBC Support for Oracle and non-Oracle Databases**—Oracle9*i*AS is also certified to support Oracle 7.3, 8.0, Oracle8*i*, and Oracle9*i* databases via JDBC drivers.
  - Oracle9*i*AS is the only application server today certified with the Oracle9*i* database and Real Application Clusters.
  - Oracle9*i*AS is also certified using Merant's Type 4 pure Java JDBC drivers to work against IBM DB/2, Microsoft SQL-Server, Informix, and Sybase databases.

  Note that this level of support is exactly equivalent to BEA's support for these databases.
- **Web Server Support for Netscape, IIS, and Apache Web Servers**—Oracle9*i*AS also supports the Apache web server in the box and provides proxy support for Netscape and Microsoft's IIS web server. BEA provides its own web server in the box and provides proxy support for Netscape, Apache, and Microsoft.

Clustering Support

Application server clustering (not to be confused with database clustering) essentially means the use of a group of application servers that coordinate their actions in order to provide scalable, highly-available services in a transparent manner. There are three requirements here:

- **Heterogeneous Clusters**—Unlike the database, where every node of a cluster has an identical configuration, in the middle tier, it is important to allow users to configure a set of boxes which may have heterogeneous operating systems,

hardware, and other systems infrastructure to belong to a cluster. Further, application server clusters cannot use a shared disk architecture for clustering due to the large number of middle tier instances and to the fact that web masters typically use local disks attached to application server boxes rather than more expensive shared disk arrays. Oracle9*i*AS and WebLogic Server support heterogeneous clusters using an IP-multicast based architecture for scalability.

- **Stateless Clustering**—Clustering different application server instances together to service stateless requests is straightforward. Since the requests are stateless, an external load balancer can simply direct a request to a new Oracle9*i*AS instance either on the same node or on a different node.
- **Stateful Clustering**—The far more difficult problem is to address how you cluster systems together when the requests are stateful. For instance, an e-commerce servlet is making use of the `HTTPSession` object to save the state of a shopping cart between method requests. When the client adds another item to the shopping cart, the servlet servicing the original request may not be accessible either because the load on that instance is too high or because the instance is not available (for example, the instance has failed). In this case, the request needs to be redirected to a different J2EE container to which the `HTTPSession` object's state has been replicated from the first container.

From a comparative point of view, Oracle9*i*AS' J2EE container provides the following facilities:

- **Servlet Clustering**—Oracle9*i*AS provides facilities to cluster servlets without requiring any changes to the user's application. The changes are deployment configuration modifications which are transparent to the J2EE application.
- **Clustering Architecture and Simplicity**: An important differentiator for Oracle9*i*AS' J2EE container is the ease with which different instances can be clustered and the robustness of the architecture used for clustering.
  - Clustering Simplicity—Specifically, Oracle9*i*AS requires a user to edit a single XML file to configure various Oracle9*i*AS instances to belong to a single cluster / island whether they are multiple servers with load balancing on a single machine or multiple servers with load balancing on different machines. In contrast, it is much more complex to configure WebLogic Server clusters with load balancing either with multiple instances on one machine or on multiple machines.
  - Superior Clustering Architecture—Oracle9*i*AS' J2EE container uses dynamic IP addresses to register instances as belonging to a cluster. Its built-in load balancer or any standard load balancer, such as Cisco Local

Director or BigIP, has the ability to use a variety of load balancing algorithms to route requests to different instances. In contrast, WebLogic Server uses static IP addresses to configure clustering. Static addresses preclude the use of a load balancer to distribute requests across instances. As a result, with WebLogic Server 6.0 you get either clustering or load balancing but not both.

- **Stateless Session Bean Clustering**—Oracle9*i*AS supports clustering of stateless session beans.
- **Stateful Session Bean and Entity Bean Clustering**—Oracle9*i*AS 1.0.2.2 does not provide clustering of stateful session beans and entity beans. We will be bringing such functionality to market with an upcoming release. There are two important considerations that we are working on:
  - Clustered Performance—Existing clustering facilities such as those in WebLogic Server 6.0 impose a severe performance penalty when running the instances in a stateful fashion with clustering. As a result, most users choose to keep their middle tier completely stateless and write their state to a persistent store, for example, a database. In delivering clustered EJBs, Oracle is working on optimizing the EJB clustering implementation to avoid introducing performance penalties.
  - Programmatic Simplicity—Additionally, unlike servlets which have a natural session boundary at which to failover their state, EJBs do not have such a clear boundary. As a result, we are working on simple programmatic facilities to allow users to use EJB clustering without any changes to their applications.
- **Transparent Application Failover with J2EE Applications**—Finally, when Oracle9*i*AS' J2EE container writes state to the Oracle database, Oracle9*i*AS provides facilities to fail over JDBC connections to provide transparent application failover (TAF) across nodes of the Oracle database. With EJBs, this allows a user to maintain a stateless middle tier by writing the state to the database and full state recoverability should a database node have a failure. WebLogic Server 6.0 does not provide this capability. Only BEA Tuxedo provides this capability, but it is not certified with Oracle9*i*.

Sample Migration Case Study

This section describes in detail the steps used to migrate an application from WebLogic Server 6.0 to Oracle9*i*AS 1.0.2.2.1.

Migration was done on a Windows 2000 system with OC4J connected to an Oracle 8.1.7 database and JDK 1.3.1 as runtime environment for OC4J.

1. Classpath and Libraries. As several commonly used classes, including apache SOAP 2.2 had to be accessible we created the following startup script to set the appropriate environment.

```
>d:
>cd \oracle\oc4jv1\j2ee\home
>set java_home=d:\jdk1.3
>set path=d:\jdk1.3\bin;%path%
>set CLASSPATH=D:\oracle\oc4jv1\j2ee\home\orion.jar;
D:\OrderServerNew\OrderServer.jar;D:\OrderServerNew\DataItemCore.jar;
D:\OrderServerInstall\OrderServerJars\jdom.jar;
D:\OrderServerInstall\OrderServerJars\activation.jar;
D:\OrderServerInstall\OrderServerJars\imap.jar;
D:\OrderServerInstall\OrderServerJars\mail.jar;
D:\OrderServerInstall\OrderServerJars\mailapi.jar;
D:\OrderServerInstall\OrderServerJars\pop3.jar;
D:\OrderServerInstall\OrderServerJars\smtp.jar;
D:\OrderServerInstall\OrderServerJars\soap.jar;
D:\OrderServerNew\Performance.jar

>java -hotspot -Xss128k -ms128m -mx128m -cp %CLASSPATH%
-Dplot.home=D:\OrderServerNew\OC4JRepository
-Dplot.orderserver.url=ormi://localhost:23791
-Dplot.orderserver.factory=com.evermind.server.rmi.RMIInitialContextFactory
-Dplot.orderserver.principal=admin -Dplot.orderserver.credential=admin
-Dplot.orderserver.EJBAppName=OrderServer
com.evermind.server.ApplicationServer -verbosity 10
```

   Doing benchmarks we discovered that there are several problems when including another XML parser (`xerces.jar`) in the classpath. Development also recommends to start OC4J with `java –jar –orion.jar` (so the manifest file in the `orion.jar` is used). It can slow down OC4J to 50% of it's normal performance. So, if the application, e.g. Apache SOAP, needs an XML parser, then include it in the `WEB-INF\lib` directory and change the entry in the `web.xml` file for the XML-Parser as follows.

```
<init-param>
   <param-name>XMLParser</param-name>
   <param-value>org.apache.crimson.jaxp.DocumentBuilderFactoryImpl
   </param-value>
</init-param>
```

We tried the crimson parser and it worked faster than xerces.

The second thing we discovered doing benchmarks with WebLogic Server 6.0 which uses its internal JDK1.3.0 and OC4J using JDK1.3.1 was that the `java -server` option gives very bad performance for OC4J.

Using `java -hotspot` instead, WebLogic Server had 2 times better performance for looping through a large JDBC result set. OC4J had 4 times better performance with the same operation.

2. Automatic invocation of a class at server startup, WebLogic Server 6.0 has a parameter `weblogic.system.startupClass.initialise=<class-name>` that enables automatic initialization of environment constants and services at startup.

To get similar behaviour in OC4J we created an startup application as java enterprise client, put it in an .ear and set two parameters in the config files.

`server.xml`:

```
<application name="Startup" parent="OrderServer"
 path="../applications/startup.ear" auto-start="true" />
```

`application.xml` for `Startup.ear`

```
<application>
   <display-name>StartupSettings</display-name>
   <module>
      <java>Start.jar</java>
   </module>
</application>
```

Example of a startup class:

```
import my.util.StartupHelper;

public class OracleASStartupClient {
   private static void startup() {
      StartupHelper.setConstantswithProperties();
      StartupHelper.cacheSettings();
      StartupHelper.installClientInteface();
      StartupHelper.installLogging();
   }

   public static String startBatch (String name, Hashtable args) throws
      Exception {
```

```
      Object obj =  Class.forName(name).getConstructor(
         new Class[]{Class.forName("java.util.Hashtable")}).newInstance(
         new Object[]{args});

      if (obj instanceof Thread) {
         ((Thread)obj).start();
         }
      else throw  new Exception("Class to start must be of Type Thread");
      // Write your startup code here...
      return  "BatchStarter Successfully completed.";
   }

   public static void main(String[] args) {
      System.out.println("***** STARTUP begin ******************");
      startup();

/*    try {
        startBatch("my.orderserver.batch.MessagePersisterBatch",
           new Hashtable());
     catch(Exception e) {
     e.printStackTrace();*/

      System.out.println("******* STARTUP end ********************");
      return;
   }
}
```

3. Adapt application from WebLogic Server global JNDI context to OC4J specific context. The JNDI tree in WebLogic Server enables each enterprise application to see all other ear's methods. In OC4J, it is standards compliant, and therefore, we had to use the `parent-application` parameter in `server.xml` (see 2 above). The following class was used to establish the JNDI context:

```
public static Context getInitialContext (boolean security) throws
Exception {
   Hashtable env= new Hashtable();
   env.put(Context.INITIAL_CONTEXT_FACTORY,Constants.OS_FACTORY);
   env.put(Context.PROVIDER_URL, Constants.ORDERSERVER_URL);
   if(security) {
      env.put(Context.SECURITY_PRINCIPAL,
              Constants.ORDERSERVER_PRINCIPAL);
      env.put(Context.SECURITY_CREDENTIALS,
              Constants.ORDERSERVER_CREDENTIAL);
```

```
        try {
           Context ctx = new InitialContext(env);
           return ctx;
        }
        catch (Exception e) {
           e.printStackTrace();
        }
        return  new InitialContext();
    }
```

4. Enable call-by-reference for EJB methods. As the application was programmed to make use of call-by-reference for EJB method calls, we had to set this behaviour (default is call-by-value) in `orion-ejb-jar.xml`.

   `orion-ejb-jar.xml` for `OrderServerGroup`:

```
<?xml version="1.0"?>
<!DOCTYPE orion-ejb-jar PUBLIC "-//Evermind//DTD Enterprise JavaBeans 1.1
runtime//EN" "http://xmlns.oracle.com/ias/dtds/orion-ejb-jar.dtd">

<orion-ejb-jar deployment-version="1.0.2.2.1" deployment-time="eadf228126">
   <enterprise-beans>
      <session-deployment name="AblaufsteuerungBean"
      location="AblaufsteuerungBean" wrapper=
      "AblaufsteuerungHome_StatelessSessionHomeWrapper1" timeout="0"
      persistence-filename="AblaufsteuerungBean" />

      <session-deployment name="OrderService" copy-by-value=false
      location="OrderService" wrapper=
      "OrderServiceHome_StatelessSessionHomeWrapper3" timeout="0"
      persistence-filename="OrderService" />

      <session-deployment name="TransparentService"
      location="TransparentService" wrapper=
      "TransparentServiceHome_StatelessSessionHomeWrapper5" timeout="0"
      persistence-filename="TransparentService" />
   </enterprise-beans>

   <assembly-descriptor>
      <default-method-access>
         <security-role-mapping name="&lt;default-ejb-caller-role&gt;"
         impliesAll="true" />
      </default-method-access>
   </assembly-descriptor>
```

```
</orion-ejb-jar>
```

5. Enable connection pooling with OC4J. We used the same Oracle JDBC 2.0 drivers `classes12.jar` for WebLogic Server and Oracle9*i*AS. To enable connection pooling, we had to switch off MTS on Oracle 8.1.7. We used the `ejb-location` in the following `data-sources.xml` config file:

```
<data-source
class="com.evermind.sql.DriverManagerDataSource"
name="OrderServerDS"
location="myOrderServerDS"
pooled-location="pooledOrderServerDS"
xa-location="jdbc/xa/OrderServerXADS"
ejb-location="OrderServerDS"
connection-driver="oracle.jdbc.driver.OracleDriver"
username="plot"
password="plot"
url="jdbc:oracle:thin:@localhost:1529:o8i"
inactivity-timeout="3600"
min-connections="20"
max-connections="100"
/>
```

6. Transaction timeout for EJB's. During benchmarking, we discovered in some tests that the EJB container ran into a time-out and rolled back transactions. To fix that, we set the `timeout` parameter in `orion-ejb-jar.xml` to zero (no time-outs).

```
<enterprise-beans>
   <session-deployment name="TransparentService"
   location="TransparentService"
   wrapper="TransparentServiceHome_StatelessSessionHomeWrapper5" timeout="0"
   persistence-filename="TransparentService"
   />
</enterprise-beans>
```

B

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 Weblogic Server 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

P

R

S

T

U

W