

Oracle® Application Server Containers for J2EE

Services Guide

10g Release 2 (10.1.2) for Windows or UNIX

Part No. B14012-02

July 2005

Oracle Application Server Containers for J2EE Services Guide, 10g Release 2 (10.1.2) for Windows or UNIX
B14012-02

Copyright © 2002, 2005, Oracle. All rights reserved.

Primary Author: Alfred Franci

Contributing Author: Janis Greenberg, Mark Kennedy, Peter Purich, Elizabeth Hanes Perry, Sheryl Maring.

Contributor: Anirruddha Thakur, Anthony Lai, Ashok Banerjee, Brian Wright, Cheuk Chau, Debabrata Panda, Editor Ellen Siegal, Erik Bergenholtz, Gary Gilchrist, Irene Zhang, J.J. Snyder, Jon Currey, Jyotsna Laxminarayanan, Krishna Kunchithapadam, Kuassi Mensah, Lars Ewe, Lelia Yin, Mike Lehmann, Mike Sanko, Min-Hank Ho, Nickolas Kavantzias, Rachel Chan, Rajkumar Irudayaraj, Raymond Ng, Sastry Malladi, Stella Li, Sunil Kunisetty, Thomas Van Raalte.

The Programs (which include both the software and documentation) contain proprietary information; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. This document is not warranted to be error-free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose.

If the Programs are delivered to the United States Government or anyone licensing or using the Programs on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth in FAR 52.227-19, Commercial Computer Software—Restricted Rights (June 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and we disclaim liability for any damages caused by such use of the Programs.

Oracle, JD Edwards, PeopleSoft, and Retek are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

The Programs may provide links to Web sites and access to content, products, and services from third parties. Oracle is not responsible for the availability of, or any content provided on, third-party Web sites. You bear all risks associated with the use of such content. If you choose to purchase any products or services from a third party, the relationship is directly between you and the third party. Oracle is not responsible for: (a) the quality of third-party products or services; or (b) fulfilling any of the terms of the agreement with the third party, including delivery of products or services and warranty obligations related to purchased products or services. Oracle is not responsible for any loss or damage of any sort that you may incur from dealing with any third party.

Contents

Send Us Your Comments	xiii
Preface	xv
Intended Audience.....	xv
Documentation Accessibility	xv
Structure	xvi
Related Documents	xvii
Conventions	xviii
1 Introduction to OC4J Services	
Java Naming and Directory Interface (JNDI)	1-1
Java Message Service (JMS)	1-1
Remote Method Invocation (RMI).....	1-2
Data Sources	1-2
Java Transaction API (JTA)	1-2
J2EE Connector Architecture (J2CA).....	1-2
Java Object Cache.....	1-2
2 Java Naming and Directory Interface	
Introduction	2-1
Initial Context	2-1
Example.....	2-2
Constructing a JNDI Context	2-2
The JNDI Environment	2-3
Creating the Initial Context in OC4J	2-4
From J2EE Application Clients.....	2-4
Environment Properties	2-5
Accessing Objects from an Application Client	2-6
From J2EE Application Components	2-8
Objects in the Same Application.....	2-8
Objects Not in the Same Application	2-9
JNDI State Replication	2-11
Enabling JNDI State Replication	2-11
JNDI State Replication Limitations.....	2-11
Multiple Islands on a Given Subnet.....	2-11

Propagating Changes Across the Cluster	2-11
Binding a Remote Object.....	2-12
JNDI Lookups in a Multiple-Instance Environment	2-12

3

Java Message Service (JMS)

Overview	3-1
Oracle Application Server JMS	3-2
Configuring OracleAS JMS Ports.....	3-2
Configuring OracleAS JMS Destination Objects	3-2
Default Destination Objects.....	3-4
Default Connection Factories	3-4
Steps for Sending a Message	3-5
Steps for Receiving a Message	3-6
OracleAS JMS Utilities.....	3-8
OracleAS JMS File-Based Persistence.....	3-10
Overview	3-10
Enabling Persistence	3-11
Recovery	3-11
Abnormal Termination.....	3-14
Predefined OracleAS JMS Exception Queue	3-14
Message Expiration.....	3-15
Message Paging	3-15
OracleAS JMS Configuration File Elements for jms.xml	3-16
Examples	3-20
OracleAS JMS System Properties.....	3-21
Resource Providers	3-23
Configuring a Custom Resource Provider	3-23
Oracle JMS	3-24
Using OJMS as a Resource Provider	3-24
Install and Configure the JMS Provider	3-25
Create User and Assign Privileges	3-25
Create JMS Destination Objects	3-25
Define the OJMS Resource Provider	3-27
Access the OJMS Resources	3-29
Using OJMS with Oracle Application Server and the Oracle Database.....	3-32
Error When Copying aqapi.jar.....	3-32
OJMS Certification Matrix	3-32
Map Logical Names in Resource References to JNDI Names	3-33
JNDI Naming for OracleAS JMS	3-34
JNDI Naming for OJMS.....	3-35
JNDI Naming Property Setup for Java Application Clients	3-35
Client Sends JMS Message Using Logical Names	3-35
Third-Party JMS Providers	3-37
Using WebSphere MQ as a Resource Provider	3-37
Configuring WebSphere MQ	3-37
Using SonicMQ as a Resource Provider	3-38

Configuring SonicMQ	3-38
Using SwiftMQ as a Resource Provider.....	3-39
Configuring SwiftMQ.....	3-39
Using Message-Driven Beans	3-39
High Availability and Clustering for JMS	3-40
OracleAS JMS High Availability Configuration.....	3-41
Terminology.....	3-41
OracleAS JMS Server Distributed Destinations.....	3-42
Cold Failover Cluster	3-43
OracleAS Dedicated JMS Server	3-44
Modifying the OPMN Configuration	3-45
Configuring OracleAS JMS.....	3-46
Queue Connection Factory Definition Example.....	3-46
Deploying Applications	3-46
High Availability	3-47
OJMS High Availability Configuration	3-47
Failover Scenarios When Using a RAC Database With OJMS	3-47
Using JMS with RAC Network Failover.....	3-47
Using OJMS With Transparent Application Failover (TAF)	3-48
Server Side Sample Code for Failover for Both JMS Providers.....	3-49
Clustering Best Practices	3-50

4 Data Sources

Introduction.....	4-1
Types of Data Sources	4-1
Emulated Data Sources	4-2
Nonemulated Data Sources	4-4
Native Data Sources	4-4
Mixing Data Sources.....	4-5
Defining Data Sources	4-6
Configuration Files	4-7
Defining Location of the Data Source XML Configuration File.....	4-7
Application-Specific Data Source XML Configuration File.....	4-8
Data Source Attributes	4-8
Defining Data Sources in Oracle Enterprise Manager 10g.....	4-10
Defining Data Sources in the XML Configuration File.....	4-11
Fatal Error Code Enhancement	4-11
Password Indirection.....	4-12
Configuring an Indirect Password with Oracle Enterprise Manager 10g	4-12
Configuring an Indirect Password Manually	4-13
Associating a Database Schema with a Data Source.....	4-14
The database-schema.xml File	4-14
Example Configuration.....	4-15
Using Data Sources	4-15
Portable Data Source Lookup.....	4-15
Retrieving a Connection from a Data Source.....	4-17
Retrieving Connections with a Nonemulated Data Source	4-18

Retrieving a Connection Outside a Global Transaction.....	4-18
Retrieving a Connection Within a Global Transaction.....	4-18
Connection Retrieval Error Conditions	4-18
Using Different User Names for Two Connections to a Single Data Source	4-19
Improperly configured OCI JDBC Driver	4-19
Using Two-Phase Commits and Data Sources	4-19
Using Oracle JDBC Extensions	4-21
Using Connection Caching Schemes	4-21
Using the OCI JDBC Drivers	4-22
Notes on Oracle JDBC-OCI driver upgrade in the Oracle Application Server	4-23
Using DataDirect JDBC Drivers	4-23
Installing and Setting Up DataDirect JDBC Drivers	4-24
Example DataDirect Data Source Entries	4-24
SQLServer	4-25
DB2	4-25
Sybase	4-25
High Availability Support for Data Sources.....	4-26
Oracle Maximum Availability Architecture (MAA).....	4-26
Oracle Data Guard	4-26
Real Application Clusters (RAC).....	4-26
Network Failover	4-26
TAF Failover	4-27
High Availability (HA) Support in OC4J	4-27
Configuring Network Failover with OC4J	4-28
Configuring Transparent Application Failover (TAF) with OC4J	4-28
Configuring a TAF Descriptor (tnsnames.ora)	4-29
Connection Pooling.....	4-30
Acknowledging TAF Exceptions	4-31
SQL Exception Handling	4-32

5 Oracle Remote Method Invocation

Introduction to RMI/ORMI.....	5-1
ORMI Enhancements.....	5-1
Increased RMI Message Throughput.....	5-1
Enhanced Threading Support	5-2
Co-Located Object Support	5-2
Client-Side Requirements	5-2
Configuring OC4J for RMI.....	5-2
Configuring RMI Using Oracle Enterprise Manager 10g.....	5-3
Configuring RMI Manually	5-4
Editing server.xml.....	5-5
Editing rmi.xml	5-5
Editing opmn.xml	5-6
RMI Configuration Files.....	5-7
JNDI Properties for RMI.....	5-7
Naming Provider URL	5-7
Context Factory Usage	5-9

Example Lookups.....	5-10
OC4J Standalone	5-10
OC4J in Oracle Application Server: Releases Before 9.0.4	5-10
OC4J in Oracle Application Server Since 9.0.4	5-11
Configuring ORMI Tunneling through HTTP	5-11
Configuring an OC4J Mount Point	5-12
Configuring ORMI over HTTP in a Clustered Application	5-13

6 J2EE Interoperability

Introduction to RMI/IIOP.....	6-1
Transport	6-1
Naming	6-2
Security	6-2
Transactions	6-2
Client-Side Requirements	6-2
The rmic.jar Compiler.....	6-3
Switching to Interoperable Transport	6-3
Simple Interoperability in a Standalone Environment.....	6-3
Advanced Interoperability in a Standalone Environment.....	6-4
Simple Interoperability in Oracle Application Server Environment.....	6-5
Configuring for Interoperability Using Oracle Enterprise Manager 10g.....	6-5
Configuring for Interoperability Manually.....	6-6
Advanced Interoperability in Oracle Application Server Environment.....	6-8
Configuring for Interoperability Using Oracle Enterprise Manager 10g.....	6-8
Configuring for Interoperability Manually.....	6-8
The corbaname URL	6-10
The OPMN URL	6-10
Exception Mapping.....	6-11
Invoking OC4J-Hosted Beans from a Non-OC4J Container	6-11
Configuring OC4J for Interoperability	6-12
Interoperability OC4J Flags	6-12
Interoperability Configuration Files.....	6-12
JNDI Properties for Interoperability (jndi.properties).....	6-13
Context Factory Usage	6-13
Enabling IIOP in OC4J	6-13

7 Java Transaction API

Introduction.....	7-1
Demarcating Transactions	7-1
Enlisting Resources	7-2
Single-Phase Commit	7-2
Enlisting a Single Resource.....	7-2
Configure the Data Source.....	7-2
Retrieve the Data Source Connection	7-3
Perform JNDI Lookup	7-3
Retrieve a Connection	7-4

Demarcating the Transaction	7-4
Container-Managed Transactional Demarcation	7-4
Bean-Managed Transactions	7-6
JTA Transactions	7-6
JDBC Transactions	7-6
Two-Phase Commit	7-6
Configuring Two-Phase Commit Engine	7-7
Database Configuration Steps	7-7
OC4J Configuration Steps	7-8
Limitations of Two-Phase Commit Engine	7-10
Configuring Timeouts	7-10
Recovery for CMP Beans when a Database Instance Fails	7-11
Connection Recovery for CMP Beans That Use Container-Managed Transactions	7-11
Connection Recovery for CMP Beans That Use Bean-Managed Transactions	7-11
Using Transactions With MDBs	7-11
Transaction Behavior for MDBs Using OC4J JMS	7-12
Transaction Behavior for MDBs Using Oracle JMS	7-12
MDBs that Use Container-Managed Transactions	7-12
MDBs that Use Bean-Managed Transactions and JMS Clients	7-13

8 J2EE Connector Architecture (J2CA)

Introduction	8-1
Resource Adapters	8-1
Standalone Resource Adapters	8-2
Embedded Resource Adapters	8-2
Example of RAR File Structure	8-2
The ra.xml Descriptor	8-2
Application Interface	8-2
Quality of Service Contracts	8-3
Deploying and Undeploying Resource Adapters	8-3
Deployment Descriptors	8-3
The oc4j-ra.xml Descriptor	8-4
The oc4j-connectors.xml Descriptor	8-6
Standalone Resource Adapters	8-7
Deployment	8-7
Example:	8-8
Embedded Resource Adapters	8-9
Deployment	8-9
Locations of Relevant Files	8-9
Specifying Quality of Service Contracts	8-10
Configuring Connection Pooling	8-10
Managing EIS Sign-On	8-11
Component-Managed Sign-On	8-12
Example:	8-12
Container-Managed Sign-On	8-13
Example:	8-14
Declarative Container-Managed Sign-On	8-14

Programmatic Container-Managed Sign-On	8-15
OC4J-Specific Authentication Classes.....	8-15
JAAS Pluggable Authentication Classes	8-18
Special Features Accessible Via Programmatic Interface.....	8-19

9 Java Object Cache

Java Object Cache Concepts	9-1
Java Object Cache Basic Architecture.....	9-2
Distributed Object Management.....	9-3
How the Java Object Cache Works.....	9-4
Cache Organization	9-5
Java Object Cache Features	9-6
Java Object Cache Object Types	9-6
Memory Objects	9-6
Disk Objects.....	9-7
StreamAccess Objects	9-7
Pool Objects.....	9-7
Java Object Cache Environment	9-8
Cache Regions.....	9-8
Cache Subregions	9-8
Cache Groups	9-8
Region and Group Size Control	9-9
Cache Object Attributes.....	9-10
Using Attributes Defined Before Object Loading	9-10
Using Attributes Defined Before or After Object Loading	9-13
Developing Applications Using Java Object Cache	9-15
Importing Java Object Cache	9-15
Defining a Cache Region.....	9-15
Defining a Cache Group.....	9-16
Defining a Cache Subregion	9-16
Defining and Using Cache Objects	9-17
Implementing a CacheLoader Object.....	9-17
Using CacheLoader Helper Methods.....	9-18
Invalidating Cache Objects	9-19
Destroying Cache Objects	9-19
Multiple Object Loading and Invalidation.....	9-20
Java Object Cache Configuration	9-21
Examples	9-24
Declarative Cache.....	9-25
Declarative Cache File Sample.....	9-27
Declarative Cache File Format	9-27
Examples	9-29
Declarable User-Defined Objects.....	9-31
Declarable CacheLoader, CacheEventListener, and CapacityPolicy	9-32
Initializing the Java Object Cache in a Non-OC4J Container	9-32
Capacity Control	9-33
Implementing a Cache Event Listener	9-35

Restrictions and Programming Pointers	9-36
Working with Disk Objects	9-38
Local and Distributed Disk Cache Objects	9-38
Local Objects	9-38
Distributed Objects	9-38
Adding Objects to the Disk Cache	9-38
Automatically Adding Objects	9-39
Explicitly Adding Objects	9-39
Using Objects that Reside Only in Disk Cache	9-39
Working with StreamAccess Objects	9-41
Creating a StreamAccess Object	9-41
Working with Pool Objects	9-42
Creating Pool Objects	9-42
Using Objects from a Pool	9-43
Implementing a Pool Object Instance Factory	9-43
Pool Object Affinity	9-44
Running in Local Mode	9-44
Running in Distributed Mode	9-45
Configuring Properties for Distributed Mode	9-45
Setting the distribute Configuration Property	9-45
Setting the discoveryAddress Configuration Property	9-45
Using Distributed Objects, Regions, Subregions, and Groups	9-46
Using the REPLY Attribute with Distributed Objects	9-46
Using SYNCHRONIZE and SYNCHRONIZE_DEFAULT	9-47
Cached Object Consistency Levels	9-49
Using Local Objects	9-49
Propagating Changes Without Waiting for a Reply	9-49
Propagating Changes and Waiting for a Reply	9-50
Serializing Changes Across Multiple Caches	9-50
Sharing Cached Objects in an OC4J Servlet	9-50
Using User-Defined Class Loaders	9-51
HTTP and Security for Distributed Cache	9-51
HTTP	9-52
SSL	9-52
Firewall	9-53
Restricting Incoming Connections	9-54
Monitoring and Debugging	9-55
XML Schema for Cache Configuration	9-58
XML schema for attribute declaration	9-60

Index

Send Us Your Comments

Oracle Application Server Containers for J2EE Services Guide, 10g Release 2 (10.1.2) for Windows or UNIX

B14012-02

Oracle welcomes your comments and suggestions on the quality and usefulness of this publication. 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 about this manual?

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

- Electronic mail: appserverdocs_us@oracle.com
- FAX: (650) 506-7225 Attn: Java Platform Group, Information Development Manager

- Postal service:

Oracle Corporation
Java Platform Group, Information Development Manager
500 Oracle Parkway, Mailstop 4op9
Redwood Shores, CA 94065
USA

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

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

Preface

Oracle Application Server 10g Release 2 (10.1.2) includes a J2EE environment known as Oracle Application Server Containers for J2EE (OC4J). This book describes the services provided by OC4J.

This preface contains these topics:

- [Intended Audience](#)
- [Documentation Accessibility](#)
- [Structure](#)
- [Related Documents](#)
- [Conventions](#)

Intended Audience

This manual is intended for developers familiar with the J2EE architecture who want to understand Oracle J2EE Services.

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. Accessibility standards will continue to evolve over time, and Oracle 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 more information, visit the Oracle Accessibility Program Web site at

<http://www.oracle.com/accessibility/>

Accessibility of Code Examples in Documentation

Screen readers 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, some screen readers 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 does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Structure

This document contains the following chapters:

Chapter 1 Introduction to OC4J Services

This chapter gives an overview of the service technologies included in OC4J.

Chapter 2 Java Naming and Directory Interface

This chapter covers using the Java Naming and Directory Interface (JNDI) to look up objects.

Chapter 3 Java Message Service (JMS)

This chapter discusses plugging Resource Providers into the Java Message Service (JMS) and the two Oracle JMS (OJMS) providers that Oracle furnishes.

Chapter 4 Data Sources

This chapter discusses data sources, vendor-independent encapsulations of a connection to a database server.

Chapter 5 Oracle Remote Method Invocation

This chapter describes OC4J support for Remote Method Invocation (RMI) over the proprietary Oracle RMI (ORMI) protocol.

Chapter 6 J2EE Interoperability

This chapter describes OC4J support for EJB2.0 interoperation using RMI over the standard Internet Inter-Orb Protocol (IIOP) protocol.

Chapter 7 Java Transaction API

This chapter documents the Oracle implementation of the Java Transaction API (JTA).

Chapter 8 J2EE Connector Architecture (J2CA)

This chapter describes how to use the J2EE Connector Architecture in an OC4J application.

Chapter 9 Java Object Cache

This chapter describes the OC4J Java Object Cache, including its architecture and programming features.

Related Documents

For more information, see the following additional OC4J documents available from the Oracle Java Platform Group:

- *Oracle Application Server Containers for J2EE Users Guide*
This book presents an overview and general information for OC4J; primer chapters for servlets, JSP pages, and EJBs; and general configuration and deployment instructions.
- *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developers Guide*
This book provides information for JSP developers who want to run their pages in OC4J. It includes a general overview of JSP standards and programming considerations, as well as discussion of Oracle value-added features and steps for getting started in the OC4J environment.
- *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*
This book discusses conceptual information and detailed syntax and usage information for tag libraries, Java Beans, and other OC4J Java utilities.
- *Oracle Application Server Containers for J2EE Servlet Developers Guide*
This book includes information for servlet developers regarding use of servlets and the servlet container in OC4J.
- *Oracle Application Server Containers for J2EE Enterprise Java Beans Developers Guide*
This book documents the EJB implementation and EJB container in OC4J.

The following documents are available from the Oracle Application Server group:

- *Oracle Application Server 10g Administrators Guide*
- *Oracle Enterprise Manager Administrators Guide*
- *Oracle HTTP Server Administrators Guide*
- *Oracle Application Server 10g Performance Guide*
- *Oracle Application Server 10g Globalization Guide*
- *Oracle Application Web Cache Administrators Guide*

The following are available from the JDeveloper group:

- Oracle JDeveloper online help
- Oracle JDeveloper documentation on the Oracle Technology Network:
<http://www.oracle.com/technology/products/jdev/index.html>

The following OTN resources are available for further information about OC4J:

- OTN Web site for OC4J:
<http://www.oracle.com/technology/tech/java/oc4j/index.html>
- OTN OC4J discussion forums, accessible through the following address:
<http://otn.oracle.com/forums/forum.jsp?id=486963>

Printed documentation is available for sale in the Oracle store at:

<http://oraclestore.oracle.com>

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

<http://otn.oracle.com/membership/>

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

<http://www.oracle.com/technology/documentation/index.html>

Conventions

The following conventions are also used in this manual:

Convention	Meaning
...	Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted
boldface text	Boldface type in text indicates a term defined in the text, the glossary, or in both locations.
<i>Italics</i>	Italic typeface indicates book titles or emphasis, or terms that are defined in the text.
Monospace (fixed-width) font	Monospace typeface within text indicates items such as executables, file names, directory names, Java class names, Java method names, variable names, other programmatic elements (such as JSP tags or attributes, or XML elements or attributes), or database SQL commands or elements (such as schema names, table names, or column names).
Italic monospace (fixed-width) font	Italic monospace font represents placeholders or variables.
< >	Angle brackets enclose user-supplied names.
[]	Brackets enclose optional clauses from which you can choose one or none.
	A vertical bar represents a choice of two or more options. Enter one of the options. Do not enter the vertical bar.

Introduction to OC4J Services

Oracle Application Server Containers for J2EE (OC4J) supports the following technologies, each of which has its own chapter in this book:

- [Java Naming and Directory Interface \(JNDI\)](#)
- [Java Message Service \(JMS\)](#)
- [Remote Method Invocation \(RMI\)](#)
- [Data Sources](#)
- [Java Transaction API \(JTA\)](#)
- [J2EE Connector Architecture \(J2CA\)](#)
- [Java Object Cache](#)

This chapter gives a brief overview of each technology in the preceding list.

Note: In addition to these technologies, OC4J supports the JavaMail API, the JavaBeans Activation Framework (JAF), and the Java API for XML Processing (JAXP). For information about these technologies, see the Sun Microsystems J2EE documentation.

Java Naming and Directory Interface (JNDI)

The Java Naming and Directory Interface (JNDI) service that is implemented by OC4J provides naming and directory functionality for Java applications. JNDI is defined independently of any specific naming or directory service implementation. As a result, JNDI enables Java applications to access different, possibly multiple, naming and directory services using a single API. Different naming and directory service provider interfaces (SPIs) can be plugged in behind this common API to handle different naming services.

See [Chapter 2, "Java Naming and Directory Interface"](#), for details.

Java Message Service (JMS)

Java Message Service (JMS) provides a common way for Java programs to access enterprise messaging products. JMS is a set of interfaces and associated semantics that define how a JMS client accesses the facilities of an enterprise messaging product.

See [Chapter 3, "Java Message Service \(JMS\)"](#), for details.

Remote Method Invocation (RMI)

Remote Method Invocation (RMI) is one Java implementation of the remote procedure call paradigm, in which distributed applications communicate by invoking procedure calls and interpreting the return values.

OC4J supports RMI over both the Oracle Remote Method Invocation (ORMI) protocol and over the Internet Inter-ORB Protocol (IIOP).

By default, OC4J uses RMI/ORMI. In addition to the benefits provided by RMI/IIOP, RMI/ORMI provides additional features such as invoking RMI/ORMI over HTTP, a technique known as "RMI tunneling."

See [Chapter 5, "Oracle Remote Method Invocation"](#), for details on RMI/ORMI.

Version 2.0 of the Enterprise Java Beans (EJB) specification uses RMI over the Internet Inter-ORB Protocol (IIOP) to make it easy for EJB-based applications to invoke one another across different containers. You can make your existing EJB interoperable without changing a line of code: simply edit the bean properties and redeploy. J2EE uses RMI to provide interoperability between EJBs running on different containers.

See [Chapter 6, "J2EE Interoperability"](#), for details on interoperability (RMI/IIOP).

Data Sources

A data source, which is the instantiation of an object that implements the `javax.sql.DataSource` interface, enables you to retrieve a connection to a database server.

See [Chapter 4, "Data Sources"](#), for details.

Java Transaction API (JTA)

EJBs use Java Transaction API (JTA) 1.0.1 for managing transactions. These transactions involve single-phase and two-phase commits.

See [Chapter 7, "Java Transaction API"](#), for details.

J2EE Connector Architecture (J2CA)

J2EE Connector Architecture (J2CA) defines a standard architecture for connecting the J2EE platform to heterogeneous Enterprise Information Systems (EISs). Examples of EISs include ERP, mainframe transaction processing, database systems, and legacy applications that are not written in the Java programming language.

See [Chapter 8, "J2EE Connector Architecture \(J2CA\)"](#), for details.

Java Object Cache

The Java Object Cache (formerly OCS4J) is a set of Java classes that manage Java objects within a process, across processes, and on a local disk. The primary goal of the Java Object Cache is to provide a powerful, flexible, easy-to-use service that significantly improves server performance by managing local copies of objects that are expensive to retrieve or create. There are no restrictions on the type of object that can be cached or the original source of the object. The management of each object in the cache is easily customized. Each object has a set of attributes associated with it to control such things as how the object is loaded into the cache, where the object is stored (in memory, on disk, or both), how it is invalidated (based on time or by explicit

request), and who should be notified when the object is invalidated. Objects can be invalidated as a group or individually. See [Chapter 9, "Java Object Cache"](#), for details.

Java Naming and Directory Interface

This chapter describes the Java Naming and Directory Interface (JNDI) service that is implemented by Oracle Application Server Containers for J2EE (OC4J) applications. It covers the following topics:

- [Introduction](#)
- [Constructing a JNDI Context](#)
- [The JNDI Environment](#)
- [Creating the Initial Context in OC4J](#)

Introduction

JNDI, part of the J2EE specification, provides naming and directory functionality for Java applications. Because JNDI is defined independently of any specific naming or directory service implementation, it enables Java applications to access different naming and directory services using a single API. Different naming and directory *service provider interfaces* (SPIs) can be plugged in behind this common API to handle different naming services.

Before reading this chapter, you should be familiar with the basics of JNDI and the JNDI API. For basic information about JNDI, including tutorials and the API documentation, visit the Sun Microsystems Web site at:

<http://java.sun.com/products/jndi/index.html>

A JAR file implementing JNDI, `jndi.jar`, is available with OC4J. Your application can take advantage of the JNDI API without having to provide any other libraries or JAR files. A J2EE-compatible application uses JNDI to obtain naming contexts that enable the application to locate and retrieve objects such as data sources, Java Message Service (JMS) services, local and remote Enterprise Java Beans (EJBs), and many other J2EE objects and services.

Note: For information about controlling access to JNDI namespaces, see the *Oracle Application Server Security Guide*.

Initial Context

The concept of the initial context is central to JNDI. Here are the two most frequently used JNDI operations in J2EE applications:

- Creating a new `InitialContext` object (in the `javax.naming` package)
- Using the `InitialContext`, to look up a J2EE or other resource

When OC4J starts up, it constructs a JNDI initial context for each application by reading resource references in the configuration XML file of each application.

Note: After the initial configuration, the JNDI tree for each application is purely memory-based. Additions made to the context at run time are not persisted. When OC4J is restarted, additional bindings made by the application components to the JNDI name space, such as making a `Context.bind` API call in application code, are no longer available. However, anything that is bound declaratively through the various XML files is reconstructed upon startup.

Example

The following example shows two lines of Java code to use on the server side in a typical Web or EJB application:

```
Context ctx = new InitialContext();
myEJBHome myhome =
    (HelloHome) ctx.lookup("java:comp/env/ejb/myEJB");
```

The first statement creates a new initial context object, using the default environment. The second statement looks up an EJB home interface reference in the JNDI tree of the application. In this case, `myEJB` might be the name of a session bean that is declared in the `web.xml` (or `orion-web.xml`) configuration file, in an `<ejb-ref>` tag. For example:

```
<ejb-ref>
  <ejb-ref-name>ejb/myEJB</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>myEjb.HelloHome</home>
  <remote>myEjb.HelloRemote</remote>
</ejb-ref>
```

This chapter focuses on setting up the initial contexts for using JNDI and describing how OC4J performs JNDI lookups. For more information about the other JNDI classes and methods, see the Javadoc at:

<http://java.sun.com/products/jndi/1.2/javadoc/index.html>

Constructing a JNDI Context

When OC4J starts up, it constructs a JNDI context for each application that is deployed in the server. There is always at least one application for an OC4J server, the global application, which is the default parent for each application in a server instance. User applications inherit properties from the global application and can override property values defined in the global application, define new values for properties, and define new properties as required.

For more information about configuring the OC4J server and its contained applications, see the *Oracle Application Server Containers for J2EE User's Guide*.

Note: During EJB deployment in OC4J, you load the bean class to find out its methods so that you can generate EJB wrappers. Because the code in the static block is executed as the class is being loaded, the JNDI environment context is not yet set up. Even during runtime, the bean is in the "does not exist" stage. In this stage of the life cycle, the JNDI environment context is undefined, and the bean provider cannot rely on it to be available.

To work around this problem, set up and cache the context either during the construction of the bean, in the `ejbCreate()` method, or in the `setSessionContext()` method.

The environment that OC4J uses to construct a JNDI initial context can be found in three places:

- System property values, as set either by the OC4J server or possibly by the application container.
- A `jndi.properties` file contained in the application EAR file (as part of `application-client.jar`).
- An environment specified explicitly in a `java.util.Hashtable` instance passed to the JNDI initial context constructor. ("[Accessing Objects from an Application Client](#)" on page 2-6 shows a code example of this constructor.)

The JNDI Environment

The `JNDI InitialContext` has two constructors:

```
InitialContext()
InitialContext(Hashtable env)
```

The first constructor creates a `Context` object, using the default context environment. If you use this constructor in an OC4J server-side application, then OC4J creates the initial context when the server is started, using the default environment for that application. This constructor is typically used in code that runs on the server side, such as in a JSP, EJB, or servlet.

The second constructor takes an environment parameter. You normally use the second form of the `InitialContext` constructor in client applications, where it is necessary to specify the JNDI environment. The `env` parameter in this constructor is a `java.util.Hashtable` that contains properties required by JNDI. [Table 2-1](#) lists these properties, which are defined in the `javax.naming.Context` interface.

Table 2-1 *InitialContext Properties*

Property	Meaning
<code>INITIAL_CONTEXT_FACTORY</code>	Value for the <code>java.naming.factory.initial</code> property. This property specifies which initial context factory to use when creating a new initial context object.
<code>PROVIDER_URL</code>	Value for the <code>java.naming.provider.url</code> property. This property specifies the URL that the application client code uses to look up objects on the server. Also used by <code>RMIInitialContextFactory</code> and <code>ApplicationClientInitialContextFactory</code> to search for objects in different applications. See Table 2-2 , " JNDI-Related Environment Properties " on page 2-5 for details.

Table 2-1 (Cont.) InitialContext Properties

Property	Meaning
SECURITY_PRINCIPAL	Value for the <code>java.naming.security.principal</code> property. This property specifies the user name. Required in application client code to authenticate the client. Not required for server-side code, because the authentication has already been performed.
SECURITY_CREDENTIAL	Value for the <code>java.naming.security.credential</code> property. This property specifies the password. Required in application client code to authenticate the client. Not required for server-side code, because the authentication has already been performed.

See "[Accessing Objects from an Application Client](#)" on page 2-6 for a code example that sets these properties and gets a new JNDI initial context.

Creating the Initial Context in OC4J

Section 9.1 of the J2EE 1.3 specification defines application clients as follows:

"... first tier client programs that execute in their own Java virtual machines. Application clients follow the model for Java technology-based applications: they are invoked at their main method and run until the virtual machine is terminated. However, like other J2EE application components, application clients depend on a container to provide system services. The application client container may be very light-weight compared to other J2EE containers, providing only the security and deployment services described [in this specification]."

The following sections describe the ways in which JNDI initial contexts can be used:

- [From J2EE Application Clients](#)
- [From J2EE Application Components](#)

From J2EE Application Clients

When an application client must look up a resource that is available in a J2EE server application, the client uses `ApplicationClientInitialContextFactory` in the `com.evermind.server` package to construct the initial context.

Note: If your application is a J2EE client (that is, it has an `application-client.xml` file), then you must always use `ApplicationClientInitialContextFactory` regardless of the protocol (ORMI or IIOP) that the client application is using. The protocol itself is specified by the JNDI property `java.naming.provider.url`. See [Table 2-2, "JNDI-Related Environment Properties"](#) on page 2-5 for details.

Consider an application client that consists of Java code running outside the OC4J server, and is also part of a bundled J2EE application. For example, the client code is running on a workstation and might connect to a server object, such as an EJB, to perform some application task. In this case, the environment that is accessible to JNDI must specify the value of the property `java.naming.factory.initial` as `ApplicationClientInitialContextFactory`. This can be specified in client

code, or it can be specified in the `jndi.properties` file that is part of the `application-client.jar` file included in the EAR file.

To have access to remote objects that are part of the application, `ApplicationClientInitialContextFactory` reads the `META-INF/application-client.xml` and `META-INF/orion-application-client.xml` files in the `application-client.jar` file.

When clients use the `ApplicationClientInitialContextFactory` to construct JNDI initial contexts, they can look up local objects (objects contained in the immediate application or in its parent application) using the `java:comp/env` mechanism and `RMIInitialContextFactory`. They can then use ORMI or IIOP to invoke methods on these objects. Note that objects and resources must be defined in deployment descriptors in order to be bound to the JNDI context of an application.

Environment Properties

If the ORMI protocol is being used, `ApplicationClientInitialContextFactory` reads the properties listed in [Table 2-2](#) from the environment.

Table 2-2 JNDI-Related Environment Properties

Property	Meaning
<code>dedicated.rmicontext</code>	<p>This property replaces the deprecated <code>dedicated.connection</code> setting. When two or more clients in the same process retrieve an <code>InitialContext</code>, OC4J returns a cached context. Thus, each client receives the same <code>InitialContext</code>, which is assigned to the process. Server lookup, which results in server load balancing, happens only if the client retrieves its own <code>InitialContext</code>. If you set <code>dedicated.rmicontext=true</code>, then each client receives its own <code>InitialContext</code> instead of a shared context. When each client has its own <code>InitialContext</code>, then the clients can be load balanced.</p> <p>The <code>dedicated.rmicontext</code> property defaults to false.</p>
<code>java.naming.provider.url</code>	<p>This property specifies the URL to use when looking for local or remote objects. The format is either <code>[http: https:]ormi://hostname/appname</code> or <code>corbaname:hostname:port</code>. For details on the <code>corbaname</code> URL, see "The corbaname URL" on page 6-10.</p> <p>You can supply multiple hosts (for failover) in a comma-separated list.</p>
<code>java.naming.factory.url.pkgs</code>	<p>Some versions of the JDK on some platforms automatically set the system property <code>java.naming.factory.url.pkgs</code> to include <code>com.sun.java.*</code>. Check this property and remove <code>com.sun.java.*</code> if it is present.</p>
<code>http.tunnel.path</code>	<p>This property specifies an alternative <code>RMIHttpTunnelServlet</code> path. The default path is <code>/servlet/rmi</code>, as bound to the target site Web application. For more information, see "Configuring ORMI Tunneling through HTTP" on page 5-11.</p>

Table 2–2 (Cont.) JNDI-Related Environment Properties

Property	Meaning
<code>Context.SECURITY_PRINCIPAL</code>	This property specifies the user name and is required in client-side code to authenticate the client. It is not required for server-side code because authentication has already been performed. This property name is also defined as <code>java.naming.security.principal</code> .
<code>Context.SECURITY_CREDENTIAL</code>	This property specifies the password and is required in client-side code to authenticate the client. It is not required for server-side code because authentication has already been performed. This property name is also defined as <code>java.naming.security.credentials</code> .

Accessing Objects from an Application Client

This section contains an example of how to configure an application client to access an EJB running inside an OC4J instance in the same location.

First, the EJB is deployed into OC4J. Here are excerpts of the deployment descriptors of the EJB.

The EJB is deployed with the name `EmployeeBean`. The name is defined this way in `ejb-jar.xml`:

```
<ejb-jar>
  <display-name>bmpapp</display-name>
  <description>
    An EJB app containing only one Bean Managed Persistence Entity Bean
  </description>
  <enterprise-beans>
    <entity>
      <description>no description</description>
      <display-name>EmployeeBean</display-name>
      <ejb-name>EmployeeBean</ejb-name>
      <home>bmpapp.EmployeeHome</home>
      <remote>bmpapp.Employee</remote>
      <ejb-class>bmpapp.EmployeeBean</ejb-class>
      <persistence-type>Bean</persistence-type>
      ...
    </entity>
  </enterprise-beans>
..
</ejb-jar>
```

The EJB `EmployeeBean` is bound to the JNDI location

`java:comp/env/bmpapp/EmployeeBean` in `orion-ejb-jar.xml`:

`orion-ejb-jar.xml` file:

```
<orion-ejb-jar>
  <enterprise-beans>
    <entity-deployment name="EmployeeBean"
      location="bmpapp/EmployeeBean" table="EMP">
      ...
    </entity-deployment>
    ...
  </enterprise-beans>
  ...
</orion-ejb-jar>
```

The application client program uses the `EmployeeBean` EJB, and refers to it as `EmployeeBean`. An excerpt from the application client program follows:

```
public static void main (String args[])
{
    ...
    Context context = new InitialContext();
    /**
     * Look up the EmployeeHome object. The reference is retrieved from the
     * application-local context (java:comp/env). The variable is
     * specified in the assembly descriptor (META-INF/application-client.xml).
     */
    Object homeObject =
        context.lookup("java:comp/env/EmployeeBean");
    // Narrow the reference to an EmployeeHome.
    EmployeeHome home =
        (EmployeeHome) PortableRemoteObject.narrow(homeObject,
                                                    EmployeeHome.class);

    // Create a new record and narrow the reference.
    Employee rec =
        (Employee) PortableRemoteObject.narrow(home.create(empNo,
                                                            empName,
                                                            salary),
                                                Employee.class);

    // call method on the EJB
    rec.doSomething();
    ...
}
```

Note that we are not passing a hash table when creating a context in the line:

```
Context context = new InitialContext();
```

This is because the context is created with values read from the `jndi.properties` file, which in this example contains:

```
java.naming.factory.initial=com.evermind.server.ApplicationClientInitialContextFactory
java.naming.provider.url=ormi://localhost/bmpapp
java.naming.security.principal=SCOTT
java.naming.security.credentials=TIGER
```

Alternatively, you can pass a hash table to the constructor of `InitialContext` instead of supplying a `jndi.properties` file. The code looks like this:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
"com.evermind.server.ApplicationClientInitialContextFactory");
env.put("java.naming.factory.initial",
"com.evermind.server.ApplicationClientInitialContextFactory");
env.put("java.naming.provider.url", "ormi://localhost/bmpapp");
env.put("java.naming.security.principal", "SCOTT");
env.put("java.naming.security.credentials", "TIGER");
Context initial = new InitialContext(env);
```

Because the application client code refers to the `EmployeeBean` EJB, you must declare this in the `<ejb-ref>` element in the `application-client.xml` file:

```
<application-client>
  <display-name>EmployeeBean</display-name>
  <ejb-ref>
    <ejb-ref-name>EmployeeBean</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <home>bmpapp.EmployeeHome</home>
    <remote>bmpapp.Employee</remote>
  </ejb-ref>
```

```
</application-client>
```

Recall that the `EmployeeBean` EJB is bound to the JNDI location `java:comp/env/bmpapp/EmployeeBean` as configured in the `orion-ejb-jar.xml` file. You must map the EJB name used in the application client program to the JNDI location where the EJB is actually bound to. You must do this in the `orion-application-client.xml` file:

`orion-application-client.xml` file:

```
<orion-application-client>
  <ejb-ref-mapping name="EmployeeBean" location="bmpapp/EmployeeBean" />
</orion-application-client>
```

From J2EE Application Components

You can use initial context factories in OC4J to access the following objects from J2EE application components:

- [Objects in the Same Application](#)
- [Objects Not in the Same Application](#)

Objects in the Same Application

You can use J2EE application components to access objects in the same application from servlets, JSP pages, and EJBs.

When code is running in a server, it is, by definition, part of an application. Because the code is part of an application, OC4J can establish defaults for properties that JNDI uses. Application code does not need to provide any property values when constructing a JNDI `InitialContext` object.

When this context factory is being used, the `ApplicationContext` is specific to the current application, so all the references specified in files such as `web.xml`, `orion-web.xml`, or `ejb-jar.xml` for that application are available. This means that a lookup using `java:comp/env` works for any resource that the application has specified. Lookups using this factory are performed locally in the same Java virtual machine (JVM).

If your application must look up a remote reference, such as a resource in another J2EE application in the same JVM or a resource external to any J2EE application, then you must use `RMIInitialContextFactory` or `IIOPInitialContextFactory`. See ["Objects Not in the Same Application"](#) on page 2-9.

Example As a concrete example, consider a servlet that must retrieve a data source to perform a JDBC operation on a database.

Specify data source location in `data-sources.xml` as follows:

```
<data-source
  class="oracle.jdbc.pool.OracleConnectionCacheImpl"
  location="jdbc/pool/OracleCache"
  username="hr"
  password="hr"
  url="jdbc:oracle:thin:@//<hostname>:<TTC port>/<DB ID>"
/>
```

For more information on data source locations, see [Chapter 4, "Data Sources"](#).

The servlet `web.xml` file defines the following resource:

```
<resource-ref>
```

```

<description>
  A data source for the database in which
  the EmployeeService enterprise bean will
  record a log of all transactions.
</description>
<res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
<res-type>javax.sql.DataSource</res-type>
<res-auth>Container</res-auth>
<res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>

```

The corresponding `orion-web.xml` mapping is:

```
<resource-ref-mapping name="jdbc/EmployeeAppDB" location="jdbc/pool/OracleCache" />
```

The name value is the same as that specified in the `<res-ref-name>` element in `web.xml`. The location value is the location or `ejb-location` in the `<data-source>` element of `data-sources.xml`.

In this case, the following code in the servlet returns the correct reference to the data source object:

```

...
try {
    InitialContext ic = new InitialContext();
    ds = (DataSource) ic.lookup("java:comp/env/jdbc/EmployeeAppDB");
    ...
}
catch (NamingException ne) {
    throw new ServletException(ne);
}
...

```

No initial context factory specification is necessary, because OC4J sets `ApplicationInitialContextFactory` as the default value of the system property `java.naming.factory.initial` when the application starts.

There is no need to supply a provider URL in this case, because no URL is required to look up an object contained within the same application or under `java:comp/`.

Note: Some versions of the JDK on some platforms automatically set the system property `java.naming.factory.url.pkgs` to include `com.sun.java.*`. Check this property and remove `com.sun.java.*` if it is present.

An application can use the `java:comp/env` mechanism to look up resources that are specified not only in its own name space, but also in the name spaces of any declared parent applications, or in the global application (which is the default parent if no specific parent application was declared).

Objects Not in the Same Application

Use one of the following context factories to access objects not in the same application:

- [RMIIInitialContextFactory](#)
- [IIOPInitialContextFactory](#)

RMIInitialContextFactory For most application purposes, you can use either the default server-side `ApplicationInitialContextFactory` or the `ApplicationClientInitialContextFactory`. In some cases, however, you must use an additional context factory:

- If your client application does not have an `application-client.xml` file, then you must use the `RMIInitialContextFactory` property and not the `ApplicationClientInitialContextFactory` property.
- If your client application accesses the JNDI name space remotely—not in the context of a specific application—then you must use `RMIInitialContextFactory`.

The `RMIInitialContextFactory` uses the following environment properties, which `ApplicationClientInitialContextFactory` also uses. [Table 2-2](#) on page 2-5 lists these properties.

- `java.naming.provider.url`
- `http.tunnel.path`
- `Context.SECURITY_PRINCIPAL`
- `Context.SECURITY_CREDENTIALS`

Here is an example of a servlet that accesses an EJB running on another OC4J instance on a different machine. The EJB in this example is the `EmployeeBean` that is used in the ["Accessing Objects from an Application Client"](#) on page 2-6.

Here is an excerpt of the servlet code:

```
Hashtable env = new Hashtable();
env.put("java.naming.factory.initial",
"com.evermind.server.rmi.RMIInitialContextFactory");
env.put("java.naming.provider.url", "ormi://remotehost/bmpapp");
env.put("java.naming.security.principal", "SCOTT");
env.put("java.naming.security.credentials", "TIGER");
Context context = new InitialContext(env);
Object homeObject =
context.lookup("java:comp/env/EmployeeBean");
```

As in the case of the application client, you must declare `<ejb-ref>` elements in the `web.xml` file for this servlet:

```
<ejb-ref>
<ejb-ref-name>EmployeeBean</ejb-ref-name>
<ejb-ref-type>Entity</ejb-ref-type>
<home>bmpapp.EmployeeHome</home>
<remote>bmpapp.Employee</remote>
</ejb-ref>
```

In addition, `orion-web.xml`, must include a mapping from the logical name `EmployeeBean` to the actual JNDI name where the EJB is bound, as shown in the following example:

```
<ejb-ref-mapping name="EmployeeBean" location="bmpapp/EmployeeBean" />
```

IIOPInitialContextFactory The conditions under which to use this factory are the same as those for `RMIInitialContextFactory` except that the protocol being used is IIOP instead of ORMI.

Note: You can use this factory only for looking up EJBs.

JNDI State Replication

JNDI state replication ensures that changes made to the context on one OC4J instance of an OC4J cluster is replicated to the name space of every other OC4J instance.

When JNDI state replication is enabled, you can bind a serializable value into an application context (using a remote client, EJB, or servlet) on one server and read it on another server. You can also create and destroy subcontexts in this way.

This section explains:

- [Enabling JNDI State Replication](#)
- [JNDI State Replication Limitations](#)

Enabling JNDI State Replication

JNDI state replication is enabled when EJB clustering is enabled.

To take advantage of JNDI state replication, you must enable EJB clustering, even if you do not specifically require EJB clustering (for example, when using JNDI to find startup classes or data sources).

For information on enabling EJB clustering, see the EJB clustering chapter in the *Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide*.

For information on OC4J clustering in general, see the clustering chapter in the *Oracle Application Server Containers for J2EE User's Guide*.

JNDI State Replication Limitations

Consider the following limitations when relying on JNDI state replication:

- [Multiple Islands on a Given Subnet](#)
- [Propagating Changes Across the Cluster](#)
- [Binding a Remote Object](#)

Multiple Islands on a Given Subnet

Although OC4J processes can be organized into groups (known as islands) to improve state-replication performance, EJB applications replicate state between all OC4J processes in the OC4J instance and do not use the island subgrouping. This is described in the clustering chapter in the *Oracle Application Server Containers for J2EE User's Guide*.

Consequently, JNDI clustering is not limited to an island subnet. If there are multiple islands on a single subnet, then all islands on that subnet share the global JNDI context.

Propagating Changes Across the Cluster

Rebinding (renaming) and unbinding are not propagated: they apply locally but are not shared across the cluster.

Bindings to values that are not serializable are also not propagated across the cluster.

Binding a Remote Object

If you bind a remote object (typically a home or EJB object) in an application context, then that JNDI object is shared across the cluster but there is a single point of failure if the first server it is bound to fails.

JNDI Lookups in a Multiple-Instance Environment

In an environment with multiple OC4J instances, JNDI lookups require additional information:

- opmn prefix
- ons host name
- ons port number

This applies to EJBs and JMS, as well.

Java Message Service (JMS)

This chapter discusses the following topics:

- [Overview](#)
- [Oracle Application Server JMS](#)
- [Resource Providers](#)
- [Oracle JMS](#)
- [Map Logical Names in Resource References to JNDI Names](#)
- [Third-Party JMS Providers](#)
- [Using Message-Driven Beans](#)
- [High Availability and Clustering for JMS](#)

Download the JMS example used in this chapter from the OC4J sample code page on the OTN Web site at

<http://www.oracle.com/technology/tech/java/oc4j/demos/index.html>

Overview

Java clients and Java middle-tier services must be capable of using enterprise messaging systems. Java Message Service (JMS) offers a common way for Java programs to access these systems. JMS is the standard messaging API for passing data between application components and allowing business integration in heterogeneous and legacy environments.

JMS provides two programming models:

- **Point-to-Point**—Messages are sent to a single consumer using a JMS queue.
- **Publish and Subscribe**—Messages are broadcast to all registered listeners through JMS topics.

JMS queues and topics are bound in the JNDI environment and made available to J2EE applications.

You can choose between several JMS providers, depending on their integration and quality-of-service (QOS) requirements, as follows:

- **Oracle Application Server JMS**—A JMS provider that is installed with OC4J and executes in-memory.
- **Oracle JMS (OJMS)**—A JMS provider that is a feature of the Oracle database and is based on the Streams Advanced Queuing messaging system.

- [Third-Party JMS Providers](#)—You can integrate with the following third-party JMS providers: WebSphere MQ, SonicMQ, and SwiftMQ.

Oracle Application Server JMS

OracleAS JMS is a Java Message Service that provides the following features:

- Complies with the JMS 1.0.2b specification.
- Offers a choice between in-memory or file-based message persistence.
- Provides an exception queue for undeliverable messages.

This section covers the following topics:

- [Configuring OracleAS JMS Ports](#)
- [Configuring OracleAS JMS Destination Objects](#)
- [Steps for Sending a Message](#)
- [OracleAS JMS Utilities](#)
- [OracleAS JMS File-Based Persistence](#)
- [Abnormal Termination](#)
- [Predefined OracleAS JMS Exception Queue](#)
- [Message Paging](#)
- [OracleAS JMS Configuration File Elements for jms.xml](#)
- [OracleAS JMS System Properties](#)

Configuring OracleAS JMS Ports

You can use the Oracle Enterprise Manager 10g to configure the port range for OracleAS JMS. The default range is between 3201 and 3300.

From the OC4J Home page, select the Administration page, then the Instance Properties column, then Server Properties. Scroll to the Multiple VM Configuration section.

Configuring OracleAS JMS Destination Objects

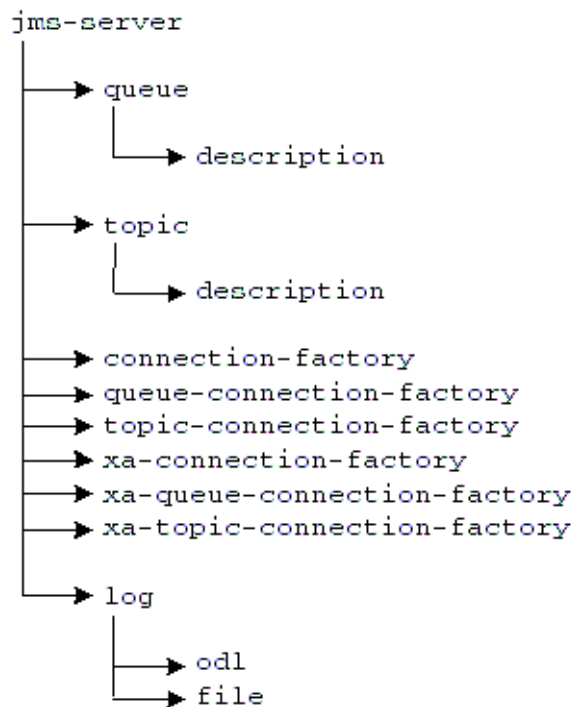
OracleAS JMS `Destination` objects are configured in the `jms.xml` file. OracleAS JMS `Destination` objects can be either queues or topics. OracleAS JMS is already installed with OC4J, so the only configuration necessary is for the queues, topics, and their connection factories that your applications use.

- Oracle Enterprise Manager 10g configuration—To edit the `jms.xml` file directly through Oracle Enterprise Manager 10g, select Advanced Properties under the Instance Properties column on the Administration page. In this section, choose `jms.xml` to modify the straight XML file.
- Standalone OC4J configuration—You can configure the default `jms.xml` file in `J2EE_HOME/config/jms.xml`. If you want, you can change the name and location of this file. To modify the name and location of the JMS configuration file, specify the new name and location in the OC4J server configuration file—`J2EE_HOME/config/server.xml`. The `server.xml` file designates the name and location of the JMS configuration file through the `<jms-config>` element.

Note: Configuration changes made to OracleAS JMS (by modifying `jms.xml`) take effect only on OC4J restart, or shutdown and start.

Figure 3–1 shows the order in which the elements in the `jms.xml` file are structured. "OracleAS JMS Configuration File Elements for `jms.xml`" on page 3-16 provides a complete description of all elements and their attributes in the `jms.xml` file.

Figure 3–1 Configuration Elements Hierarchy



The `jms.xml` file defines the topics and queues used. For each Destination object (queue or topic)—you must specify its name (also known as its location) and connection factory in the `jms.xml` file. The following `jms.xml` file example configuration defines a queue used by the `Oc4jJmsDemo` demo.

The queue is defined as follows:

- The name (location) of the queue is `jms/demoQueue`.
- Its queue connection factory is defined as `jms/QueueConnectionFactory`.

The topic is defined as follows:

- The name (location) of the topic is `jms/demoTopic`.
- Its topic connection factory is defined as `jms/TopicConnectionFactory`.

```

<?xml version="1.0" ?>
<!DOCTYPE jms-server PUBLIC "OracleAS JMS server"
"http://xmlns.oracle.com/ias/dtlds
  
```

```
/jms-server.dtd">  
  
<jms-server port="9127">  
  <queue location="jms/demoQueue"> </queue>  
  <queue-connection-factory location="jms/QueueConnectionFactory">  
  </queue-connection-factory>  
  
  <topic location="jms/demoTopic"> </topic>  
  <topic-connection-factory location="jms/TopicConnectionFactory">  
  </topic-connection-factory>  
  
  <!-- path to the log-file where JMS-events/errors are stored -->  
  <log>  
    <file path="../../log/jms.log" />  
  </log>  
</jms-server>
```

Note: Because all of these values are defaults, you do not have to configure them. However, the example shows the configuration for the queue, the topic, and their connection factories so that you understand how to configure your own `Destination` objects and connection factories.

See "[OracleAS JMS Configuration File Elements for jms.xml](#)" on page 3-16 for descriptions of the elements in the `jms.xml` file.

Default Destination Objects

OracleAS JMS creates two default `Destination` objects, as follows:

- The default queue is defined as `jms/demoQueue`.
- The default topic is defined as `jms/demoTopic`.

You can use these `Destination` objects in your code without adding them to the `jms.xml` configuration file.

The default connection factories that are automatically associated with these objects are as follows:

- `jms/QueueConnectionFactory`
- `jms/TopicConnectionFactory`

Default Connection Factories

OracleAS JMS creates six default connection factories for the XA/non-XA and various JMS domains. You can use these connection factories in your code without adding them to the `jms.xml` configuration file, rather than defining new connection factories. The only reason to define a new connection factory in the `jms.xml` file is if you need to specify nondefault values for one or more of the optional attributes of `connection-factory` elements.

The default connection factories are as follows:

- `jms/ConnectionFactory`
- `jms/QueueConnectionFactory`
- `jms/TopicConnectionFactory`
- `jms/XAConnectionFactory`

- `jms/XAQueueConnectionFactory`
- `jms/XATopicConnectionFactory`

Thus, if you used only the default connection factories, then you could define only the topic and queues necessary in the `jms.xml` file. The following example defines the `jms/demoQueue` and the `jms/demoTopic`. Both of these objects use their respective default connection factories.

```
<?xml version="1.0" ?>
<!DOCTYPE jms-server PUBLIC "OracleAS JMS server"
"http://xmlns.oracle.com/ias/dtds
/jms-server.dtd">

<jms-server port="9127">
  <queue location="jms/demoQueue"> </queue>
  <topic location="jms/demoTopic"> </topic>
  <!-- path to the log-file where JMS-events/errors are stored -->
  <log>
    <file path="../../log/jms.log" />
  </log>
</jms-server>
```

OracleAS JMS internally creates the default connection factory objects and binds them to the default names within the OC4J server where the JMS connection is created.

However, you can also redefine the default connection factories to have specific attributes by configuring them in the `jms.xml` file.

Steps for Sending a Message

A JMS client sends a JMS message by doing the following:

1. Retrieve both the configured JMS `Destination` object (queue or topic) and its connection factory using a JNDI lookup.
2. Create a connection from the connection factory.
3. If you are receiving messages, then start the connection.
4. Create a session using the connection.
5. Provide the retrieved JMS `Destination`, create a sender for a queue, or a publisher for a topic.
6. Create the message.
7. Send out the message using either the queue sender or the topic publisher.
8. Close the queue session.
9. Close the connection for either JMS `Destination` types.

[Example 3-1](#) demonstrates these steps for sending a JMS message. For the complete example, download the JMS example used in this chapter from the OC4J sample code page on the OTN Web site at

<http://www.oracle.com/technology/tech/java/oc4j/demos/index.html>

Note: For simplicity, most of the error code is removed in [Example 3-1](#). To see the error processing, see the sample code available on the OTN Web site.

Example 3-1 OracleAS JMS Client that Sends Messages to a Queue

The JNDI lookup for OracleAS JMS requires that the OracleAS JMS Destination and connection factory be defined within the `jms.xml` file, prepended with the `java:comp/env/` prefix.

Note: Alternatively, you could use logical names in the JNDI lookup. See ["Map Logical Names in Resource References to JNDI Names"](#) on page 3-33 for directions. The only difference between an OracleAS JMS client and an OJMS client is the name provided in the JNDI lookup. To make your client independent of either JMS provider, use logical names in the implementation, and change only the OC4J-specific deployment descriptor.

The `dosend` method, shown in the following example, sets up a queue to send messages. After creating the queue sender, this example sends out several messages.

```
public static void dosend(int nmsgs)
{
    // 1a. Retrieve the queue connection factory.
    QueueConnectionFactory qcf = (QueueConnectionFactory)
        ctx.lookup("java:comp/env/jms/QueueConnectionFactory");
    // 1b. Retrieve the queue.
    Queue q = (Queue) ctx.lookup("java:comp/env/jms/demoQueue");

    // 2. Create the JMS connection.
    QueueConnection qc = qcf.createQueueConnection();
    // 3. Start the queue connection.
    qc.start();
    // 4. Create the JMS session over the JMS connection.
    QueueSession qs = qc.createQueueSession(false,
        Session.AUTO_ACKNOWLEDGE);
    //5. Create a sender on the JMS session to send messages.
    QueueSender snd = qs.createSender(q);

    // Send out messages...
    for (int i = 0; i < nmsgs; ++i)
    {
        //6. Create the message using the createMessage method of the
        // JMS session.
        Message msg = qs.createMessage();
        //7. Send the message out over the sender (snd) using the
        // send method.
        snd.send(msg);
        System.out.println("msg:" + " id=" + msg.getJMSMessageID());
    }

    //8 & 9 Close the sender, the JMS session and the JMS connection.
    snd.close();
    qs.close();
    qc.close();
}
}
```

Steps for Receiving a Message

A JMS client receives a JMS message by doing the following:

1. Retrieve both the configured JMS Destination object (queue or topic) and its connection factory using a JNDI lookup.
2. Create a connection from the connection factory.
3. If you are receiving messages, then start the connection.
4. Create a session using the connection.
5. Providing the retrieved JMS Destination, create a receiver for a queue or a topic subscriber.
6. Receive the message using the queue receiver or the topic subscriber.
7. Close the queue session.
8. Close the connection for either JMS Destination types.

Example 3–2 demonstrates these steps for receiving a JMS message. For the complete example, download the JMS example used in this chapter from the OC4J sample code page on the OTN Web site at

<http://www.oracle.com/technology/tech/java/oc4j/demos/index.html>

Note: For simplicity, most of the error code is removed in [Example 3–2](#). To see the error processing, see the sample code available on the OTN Web site.

Example 3–2 OracleAS JMS Client That Receives Messages Off a Queue

The `dorcv` method, shown in the following example, sets up a queue to receive messages off of it. After creating the queue receiver, it loops to receive all messages off the queue and compares it to the number of expected messages.

```
public static void dorcv(int nmsgs)
{
    Context ctx = new InitialContext();

    // 1a. Retrieve the queue connection factory.
    QueueConnectionFactory qcf = (QueueConnectionFactory)
        ctx.lookup("java:comp/env/jms/QueueConnectionFactory");
    // 1b. Retrieve the queue.
    Queue q = (Queue) ctx.lookup("java:comp/env/jms/demoQueue");

    // 2. Create the JMS connection.
    QueueConnection qc = qcf.createQueueConnection();
    // 3. Start the queue connection.
    qc.start();
    // 4. Create the JMS session over the JMS connection.
    QueueSession qs = qc.createQueueSession(false,
        Session.AUTO_ACKNOWLEDGE);
    // 5. Create a receiver, as we are receiving off of the queue.
    QueueReceiver rcv = qs.createReceiver(q);

    // 6. Receive the messages.
    int count = 0;
    while (true)
    {
        Message msg = rcv.receiveNoWait();
        System.out.println("msg:" + " id=" + msg.getJMSMessageID());
        ++count;
    }
}
```

```

    if (nmsgs != count)
    {
        System.out.println("expected: " + nmsgs + " found: " + count);
    }

    // 7 & 8 Close the receiver, the JMS session and the JMS connection.
    rcv.close();
    qs.close();
    qc.close();
}

```

OracleAS JMS Utilities

OC4J JMS comes with an OC4J-specific command-line utility, `com.evermind.server.jms.JMSUtils`, for debugging and information access.

The path `J2EE_HOME/oc4j.jar` must be in the CLASSPATH variable. Then execute `JMSUtils`, as follows:

```

java com.evermind.server.jms.JMSUtils [gen_options] [command]
                                     [command_options]

```

The OracleAS JMS server must be running, and only the administrator can use `JMSUtils`. You define a user within the administrator role in the security User Manager. For information on defining users within security roles, see the

Oracle Application Server Containers for J2EE Security Guide.

The generic options for `JMSUtils` facilitate connecting to the OracleAS JMS server. [Table 3–1](#) describes these options.

Table 3–1 JMSUtils Options

Option	Description
<code>-host <hostname></code>	The (remote) host where the OracleAS JMS server is installed. This is not required if the client exists on the same node as the OracleAS JMS server.
<code>-port <port></code>	The (remote) port through which the OracleAS JMS server is accessed. The default JMS port number is 9127.
<code>-username <username></code>	The username to access the OracleAS JMS server for creating the JMS connection. This user is defined in the User Manager security configuration within the administrative roles.
<code>-password <password></code>	The password to access the OracleAS JMS server for creating the JMS connection. This password is defined in the User Manager security configuration within the administrative roles.
<code>-clientID <ID></code>	Use this identifier for all JMS connections. This is required only for identifying durable subscriptions on topics.

The commands describe the action to be taken and are discussed in [Table 3–2](#). Some of these commands have their own options (`command_options`) to further describe the action desired.

To display the syntax usage, issue the `JMSUtils` command with no argument. To display extensive information about the available command set, the argument options, and the behavior of each command, issue the following:

```

java com.evermind.server.jms.JMSUtils help

```

Table 3–2 OC4J JMS Utilities

Utility	Command Description
help	Print detailed help for all utilities commands.
check [<other-selector>]	Check validity of a JMS message selector, identified by the <code>-selector</code> command option. Optionally, check if two specified selectors are treated as equivalent (useful for reactivating durable subscriptions), where the second selector is identified in the optional <code><other-selector></code> .
knobs	Display all available system properties (shown in Table 3–6) and their current settings on the OC4J JMS server.
stats	Display all available DMS statistics on the OC4J JMS server (this includes non-JMS statistics as well). (For information on DMS, see the <i>Oracle Application Server Performance Guide</i> .)
destinations	Print a list of all permanent <code>Destination</code> objects known to OC4J JMS.
durables	Print a list of all durable subscriptions known to OC4J JMS.
subscribe <topic>	Create a new durable subscription on the <code><topic></code> . Specify a name, message selector, whether it is local, and a durable subscription client identifier. This replaces existing, inactive durable subscriptions. The name is identified using the <code>-name</code> command option. The message selector is identified using the <code>-selector</code> command option. Whether the durable subscription is local or not is identified using the <code>-noLocal</code> command option. The client identifier is defined with the <code>-clientID</code> generic option.
unsubscribe	Drop an existing, inactive durable subscription. The durable subscription is identified by a name (<code>-name</code> command option) and the client identifier (<code>-clientID</code> generic option).
browse <destination>	Browse messages on a given destination (queue or topic durable subscription, defined in <code>jms.xml</code>).
drain <destination>	Dequeue messages on a given destination (queue or topic durable subscription).
copy <from-destination> <to-destination>	Copy messages from one destination (queue or topic durable subscription) to a different destination. If the source and sink destinations are the same, then the command is not executed, generating an error instead.
move <from-destination> <to-destination>	Move messages from one destination (queue or topic durable subscription) to a different destination. If the source and sink destinations are the same, then the command is not executed, generating an error instead.

[Table 3–3](#) describes the command options.

Table 3–3 JMSUtils Command Options

Command Option	Description
<code>-selector <selector></code>	Create queue receivers and durable subscribers with the specified JMS message selector.
<code>-noLocal</code> [true false]	If set to <code>true</code> , the subscriber does not see the messages that are published in the same connection. Use when creating a durable subscriber. The default value is <code>false</code> .

Table 3–3 (Cont.) JMSUtils Command Options

Command Option	Description
-name <name>	Defines a name for a durable subscription, operating on a topic. This option is mandatory for commands that read topics, and is ignored for reading queues.
-silent	Do not print messages while processing. Keeps a count of the total number of messages processed, which is printed to standard error.
-count <count>	Do not process more than the indicated number of messages during the current operation. If the count is negative or zero, then all selected messages are processed.

An example of using `JMSUtils` to browse the exception queue is as follows:

```
java com.evermind.server.jms.JMSUtils -username admin -password welcome
      browse jms/Oc4jJmsExceptionQueue
```

OracleAS JMS File-Based Persistence

OC4J JMS supports file-based persistence for JMS `Destination` objects (queues and topics). The following sections provide more detail on file-based persistence:

- [Overview](#)
- [Enabling Persistence](#)
- [Recovery](#)

Overview

If persistence is enabled, then OC4J automatically performs the following:

- If a persistence file does not exist, OC4J automatically creates the file and initializes it with the appropriate data.
- If the persistence file exists and is empty, then OC4J initializes it with the appropriate data.

Caution: A persistence file must not be moved, removed, or modified when the OC4J server is active. Doing so can result in data corruption and message loss. If OC4J is not active, then removing a persistence file is equivalent to deleting all messages and durable subscriptions in the destination associated with that persistence file. When OC4J starts up again, the JMS server reinitializes the file as usual.

Even if persistence is enabled, only certain messages are persisted to a file. For a message to be persisted, all of the following must be true:

- The `Destination` object is defined to be persistent within the `persistence-file` attribute in the `jms.xml` file.
- The message has a persistent delivery mode, which is the default. Messages sent to persistent destinations that are defined with a non-persistent delivery mode (defined as `DeliveryMode.NON_PERSISTENT`) are not persistent.

The complete semantics of which messages are persisted are documented within the JMS specification.

Enabling Persistence

To enable file-based persistence for `Destination` objects, specify the `persistent-file` attribute in the `jms.xml` file. For JMS objects within standalone OC4J, this is all you have to do to enable persistence. The following XML configuration demonstrates how the `persistence-file` attribute defines the name of the file as `pers`. See ["OracleAS JMS Configuration File Elements for `jms.xml`"](#) on page 3-16 for information on the path and naming conventions for the `persistence-file` attribute.

```
<queue name="foo" location="jms/persist" persistence-file="pers">
</queue>
```

The path for the `persistence-file` attribute is either an absolute path of the file or a path relative to the persistence directory defined in `application.xml`.

The JMS server will not create any directories for persistence files. So when a persistence file is defined in `jms.xml` it must either be in an existing absolute directory, for example:

```
persistence-file="/this/dir/exists/PersistenceFile"
```

or simply be a filename for example:

```
persistence-file="PersistenceFile"
```

In the latter case, by default the persistence file will be created in `$J2EE_HOME/persistence` (for a standalone instance) or `$J2EE_HOME/persistence/<island_name>` (for an iAS environment).

Oracle Application Server may have multiple OC4J instances writing to the same file directory, even with the same persistence filename. Setting this attribute enables file-based persistence, but also creates the possibility that your persistence files can be overwritten by another OC4J instance.

Recovery

The file-based persistence of OC4J JMS provides recoverable and persistent storage of messages. Each OC4J JMS `Destination`, which can be either a queue or topic, can be associated with a relative or absolute path name that points to a file that stores the messages sent to the `Destination` object. The file can reside anywhere in the file system (and not necessarily inside a `J2EE_HOME` directory). Multiple persistence files can be placed in the same directory; persistence files can be placed on a remote network file system or can be part of a local file system.

The following sections discuss the various aspects of persistence recovery for OracleAS JMS:

- [Scope of Recoverability](#)
- [Persistence File Management](#)
- [Reporting Errors to the JMS Client](#)
- [OracleAS JMS Recovery Steps](#)

Scope of Recoverability OC4J JMS cannot recover from all possible types of failures. If any of the following failures occur, then OC4J JMS does not guarantee the recoverability of the persistence file.

- Media corruption—the disk system holding the persistence file fails abnormally or gets corrupted.

- External corruption—the persistence file is deleted, edited, modified, or otherwise corrupted (by software). Only the OC4J JMS server should write into a persistence file.
- Silent failure or corruption—the I/O methods in the JDK fail silently or corrupt data that are being read or written silently.
- A `java.io.FileDescriptor.sync()` failure—the `sync()` call does not properly and completely flush all file buffers associated with the given descriptor to the underlying file system.

Persistence File Management When the OC4J JMS server is running, you must not copy, delete, or rename persistence files currently in use. It is an *unrecoverable error* to perform any of these actions on any of these files when they are being used by OC4J JMS.

However, when no OC4J JMS server is using a persistence file, you can perform the following administrative and maintenance operations on these files:

- *deleting*: Deleting a persistence file removes all messages and, in the case of topics, all durable subscriptions. On startup, OC4J JMS initializes a new (and empty) persistence file.
- *copying*: An existing persistence file can be copied for archival or backup purposes. If an existing persistence file gets corrupted, an earlier version can be used (as long as the association between the OC4J JMS `Destination` name and the file is maintained), pointed to by any suitable path name, to go back to the previous contents of the JMS `Destination`.

Persistence files cannot be concatenated, split up, rearranged, or merged. Attempting any of these operations causes unrecoverable corruption of the data in these files.

In addition to persistence files specified by a user and lock files, OC4J JMS uses a special file, `jms.state`, for internal configuration and transaction state management. OC4J JMS cleans up this file and its contents during normal operations. You must never delete, move, copy, or otherwise modify this file, even for archival purposes. Attempting to manipulate the `jms.state` file can lead to message and transaction loss.

Note: The location of the `jms.state` file is different whether you are operating OC4J in standalone or Oracle Application Server mode, as follows:

- Standalone: `J2EE_HOME/persistence` directory
- Oracle Application Server: `J2EE_HOME/persistence/<island_name>` directory

The location of the `persistence` directory is defined in the `application.xml` file.

Reporting Errors to the JMS Client The sequence of operations when a JMS client enqueues or dequeues a message, or commits or rolls back a transaction, is as follows:

- Client makes a function call
 - pre-persistence operations
 - persistence occurs
 - post-persistence operations

- Client function call returns

If a failure occurs during the pre-persistence or persistence phase, then the client receives a `JMSException` or some other type of error, but no changes are made to the persistence file.

If a failure occurs in the post-persistence phase, the client may receive a `JMSException` or some other type of error; however, the persistence file is still updated, and OC4J JMS recovers as if the operation succeeded.

OracleAS JMS Recovery Steps Lock files prevent multiple OC4J processes from writing into the same persistence file. If multiple OC4J JVMs are configured to point to the same file in the same `persistence-file` location, then they could overwrite each others data and cause corruption or loss of persisted JMS messages. To protect against these kinds of sharing violations, OracleAS JMS associates each persistence file with a lock file. Thus, each persistence file—for example, `/path/to/persistenceFile`—is associated with a lock file named `/path/to/persistenceFile.lock`. See ["Enabling Persistence"](#) on page 3-11 for more information on persistence files.

OC4J must have appropriate permissions to create and delete the lock file. If OC4J is terminated normally, then the lock files are cleaned up automatically. However, if OC4J is terminated abnormally, the lock files continue to exist in the file system. Because OC4J cannot distinguish leftover lock files from sharing violations, the user must manually remove all lock files before restarting OC4J after abnormal termination. OracleAS JMS will not attempt to create the relevant persistent JMS destinations if it detects an existing lock file for it.

OC4J JMS never attempts to delete lock files automatically. Lock files must be manually deleted for OC4J JMS to use a given persistence file. The remainder of the discussion in this subsection assumes that all lock files in question have been removed.

Note: This manual intervention is required only on abnormal shutdown. See ["Abnormal Termination"](#) on page 3-14.

OC4J JMS performs recovery operations on all persistence files as configured in OC4J JMS at the time of abnormal termination. In other words, if OC4J terminates abnormally, the user modifies the JMS server configuration and restarts OC4J, the JMS server still attempts to recover all the persistence files in the original configuration, and, after recovery is successful, moves to using the new configuration specified.

If recovery of the old configuration fails, then the OC4J JMS server does not start. The server must be shut down or restarted repeatedly to give recovery another chance, until recovery is successful.

OC4J JMS caches its current persistence configuration in the `jms.state` file, which is also used to maintain transaction state. If you wish to bypass all recovery of the current configuration, you can remove the `jms.state` file, remove all lock files, possibly change the OC4J JMS server configuration, and start the server in a clean-slate mode. (We do not recommend doing this.) If the OC4J JMS server cannot find a `jms.state` file, then it creates a new one.

If, for some reason, the `jms.state` file itself is corrupted, then the only recourse is to delete it, with the attendant loss of all pending transactions—that is, transactions that have been committed, but the commits not yet performed by all individual `Destination` objects participating in the transactions.

If messaging activity was in progress during abnormal termination, then OC4J JMS tries to recover its persistence files. Any data corruption (of the types mentioned

earlier) is handled by clearing out the corrupted data; this may lead to a loss of messages and transactions.

If the headers of a persistence file are corrupted, OC4J JMS may not be able to recover the file, because such a corrupted file is often indistinguishable from user configuration errors. The `oc4j.jms.forceRecovery` administration property (described in [Table 3–6](#)) instructs the OC4J JMS server to proceed with recovery, clearing out all invalid data at the cost of losing messages or masking user configuration errors.

Abnormal Termination

If OC4J terminates normally, then the lock files are cleaned up automatically. However, if OC4J terminates abnormally (for example, a `kill -9` command), the lock files remain in the file system. Because OC4J cannot distinguish leftover lock files from sharing violations, you must manually remove all lock files before restarting OC4J after abnormal termination. OC4J JMS does not even attempt to create the relevant persistent JMS `Destination` objects if it detects already existing lock files for them.

The default location of the lock files is in the persistence directory—`J2EE_HOME/persistence`. (The persistence directory is defined in the `application.xml` file.) Other locations can be set within the `persistence-file` attribute of the `Destination` object.

Predefined OracleAS JMS Exception Queue

As an extension to the JMS specification, OC4J JMS comes with a predefined exception queue for handling undeliverable messages. This is a single, persistent, global exception queue to store undeliverable messages in all of its `Destination` objects. The exception queue has a fixed name (`jms/Oc4jJmsExceptionQueue`), a fixed JNDI location (`jms/Oc4jJmsExceptionQueue`), and a fixed persistence file (`Oc4jJmsExceptionQueue`).

Note: The location of the `Oc4jJmsExceptionQueue` persistence file varies according to whether you are operating OC4J in standalone or Oracle Application Server mode, as follows:

- Standalone: `J2EE_HOME/persistence` directory
- Oracle Application Server: `J2EE_HOME/persistence/<island_name>` directory

The location of the persistence directory is defined in the `application.xml` file.

The exception queue is always available to OC4J JMS and its clients, and should not be explicitly defined in the `jms.xml` configuration file; attempting to do so is an error. The name, JNDI location, and persistence path name of the exception queue must be considered reserved words in their respective name spaces; any attempt to define other entities with these names is an error.

Messages can become undeliverable because of message expiration and listener errors. The following subsection explains what happens to undeliverable messages in the first case.

Message Expiration

By default, if a message sent to a persistent `Destination` expires, then OC4J JMS moves the message to the exception queue. The `JMSXState` of the expiring message is set to the value 3 (for `EXPIRED`), but the message headers, properties, and body are not otherwise modified. The message is wrapped in an `ObjectMessage` (with appropriate property name and value copies performed as described elsewhere in this chapter), and the wrapping message is sent to the exception queue.

To affect the behavior of what goes into the exception queue, use the `oc4j.jms.saveAllExpired` property (described in [Table 3-6](#)).

The wrapping `ObjectMessage` has the same `DeliveryMode` as the original message.

By default, messages expiring on nonpersistent or temporary `Destination` objects are not moved to the exception queue. The messages sent to these `Destination` objects are not worth persisting and neither should their expired versions be.

You can specify that all expired messages be sent to the OC4J JMS exception queue, regardless of whether they are sent to persistent, nonpersistent, or temporary `Destination` objects, by setting the `oc4j.jms.saveAllExpired` administration property (described in [Table 3-6](#)) to `true` when starting the OC4J server. In this case, all expired messages are moved to the exception queue.

Message Paging

The OracleAS JMS server supports paging in and out message bodies under the following circumstances:

- The message has a persistent delivery mode.
- The message is sent to a persistent `Destination` object (see "[OracleAS JMS File-Based Persistence](#)" on page 3-10).
- The OC4J server JVM has insufficient memory.

Only message bodies are paged. Message headers and properties are never considered for paging. You set the paging threshold through the OracleAS JMS system property, `oc4j.jms.pagingThreshold`, which is a double value (narrowed into the range [0,1]) that represents the memory usage fraction above which the OracleAS JMS server considers message bodies for paging. This value is an estimate of the fraction of memory in use by the Java virtual machine (JVM). This value can range from 0.0 (the program uses no memory at all) to 1.0 (the program is using all available memory).

The value ranges from somewhere above 0.0 to somewhere below 1.0: it is almost impossible to write a Java program that uses no JVM memory, and programs almost always die by running out of memory before the JVM heap gets full.

For example, if the paging threshold is 0.5, and the memory usage fraction of the JVM rises to 0.6, OracleAS JMS tries to page out as many message bodies as possible until the memory usage fraction reduces below the threshold, or no more message bodies can be paged out.

When a message that has been paged out is requested by a JMS client, the OracleAS JMS server automatically pages in the message body (regardless of the memory usage in the JVM) and delivers the correct message header/body to the client. After the message has been delivered to the client, it may once again be considered for paging out, depending on the memory usage in the server JVM.

If the memory usage fraction drops below the paging threshold, then the OracleAS JMS server stops paging out message bodies. The bodies of messages already paged

out are not automatically paged back in. The paging in of message bodies happens only on demand (that is, when a message is dequeued or browsed by a client).

By default, the paging threshold of the OracleAS JMS server is set to 1.0. In other words, by default, the OracleAS JMS server never pages message bodies.

Depending on the JMS applications, and the sizes of the messages they send/receive, and the results of experiments and memory usage monitoring on real-life usage scenarios, the user should choose a suitable value for the paging threshold.

No value of the paging threshold is ever incorrect. JMS semantics are always preserved regardless of whether paging in enabled or disabled. Control of the paging threshold does allow the OracleAS JMS server to handle more messages in memory than it might have been able to without paging.

OracleAS JMS Configuration File Elements for `jms.xml`

This section describes the XML elements for OC4J JMS configuration in `jms.xml`. The following is the element order structure within the XML file:

```
<jms-server>
  <queue>
    <description></description>
  </queue>
  <topic>
    <description></description>
  </topic>
  <connection-factory></connection-factory>
  <queue-connection-factory></queue-connection-factory>
  <topic-connection-factory></topic-connection-factory>
  <xa-connection-factory></xa-connection-factory>
  <xa-queue-connection-factory></xa-queue-connection-factory>
  <xa-topic-connection-factory></xa-topic-connection-factory>
  <log>
    <file></file>
  </log>
</jms-server>
```

Table 3–4 defines the JMS configuration elements.

Table 3–4 JMS Configuration Elements

Element	Description	Attributes
<code>jms-server</code>	The root element of the OC4J JMS server configuration.	<p><code>host</code>—The host name defined in a <code>String</code> (DNS or dot-notation host name) to which this OC4J JMS server should bind. By default, the JMS server binds to <code>0.0.0.0</code> (also known as <code>[ALL]</code> in the configuration file). This attribute is optional.</p> <p><code>port</code>—The port defined as an <code>int</code> (valid TCP/IP port number) to which this OC4J JMS server should bind. The default setting is <code>9127</code>. This setting applies only to the standalone configuration of OC4J. In the Oracle Application Server, the port setting in the configuration file is overridden by command-line arguments that are used (by, for example, OPMN, DCM, and others) when starting the OC4J server. This attribute is optional.</p>

Table 3–4 (Cont.) JMS Configuration Elements

Element	Description	Attributes
queue	This element configures OracleAS JMS queues. The queues are available when OC4J JMS starts up, and are available for use until the server is restarted or reconfigured. You can configure zero or more queues in any order. Any newly configured queue is not available until OC4J is restarted.	<p><code>name</code>—This required attribute is the provider-specific name (<code>String</code>) for the OC4J JMS queue. The name can be any valid nonempty string (with white space and other special characters included, although this is not recommended). The name specified here can be used in <code>Session.createQueue()</code> to convert the provider-specific name to a JMS queue. It is invalid for both a queue and a topic to specify the same name. However, multiple queues can specify the same name and different locations. There is no default name.</p> <p><code>location</code>—This required attribute states the JNDI location (<code>String</code>) to which the queue is bound. The value should follow the JNDI rules for valid names. Within the OC4J JMS container, the location is bound and accessible as is. In application clients, the name is part of the <code>java:comp/env/</code> JNDI name space, and should be appropriately declared in the relevant deployment descriptors. The <code>java:comp/env/</code> names can also be used within the container, assuming that the relevant deployment descriptors have been appropriately specified. The location should be unique across all <code>Destination</code> objects and connection factory elements in <code>jms.xml</code>. There is no default.</p> <p><code>persistence-file</code>—An optional path and filename (<code>String</code>). The path for the <code>persistence-file</code> attribute is either an absolute path of the file or a path relative to the <code>persistence</code> directory defined in <code>application.xml</code>; the default path is <code>J2EE_HOME/persistence/<island></code> for Oracle Application Server environments and <code>J2EE_HOME/persistence</code> for standalone environments.</p> <p>See "Recovery" on page 3-11 for more details on the meaning of the <code>persistence-file</code> attribute. If multiple queue elements with the same name and different locations are declared in <code>jms.xml</code>, then all of them should specify the same value for <code>persistence-file</code>, or should not specify the value at all. If at least one of these multiple declarations specifies a <code>persistence-file</code>, that value is used for this queue.</p>

Table 3–4 (Cont.) JMS Configuration Elements

Element	Description	Attributes
topic	This element configures OracleAS JMS topic. The topics are available when OC4J JMS starts up, and are available for use until the server is restarted or reconfigured. You can configure zero or more topics in any order. Any newly configured topic is not available until OC4J is restarted.	<p>name—This required attribute is the provider-specific name (<code>String</code>) for the OC4J JMS topic. The name can be any valid nonempty string (with white space and other special characters included, although this is not recommended). The name specified here can be used in <code>Session.createTopic()</code> to convert the provider-specific name to a JMS topic. It is invalid for both a queue and a topic to specify the same name. However, multiple topics can specify the same name and different locations. There is no default name.</p> <p>location—This required attribute states the JNDI location (<code>String</code>) to which the topic is bound. The value should follow the JNDI rules for valid names. Within the OC4J JMS container, the location is bound and accessible as is. In application clients, the name is part of the <code>java:comp/env/</code> JNDI name space, and should be appropriately declared in the relevant deployment descriptors.</p> <p>The <code>java:comp/env/</code> names can also be used within the container, assuming that the relevant deployment descriptors have been appropriately specified. The location should be unique across all topics and connection factory elements in <code>jms.xml</code>. There is no default.</p> <p>persistence-file—An optional path and filename (<code>String</code>). The path for the <code>persistence-file</code> attribute is either an absolute path of the file or a path relative to the <code>persistence</code> directory defined in <code>application.xml</code>; the default path is <code>J2EE_HOME/persistence/<island></code> for Oracle Application Server environments and <code>J2EE_HOME/persistence</code> for standalone environments.</p> <p>See "Recovery" on page 3-11 for more details on the meaning of the <code>persistence-file</code> attribute. If multiple queue or topic elements with the same name and different locations are declared in <code>jms.xml</code>, then all of them should specify the same value for <code>persistence-file</code>, or should not specify the value at all. If at least one of these multiple declarations specifies a <code>persistence-file</code>, that value is used for this topic.</p>
description	A user-defined string to remind the user for what the queue or topic is used.	
connection-factory	JMS domain connection factory configuration.	Table 3–5 describes the attributes for this element.
queue-connection-factory	JMS domain connection factory configuration.	Table 3–5 describes the attributes for this element.
topic-connection-factory	JMS domain connection factory configuration. Table 3–5 describes all of the attributes for this element.	Table 3–5 describes the attributes for this element.
xa-connection-factory	XA variants of connection factory configuration.	Table 3–5 describes the attributes for this element.

Table 3–4 (Cont.) JMS Configuration Elements

Element	Description	Attributes
xa-queue-connection-factory	XA variants of connection factory configuration.	Table 3–5 describes the attributes for this element.
xa-topic-connection-factory	XA variants of connection factory configuration.	Table 3–5 describes the attributes for this element.
log	Enables logging of the JMS activity in either file or ODL format. See the section "Enabling OC4J Logging" in the <i>Oracle Application Server Containers for J2EE User's Guide</i> for information on logging.	

Table 3–5 describes the attributes for any connection factory definition.

Table 3–5 Connection Factory Configuration Attributes

Attribute	Type	Mandatory?	Default	Description
location	String	Yes	(n/a)	The JNDI location to which the connection factory is bound. The value should follow the JNDI rules for valid names. Within the OC4J JMS container, the location is bound and accessible as is. In application clients, the name is part of the <code>java:comp/env/</code> JNDI name space, and should be appropriately declared in the relevant deployment descriptors. The <code>java:comp/env/</code> names can also be used within the container, assuming that the relevant deployment descriptors have been appropriately specified. The location should be unique across all <code>Destination</code> and connection factory elements in <code>jms.xml</code> .
host	String (DNS or dot notation host name)	No	[ALL]	The fixed OC4J JMS host to which this connection factory will connect. By default, a connection factory uses the same host as configured for the <code>jms-server</code> element. Nondefault values can be used to force all JMS operations to be directed to a specific OC4J JVM, bypassing any locally available OC4J JMS servers and other Oracle Application Server or clustered configurations.
port	int (valid TCP/IP port number)	No	9127	The fixed OC4J JMS port to which this connection factory connects. By default, a connection factory uses the same port as configured for the <code>jms-server</code> element (or the value of the port that was specified for Oracle Application Server or clustered configurations on the command line). Nondefault values can be used to force all JMS operations to be directed to a specific OC4J JVM, bypassing any locally available OC4J JMS servers and other Oracle Application Server or clustered configurations.

Table 3–5 (Cont.) Connection Factory Configuration Attributes

Attribute	Type	Mandatory?	Default	Description
username	String	No	The empty string	The user name for the authentication of JMS default connections created from this connection factory. The user name itself must be properly created and configured with other OC4J facilities.
password	String	No	The empty string	The password for the authentication of JMS default connections created from this connection factory. The password itself must be properly created and configured with other OC4J facilities.
clientID	String	No	The empty string	The administratively configured, fixed JMS clientID for connections created from this connection factory. If no clientID is specified, then the default is an empty string, which can also be programmatically overridden by client programs, according to the JMS specification. The clientID is used only for durable subscriptions on topics; its value does not matter for queue and nondurable topic operations.

Note: In [Table 3–5](#), the property `password` supports password indirection. For more information, refer to the *Oracle Application Server Containers for J2EE Security Guide*.

Examples

This section contains code samples that show connection factory configuration fragments.

The following code sample configures a connection factory of `jms/Cf`, a queue connection factory of `jms/Qcf`, and an XA topic connection factory of `jmx/xaTcf`.

```
<connection-factory location="jms/Cf">
</connection-factory>

<queue-connection-factory location="jms/Qcf">
</queue-connection-factory>

<xa-topic-connection-factory location="jmx/xaTcf"
  username="foo" password="bar" clientID="baz">
</xa-topic-connection-factory>
```

If you want to add a topic connection factory, you must use a unique name. For example, you could not use the same name as the connection factory (above) of `jms/Cf`. Thus, the following would be invalid.

```
<!-- Invalid: cannot reuse "location" -->
<topic-connection-factory location="jms/Cf">
</topic-connection-factory>
```

The following code sample shows queue and topic configuration fragments. This segment creates a queue `foo` and a topic `bar`.

```
<queue name="foo" location="jms/foo">
</queue>
```

```
<topic name="bar" location="jms/bar">
</topic>
```

Certain locations are reserved and cannot be redefined within the `jms.xml` configuration file. The following code sample shows that you cannot use the `jms/OC4jJmsExceptionQueue` when defining a queue location, because it is a reserved location.

```
<!-- Invalid: cannot use a reserved "location" -->
<queue name="fubar" location="jms/OC4jJmsExceptionQueue">
</queue>
```

When defining a persistence file for queues and topics, you can define the location and the filename. In addition, you can specify multiple persistence files, as long as the persistence filename is the same. Thus, the persistence file is written out to two locations for the same queue.

```
<queue name="foo" location="jms/persist" persistence-file="pers">
</queue>

<!-- OK: multiple persistence file specification ok if consistent -->
<queue name="foo" location="jms/file" persistence-file="pers">
</queue>

<!-- Invalid: multiple persistence file specifications should be consistent -->
<queue name="foo" location="jms/file1" persistence-file="notpers">
</queue>
```

Alternatively, you cannot have two objects writing out to the same persistence file. Each queue or topic must have its own persistence filename, even if the locations are different.

```
<topic name="demoTopic" location="jms/dada" persistence-file="/tmp/abcd">
</topic>

<!-- Invalid: cannot reuse persistence-file for multiple destinations -->
<topic name="demoTopic1" location="jms/dada1" persistence-file="/tmp/abcd">
</topic>
```

OracleAS JMS System Properties

OC4J JMS allows runtime configuration of the OC4J JMS server and JMS clients through JVM system properties. None of these properties affect basic JMS functionality. They pertain to OC4J JMS-specific features, extensions, and performance optimizations.

[Table 3–6](#) contains a brief summary of these administration properties.

Table 3–6 OC4J JMS Administration Properties

JVM System Property	Property Type	Default Value	Server/Client	Use
<code>oc4j.jms.serverPoll</code>	long	15000	JMS client	Interval (in milliseconds) that JMS connections ping the OC4J server and report communication exceptions to exception listeners.
<code>oc4j.jms.messagePoll</code>	long	1000	JMS client	Maximum interval (in milliseconds) that JMS asynchronous consumers wait before checking the OC4J JMS server for new messages.
<code>oc4j.jms.listenerAttempts</code>	int	5	JMS client	Number of listener delivery attempts before the message is declared undeliverable.
<code>oc4j.jms.maxOpenFiles</code>	int	64	OC4J server	Maximum number of open file descriptors (for persistence files) in the OC4J JMS server; relevant if the server is configured with more persistent <code>Destination</code> objects than the maximum number of open file descriptors allowed by the operating system.
<code>oc4j.jms.saveAllExpired</code>	boolean	false	OC4J server	Save all expired messages on all <code>Destination</code> objects (persistent, nonpersistent, and temporary) to the OC4J JMS exception queue.
<code>oc4j.jms.socketBufsize</code>	int	64 * 1024	JMS client	When using TCP/IP sockets for client-server communication, use the specified buffer size for the socket input/output streams. A minimum buffer size of 8 KB is enforced. The larger the size of messages being transferred between the client and server, the larger the buffer size should be to provide reasonable performance.
<code>oc4j.jms.debug</code>	boolean	false	JMS client	If true, enable tracing of <code>NORMAL</code> events in JMS clients and the OC4J JMS server. All log events (<code>NORMAL</code> , <code>WARNING</code> , <code>ERROR</code> , and <code>CRITICAL</code>) are sent to both <code>stderr</code> and, when possible, either <code>J2EE_HOME/log/server.log</code> or <code>J2EE_HOME/log/jms.log</code> . Setting this property to true typically generates large amounts of tracing information.
<code>oc4j.jms.noDms</code>	boolean	false	JMS client	If true, disable instrumentation.

Table 3–6 (Cont.) OC4J JMS Administration Properties

JVM System Property	Property Type	Default Value	Server/Client	Use
<code>oc4j.jms.forceRecovery</code>	boolean	false	OC4J server	If <code>true</code> , forcibly recover corrupted persistence files. By default, the OC4J JMS server does not perform recovery of a persistence file if its header is corrupted (because this condition is, in general, indistinguishable from configuration errors). Forcible recovery allows the OC4J JMS server almost always to start up correctly and make persistence files and <code>Destination</code> objects available for use.
<code>oc4j.jms.pagingThreshold</code>	double	1.0	OC4J server	Represents the memory usage fraction above which the OracleAS JMS server begins to consider message bodies for paging. This value is an estimate of the fraction of memory in use by the JVM. This value can range from 0.0 (the program uses no memory at all) to 1.0 (the program is using all available memory). See " Message Paging " on page 3-15 for more information.
<code>oc4j.jms.usePersistenceLockFiles</code>	boolean	true	OC4J server	Controls whether lock files should be used to protect OracleAS JMS persistence files from being overwritten by more than one OC4J process. By default, lock files are used to protect against accidental overwrite by more than one OC4J process. But this requires users to manually remove lock files when OC4J terminates abnormally. Setting this system property to <code>false</code> does not create lock files for persistent destinations. Set this property to <code>false</code> only if you can guarantee that only one active process accesses each persistence file. Set when starting the OC4J server. It remains in effect for all JMS clients until shutdown.

Resource Providers

OC4J provides a `ResourceProvider` interface to transparently plug in JMS providers.

The `ResourceProvider` interface of OC4J allows EJBs, servlets, and OC4J clients to access many different JMS providers. The resources are available under `java:comp/resource/`. Oracle JMS is accessed using the `ResourceProvider` interface. See "[Oracle JMS](#)" on page 3-24 for more information on Oracle JMS.

Configuring a Custom Resource Provider

You can configure a custom resource provider in one of these ways:

- If this is the resource provider for all applications (global), then configure the `global.application.xml` file.
- If this is the resource provider for a single application (local), then configure the `orion-application.xml` file of the application.

To add a custom resource provider, add the following code to the chosen XML file (as listed above):

```
<resource-provider class="providerClassName" name="JNDIName">
```

```

    <description>description </description>
    <property name="name" value="value" />
</resource-provider>

```

For the `<resource-provider>` attributes, configure the following:

- `class`—The name of the resource provider class.
- `name`—A name by which to identify the resource provider. This name is used in finding the resource provider in the application JNDI as:

```
java:comp/resource/JNDIname/
```

The subelements of the `<resource-provider>` are configured as follows:

- `description` subelement—A description of the specific resource provider.
- `property` subelement—The `name` and `value` attributes are used to identify parameters provided to the resource provider. The `name` attribute identifies the name of the parameter, and its value is provided in the `value` attribute.

When retrieving the resource provider, use the following syntax in the JNDI lookup:

```
java:comp/resource/JNDIname/resourceName
```

where `JNDIname` is the name of the resource provider (as given in the `name` attribute of the `<resource-provider>` element) and `resourceName` is the resource name, which is defined in the application implementation. See ["Using OJMS as a Resource Provider"](#) on page 3-24 for an example of Oracle JMS defined as a resource provider.

Oracle JMS

Oracle JMS (OJMS) is the JMS interface to the Oracle Database Streams Advanced Queuing (AQ) feature in the Oracle database. OJMS implements the JMS 1.0.2b specification and is compatible with the J2EE 1.3 specification. OJMS access in OC4J occurs through the resource provider interface. For more information about AQ and OJMS, see the *Oracle9i Application Developer's Guide—Advanced Queuing for Release 2 (9.2)*.

The following sections describe Oracle JMS:

- [Using OJMS as a Resource Provider](#)
- [Using OJMS with Oracle Application Server and the Oracle Database](#)

Using OJMS as a Resource Provider

To access OJMS queues, do the following:

1. Install and configure OJMS on the database. See ["Install and Configure the JMS Provider"](#) on page 3-25.
2. On the database, create an RDBMS user—which the JMS application will connect to the back-end database—and assign privileges. The user must have the necessary privileges to perform OJMS operations. OJMS allows any database user to access queues in any schema, provided that the user has the appropriate access privileges. See ["Create User and Assign Privileges"](#) on page 3-25.
3. Create the JMS Destination objects in OJMS. See ["Create JMS Destination Objects"](#) on page 3-25.

4. In the OC4J XML configuration, define an OJMS resource provider in the `<resource-provider>` element with information about the back-end database. Create data sources or LDAP directory entries, if needed. See "[Define the OJMS Resource Provider](#)" on page 3-27.
5. Access the resource in your implementation through a JNDI lookup. See "[Access the OJMS Resources](#)" on page 3-29.

Install and Configure the JMS Provider

You or your DBA must install OJMS according to the *Oracle9i Application Developer's Guide—Advanced Queuing for Release 2 (9.2)* and generic database manuals. After you have installed and configured this JMS provider, you must apply additional configuration. This includes the following:

1. You or your DBA must create an RDBMS user through which the JMS client connects to the database. Grant this user appropriate access privileges to perform OJMS operations. See "[Create User and Assign Privileges](#)" on page 3-25.
2. You or your DBA must create the tables and queues to support the JMS Destination objects. See "[Create JMS Destination Objects](#)" on page 3-25.

Note: The following sections use SQL for creating queues, topics, their tables, and assigning privileges within the JMS demo on the OC4J sample code page on the OTN Web site at

<http://www.oracle.com/technology/tech/java/oc4j/demos/index.html>

Create User and Assign Privileges

Create an RDBMS user through which the JMS client connects to the database. Grant access privileges to this user to perform OJMS operations. The privileges that you need depend on what functionality you are requesting. Refer to the *Oracle9i Application Developer's Guide—Advanced Queuing for Release 2 (9.2)* for more information on privileges necessary for each type of function.

The following example creates `jmsuser`, which must be created within its own schema, with privileges required for OJMS operations. You must be a SYS DBA to execute these statements.

```
DROP USER jmsuser CASCADE ;

GRANT connect,resource,AQ_ADMINISTRATOR_ROLE TO jmsuser
  IDENTIFIED BY jmsuser ;
GRANT execute ON sys.dbms_aqadm TO jmsuser;
GRANT execute ON sys.dbms_aq TO jmsuser;
GRANT execute ON sys.dbms_aqin TO jmsuser;
GRANT execute ON sys.dbms_aqjms TO jmsuser;

connect jmsuser/jmsuser;
```

You may need to grant other privileges, such as two-phase commit or system administration privileges, based on what the user needs. See [Chapter 7, "Java Transaction API"](#), for information on two-phase commit privileges.

Create JMS Destination Objects

Each JMS provider requires its own method for creating the JMS Destination object. Refer to the *Oracle9i Application Developer's Guide—Advanced Queuing for Release 2 (9.2)*

for more information on the DBMS_AQADM packages and OJMS messages types. For our example, OJMS requires the following methods:

Note: The SQL for creating the tables for the OJMS example is included in the JMS example available on the OC4J sample code page on the OTN Web site at

<http://www.oracle.com/technology/tech/java/oc4j/demos/index.html>

1. Create the tables that handle the JMS Destination (queue or topic).

In OJMS, both topics and queues use a queue table. The JMS example creates a single table, `demoTestQTab`, for a queue.

To create the queue table, execute the following SQL:

```
DBMS_AQADM.CREATE_QUEUE_TABLE(
    Queue_table      => 'demoTestQTab',
    Queue_payload_type => 'SYS.AQ$_JMS_MESSAGE',
    sort_list        => 'PRIORITY,ENQ_TIME',
    multiple_consumers => false,
    compatible       => '8.1.5');
```

The `multiple_consumers` parameter specifies whether there are multiple consumers. Thus, is always `false` for a queue and `true` for a topic.

2. Create the JMS Destination. If you are creating a topic, you must add each subscriber for the topic. The JMS example requires a single queue—`demoQueue`.

The following command creates a queue called `demoQueue` within the queue table `demoTestQTab`. After creation, the queue is started.

```
DBMS_AQADM.CREATE_QUEUE(
    Queue_name      => 'demoQueue',
    Queue_table     => 'demoTestQTab');

DBMS_AQADM.START_QUEUE(
    queue_name      => 'demoQueue');
```

If you want to add a topic, then the following example shows how you can create a topic called `demoTopic` within the topic table `demoTestTTab`. After creation, two durable subscribers are added to the topic. Finally, the topic is started, and a user is granted a privilege to it.

Note: Oracle AQ uses the `DBMS_AQADM.CREATE_QUEUE` method to create both queues and topics.

```
DBMS_AQADM.CREATE_QUEUE_TABLE(
    Queue_table      => 'demoTestTTab',
    Queue_payload_type => 'SYS.AQ$_JMS_MESSAGE',
    multiple_consumers => true,
    compatible       => '8.1.5');
DBMS_AQADM.CREATE_QUEUE('demoTopic', 'demoTestTTab');
DBMS_AQADM.ADD_SUBSCRIBER('demoTopic',
    sys.aq$_agent('MDSUB', null, null));
DBMS_AQADM.ADD_SUBSCRIBER('demoTopic',
    sys.aq$_agent('MDSUB2', null, null));
```

```
DBMS_AQADM.START_QUEUE('demoTopic');
```

Note: The names defined here must be the same names used to define the queue or topic in the deployment descriptors of the application.

Define the OJMS Resource Provider

You can define the OJMS resource provider either through the Oracle Enterprise Manager 10g or by hand-editing the XML files, as described in the following sections:

- [Configure the OJMS Provider Through the Oracle Enterprise Manager 10g](#)
- [Configure the OJMS Provider in the OC4J XML Files](#)

Configure the OJMS Provider Through the Oracle Enterprise Manager 10g Use Application Server Control in the JMS section to configure the OJMS provider. To add an OJMS provider, select JMS Providers under the Application Defaults column on the Administration page. This brings you to the following page:

Click the **Add new JMS Provider** button to configure each JMS provider.

You can configure either OJMS or a third-party JMS provider. OracleAS JMS is always provided and preconfigured, except for the topics and queues, with the OC4J installation.

Note: This discussion also includes the directions for configuring third-party JMS providers, because both OJMS and third-party providers are configured in the same manner.

After you choose the type of JMS provider, you must specify the following:

- OJMS: Specify the data source name and JNDI location for the database in which OJMS is installed and configured.
- Third-party JMS provider: Specify the name, JNDI initial context factory class, and JNDI URL for the third-party provider. To add JNDI properties for this JMS provider, such as `java.naming.factory.initial` and `java.naming.provider.url`, click **Add a property**. A row is added in which you can specify the name for each JNDI property and its value.

Doing this configures only the providers; it does not configure the `Destination` objects (topic, queue, and subscription).

To configure a JMS provider only for a specific application, select the application from the Applications page, scroll down to the Resources column, and select JMS Providers. The screens that appear are the same as for the default JMS provider.

Configure the OJMS Provider in the OC4J XML Files Configure the OJMS provider within the `<resource-provider>` element.

- If this is to be the JMS provider for all applications (global), configure the `global-application.xml` file.
- If this is to be the JMS provider for a single application (local), configure the `orion-application.xml` file of the application.

The following code sample shows how to configure the JMS provider using XML syntax for OJMS.

```
<resource-provider class="oracle.jms.OjmsContext" name="ojmsdemo">
  <description> OJMS/AQ </description>
  <property name="datasource" value="jdbc/emulatedDS"></property>
</resource-provider>
```

where the attributes of the `<resource-provider>` element contain the following:

- **class attribute**—The OJMS provider is implemented by the `oracle.jms.OjmsContext` class, which is configured in the `class` attribute.
- **name attribute**—The name of the OJMS resource provider is `ojmsdemo`.

In addition, the `name/value` attributes of the `<property>` element identify the data source used by OJMS. The topic or queue connects to this data source to access the tables and queues that facilitate the messaging. In this example, a data source is identified as `jdbc/emulatedDS`.

How you configure the attributes of the `<property>` element in the resource provider configuration depends on where your application is running. With OJMS and accessing AQ in the database, the resource provider must be configured using either a data sources property or a URL property, as discussed in the following sections:

- [Configuring the Resource Provider with a Data Sources Property](#)
- [Configuring the Resource Provider with a URL Property](#)

Configuring the Resource Provider with a Data Sources Property

Use a data source when the application runs within OC4J. To use a data source, first you must configure it within the `data-sources.xml` file in which the OJMS provider is installed. The JMS topics and queues use database tables and queues to facilitate messaging. The type of data source you use depends on the functionality you want.

Note: For no transactions or single-phase transactions, you can use either an emulated or nonemulated data source. For two-phase commit transaction support, you can use only a nonemulated data source. See the JTA chapter for more information.

Example 3–3 Emulated Data Source with Thin JDBC Driver

The following example contains an emulated data source that uses the Thin JDBC driver. To support a two-phase commit transaction, use a nonemulated data source. For differences between emulated and nonemulated data sources, see ["Defining Data Sources"](#) on page 4-6.

The example is displayed in the format of an XML definition; see the *Oracle Application Server Containers for J2EE User's Guide* for directions on adding a new data source to the configuration through Oracle Enterprise Manager 10g.

```
<data-source
  class="com.evermind.sql.DriverManagerDataSource"
  name="OracleDS"
  location="jdbc/emulatedOracleCoreDS"
  xa-location="jdbc/xa/emulatedOracleXADS"
  ejb-location="jdbc/emulatedDS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="jmsuser"
  password="jmsuser"
  url="jdbc:oracle:thin:@//localhost:5521/oracle.regress.rdbms.dev.us.oracle.com"
/>
```

Customize this data source to match your environment. For example, substitute the host name, port, and SID of your database for `myhost:1521:orcl`.

Note: Instead of providing the password in the clear, you can use password indirection. For details, see the *Oracle Application Server Containers for J2EE Services Guide*.

Next, configure the resource provider using the data source name. The following is an example of how to configure the resource provider for OJMS using a data source of `jdbc/emulatedDS`.

```
<resource-provider class="oracle.jms.OjmsContext" name="ojmsdemo">
  <description> OJMS/AQ </description>
  <property name="datasource" value="jdbc/emulatedDS"></property>
</resource-provider>
```

For details on configuring data sources, see ["Defining Data Sources"](#) on page 4-6.

Configuring the Resource Provider with a URL Property

In this release, the data source is not serializable. Thus, application clients must use a URL definition to access OJMS resources. When the application is a standalone client (that is, when it runs outside of OC4J), configure the `<resource-provider>` element with a URL property that has the URL of the database where OJMS is installed and, if necessary, provides the username and password for that database. The following demonstrates a URL configuration:

```
<resource-provider class="oracle.jms.OjmsContext" name="ojmsdemo">
  <description> OJMS/AQ </description>
  <property name="url"
value="jdbc:oracle:thin:@//localhost:5521/oracle.regress.rdbms.dev.us.oracle.com">
  </property>
  <property name="username" value="user"></property>
  <property name="password" value="passwd"></property>
```

Access the OJMS Resources

The steps for accessing OJMS resources are the same as for OracleAS JMS resources, as listed in ["Steps for Sending a Message"](#) on page 3-5. The only difference is the name of the resource provided in the JNDI lookup.

- The OJMS syntax for the connection factory is `"java:comp/resource" + JMS provider name + "TopicConnectionFactories"` or `"QueueConnectionFactories"` + a user defined name. The user-defined name can be anything and does not match any other configuration. The `xxxConnectionFactories` details what type of factory is being defined. For this example, the JMS provider name is defined in the `<resource-provider>` element as `ojmsdemo`.
 - For a queue connection factory: Because the JMS provider name is `ojmsdemo` and you decide to use a name of `myQCF`, the connection factory name is:


```
java:comp/resource/ojmsdemo/ QueueConnectionFactories/myQCF
```
 - For a topic connection factory: Because the JMS provider name is `ojmsdemo` and you decide to use a name of `myTCF`, the connection factory name is:


```
java:comp/resource/ojmsdemo/
TopicConnectionFactories/myTCF
```

The user defined names, as shown above by `myQCF` and `myTCF`, are not used for anything else in your logic. So, any name can be chosen.

- The OJMS syntax for any Destination is "java:comp/resource" + JMS provider name + "Topics" or "Queues" + Destination name. The Topic or Queue details what type of Destination is being defined. The Destination name is the actual queue or topic name defined in the database.

For this example, the JMS provider name is defined in the `<resource-provider>` element as `ojmsdemo`. In the database, the queue name is `demoQueue`.

- For a queue: If the JMS provider name is `ojmsdemo` and the queue name is `demoQueue`, then the JNDI name for the topic is:
`java:comp/resource/ojmsdemo/Queues/demoQueue`
- For a topic: If the JMS provider name is `ojmsdemo` and the topic name is `demoTopic`, then the JNDI name for the topic is:
`java:comp/resource/ojmsdemo/Topics/demoTopic`

[Example 3-4](#) demonstrates the steps for sending a JMS message; [Example 3-5](#) demonstrates the steps for receiving a JMS message. For the complete example, download the JMS example used in this chapter from the OC4J sample code page on the OTN Web site at

<http://www.oracle.com/technology/tech/java/oc4j/demos/index.html>

Note: For simplicity, most of the error handling is removed in [Example 3-4](#) and [Example 3-5](#). To see the error processing, see the sample code available on the OTN Web site.

Example 3-4 OJMS Client That Sends Messages to an OJMS Queue

The `dosend` method, shown in the following example, sets up a queue to send messages. After creating the queue sender, this example sends out several messages. The steps necessary for setting up the queue and sending out the message are summarized in "[Steps for Sending a Message](#)" on page 3-5.

```
public static void dosend(int nmsgs)
{
    // 1a. Retrieve the queue connection factory
    QueueConnectionFactory qcf = (QueueConnectionFactory)
        ctx.lookup(
            "java:comp/resource/ojmsdemo/QueueConnectionFactories/myQCF");
    // 1b. Retrieve the queue
    Queue q = (Queue)
        ctx.lookup("java:comp/resource/ojmsdemo/Queues/demoQueue");

    // 2. Create the JMS connection
    QueueConnection qc = qcf.createQueueConnection();
    // 3. Start the queue connection.
    qc.start();
    // 4. Create the JMS session over the JMS connection
    QueueSession qs = qc.createQueueSession(false,
        Session.AUTO_ACKNOWLEDGE);
    // Create a sender on the JMS session to send messages.
    QueueSender snd = qs.createSender(q);

    // Send out messages...
    for (int i = 0; i < nmsgs; ++i)
```



```

    {
        //Create the message using the createMessage method
        // of the JMS session
        Message msg = qs.createMessage();
        // Send the message out over the sender (snd) using the send method
        snd.send(msg);
        System.out.println("msg:" + " id=" + msg.getJMSMessageID());
    }

    // Close the sender, the JMS session and the JMS connection.
    snd.close();
    qs.close();
    qc.close();
}

```

Example 3-5 OJMS Client That Receives Messages Off a Queue

The `dorcv` method, shown in the following example, sets up a queue to receive messages off it. After creating the queue receiver, it loops to receive all messages off the queue and compares it to the number of expected messages. The steps necessary for setting up the queue and receiving messages are summarized in ["Steps for Sending a Message"](#) on page 3-5.

```

public static void dorcv(int nmsgs)
{
    Context ctx = new InitialContext();

    // 1a. Retrieve the queue connection factory
    QueueConnectionFactory qcf = (QueueConnectionFactory) ctx.lookup(
        "java:comp/resource/ojmsdemo/QueueConnectionFactories/myQCF");
    // 1b. Retrieve the queue
    Queue q = (Queue)
        ctx.lookup("java:comp/resource/ojmsdemo/Queues/demoQueue");

    // 2. Create the JMS connection
    QueueConnection qc = qcf.createQueueConnection();
    // 3. Start the queue connection.
    qc.start();
    // 4. Create the JMS session over the JMS connection
    QueueSession qs = qc.createQueueSession(false,
        Session.AUTO_ACKNOWLEDGE);
    // Create a receiver, as we are receiving off of the queue.
    QueueReceiver rcv = qs.createReceiver(q);

    // Receive the messages
    int count = 0;
    while (true)
    {
        Message msg = rcv.receiveNoWait();
        System.out.println("msg:" + " id=" + msg.getJMSMessageID());
        ++count;
    }

    if (nmsgs != count)
    {
        System.out.println("expected: " + nmsgs + " found: " + count);
    }

    // Close the receiver, the JMS session and the JMS connection.
    rcv.close();
}

```

```

        qs.close();
        qc.close();
    }

```

Using OJMS with Oracle Application Server and the Oracle Database

This section addresses common issues encountered by users of OJMS (AQ/JMS) with Oracle Application Server.

- [Error When Copying aqapi.jar](#)
- [OJMS Certification Matrix](#)

Error When Copying aqapi.jar

A common error seen when using OJMS with the Oracle Application Server is:

```
PLS-00306 "wrong number or types of arguments"
```

If you receive this message, then the `aqapi.jar` file being used in Oracle Application Server is not compatible with the version of the Oracle database being used for AQ. A common mistake is to copy the `aqapi.jar` file from the Oracle database installation into the Oracle Application Server installation, or from the Oracle Application Server installation into the Oracle database installation, under the false assumption that they are interchangeable. The confusion is because the Oracle Application Server and the Oracle database both ship the OJMS client JAR file. Do not copy this file. Use the matrix in [Table 3-7](#) to find the correct version of the database and Oracle Application Server, then use the `aqapi.jar` file that comes with the Oracle Application Server.

In an Oracle Application Server installation, the OJMS client JAR file can be found at `ORACLE_HOME/rdbms/jlib/aqapi.jar` and should be included in the CLASSPATH.

OJMS Certification Matrix

[Table 3-7](#) summarizes which version of the Oracle database to use with the Oracle Application Server when the OJMS client is running in OC4J. An X indicates that the Oracle database version and the Oracle Application Server version that intersect at that cell are certified to work together. If the intersection has no X, then the corresponding version of the Oracle database and Oracle Application Server should not be used together.

Note: This is not a certification matrix for Oracle Application Server and the Oracle database in general. It is only for OJMS when used in the Oracle Application Server.

Table 3-7 OJMS Certification Matrix

OracleAS / Oracle database	v9.0.1	v9.0.1.3	v9.0.1.4	v9.2.0.1	v9.2.0.2+	v10.1.0+
9.0.2	X	X		X		
9.0.3			X			X
9.0.4			X		X	
9.0.4.1						X
10.1.2			X		X	X

Map Logical Names in Resource References to JNDI Names

The client sends and receives messages through a `JMS Destination` object. The client can retrieve the `JMS Destination` object and connection factory either through using its explicit name or by a logical name. The examples in "Oracle Application Server JMS" on page 3-2 and "Oracle JMS" on page 3-24 use explicit names within the JNDI lookup calls. This section describes how you can use logical names in your client application, thereby limiting the JNDI names for the JMS provider within the OC4J-specific deployment descriptors. With this indirection, you can make your client implementation generic for any JMS provider.

If you want to use a logical name in your client application code, then define the logical name in one of the following XML files:

- A standalone Java client—in the `application-client.xml` file
- An EJB that acts as a client—the `ejb-jar.xml` file
- For JSPs and servlets that act as clients—the `web.xml` file

Map the logical name to the actual name of the topic or queue name in the OC4J deployment descriptors.

Create Logical Names

You can create logical names for the connection factory and `Destination` objects, as follows:

- Specify the logical name for the connection factory in the client's XML deployment descriptor file within a `<resource-ref>` element.
 - Specify the logical name for the connection factory in the `<res-ref-name>` element.
 - Specify the connection factory class type in the `<res-type>` element as either `javax.jms.QueueConnectionFactory` or `javax.jms.TopicConnectionFactory`.
 - Specify the authentication responsibility (`Container` or `Bean`) in the `<res-auth>` element.
 - Specify the sharing scope (`Shareable` or `Unshareable`) in the `<res-sharing-scope>` element.
- Specify the logical name for the `JMS Destination`, the topic or queue, within a `<resource-env-ref>` element.
 - Specify the logical name for the topic or queue in the `<resource-env-ref-name>` element.
 - Specify the `Destination` class type in the `<resource-env-ref-type>` element as either `javax.jms.Queue` or `javax.jms.Topic`.

Example

The following example illustrates how to specify logical names for a queue.

```
<resource-ref>
  <res-ref-name>myQCF</res-ref-name>
  <res-type>javax.jms.QueueConnectionFactory</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
<resource-env-ref>
```

```
<resource-env-ref-name>myQueue</resource-env-ref-name>
<resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
</resource-env-ref>
```

Map Logical Names to Actual Names

After the logical names are created, you map the logical names to actual names in the OC4J deployment descriptors. The actual names, or JNDI names, are different in OracleAS JMS than those in OJMS. However, the mapping is defined in one of the following files:

- For a standalone Java client, the mapping is defined in the `orion-application-client.xml` file.
- For an EJB acting as a client, the mapping is defined in the `orion-ejb-jar.xml` file.
- For JSPs and servlets acting as a client, the mapping is defined in the `orion-web.xml` file.

The logical names in the client's deployment descriptor are mapped as follows:

- The logical name for the connection factory defined in the `<resource-ref>` element is mapped to its JNDI name in the `<resource-ref-mapping>` element.
- The logical name for the JMS Destination defined in the `<resource-env-ref>` element is mapped to its JNDI name in the `<resource-env-ref-mapping>` element.

See the following sections for how the mapping occurs for both OracleAS JMS and OJMS and how clients use this naming convention:

- [JNDI Naming for OracleAS JMS](#)
- [JNDI Naming for OJMS](#)
- [JNDI Naming Property Setup for Java Application Clients](#)
- [Client Sends JMS Message Using Logical Names](#)

JNDI Naming for OracleAS JMS

The JNDI names for the OracleAS JMS Destination and connection factory are defined within the `jms.xml` file. As [Example 3-1](#) shows, the JNDI names for the queue and the queue connection factory are as follows:

- The JNDI name for the topic is `"jms/demoQueue."`
- The JNDI name for the topic connection factory is `"jms/QueueConnectionFactory."`

Prepend both of these names with `"java:comp/env/"`. The following example shows the mapping in the `orion-ejb-jar.xml` file:

```
<resource-ref-mapping
  name="myQCF"
  location="java:comp/env/jms/QueueConnectionFactory">
</resource-ref-mapping>

<resource-env-ref-mapping
  name="myQueue"
  location="java:comp/env/jms/demoQueue">
</resource-env-ref-mapping>
```

JNDI Naming for OJMS

The JNDI naming for OJMS Destination and connection factory objects is the same name that was specified in the `orion-ejb-jar.xml` file as described in ["Access the OJMS Resources"](#) on page 3-29.

The following example maps the logical names for the connection factory and queue to their actual JNDI names. Specifically, the queue defined logically as `"myQueue"` in the `application-client.xml` file is mapped to its JNDI name of `"java:comp/resource/ojmsdemo/Queues/demoQueue."`

```
<resource-ref-mapping
  name="myQCF"
  location="java:comp/resource/ojmsdemo/QueueConnectionFactories/myQF">
</resource-ref-mapping>

<resource-env-ref-mapping
  name="myQueue"
  location="java:comp/resource/ojmsdemo/Queues/demoQueue">
</resource-env-ref-mapping>
```

JNDI Naming Property Setup for Java Application Clients

In the Oracle Application Server, a Java application client accesses a JMS Destination object by providing the following code in the JNDI properties:

```
java.naming.factory.initial=
  com.evermind.server.ApplicationClientInitialContextFactory
java.naming.provider.url=opmn:ormi://$HOST:$OPMN_REQUEST_PORT:$OC4J_INSTANCE/
java.naming.security.principal=admin
java.naming.security.credentials=welcome
```

You must:

- Use the `ApplicationClientInitialContextFactory` as your initial context factory object.
- Supply the OPMN host and port and the OC4J instance in the provider URL.

In an OC4J standalone environment, a Java application client accesses a JMS Destination object by providing the following code in the JNDI properties:

```
java.naming.factory.initial=
  com.evermind.server.ApplicationClientInitialContextFactory
java.naming.provider.url=ormi://myhost/
java.naming.security.principal=admin
java.naming.security.credentials=welcome
```

You must:

- Use the `ApplicationClientInitialContextFactory` as your initial context factory object.
- Supply the standalone OC4J host and port in the provider URL.

Client Sends JMS Message Using Logical Names

After the resources have been defined and the JNDI properties configured, the client must perform the following steps to send a JMS message. These steps summarize the procedure that [Example 3–6](#) shows in detail.

1. Retrieve both the configured JMS Destination and its connection factory using a JNDI lookup.
2. Create a connection from the connection factory. If you are receiving messages for a queue, start the connection.
3. Create a session over the connection.
4. Providing the retrieved JMS Destination, create a sender for a queue, or a publisher for a topic.
5. Create the message.
6. Send the message using either the queue sender or the topic publisher.
7. Close the queue session.
8. Close the connection for either JMS Destination types.

Example 3-6 JSP Client Sends a Message to a Topic

The method of sending a message to a topic is similar to that of sending a message to a queue. Instead of creating a queue, you create a topic. Instead of creating a sender, you create publishers.

The following JSP client code sends a message to a topic. The code uses logical names, which are mapped in the OC4J deployment descriptor. The numbered comments in the example correspond to the steps summarized at the beginning of this section.

```
<%@ page import="javax.jms.*, javax.naming.*, java.util.*" %>
<%

//1a. Look up the topic.
jndiContext = new InitialContext();
topic = (Topic)jndiContext.lookup("demoTopic");

//1b. Look up the Connection factory.
topicConnectionFactory = (TopicConnectionFactory)
    jndiContext.lookup("myTCF");

//2 & 3. Retrieve a connection and a session on top of the connection.
topicConnection = topicConnectionFactory.createTopicConnection();
topicSession = topicConnection.createTopicSession(true,
    Session.AUTO_ACKNOWLEDGE);

//4. Create the publisher for any messages destined for the topic.
topicPublisher = topicSession.createPublisher(topic);

//5 & 6. Create and send out the message.
for (int ii = 0; ii < numMsgs; ii++)
{
    message = topicSession.createBytesMessage();
    String sndstr = "1::This is message " + (ii + 1) + " " + item;
    byte[] msgdata = sndstr.getBytes();
    message.writeBytes(msgdata);

    topicPublisher.publish(message);
    System.out.println("--->Sent message: " + sndstr);
}

//7 & 8. Close publisher, session, and connection for topic.
topicPublisher.close();
topicSession.close();
```

```
topicConnection.close();
%>
```

Third-Party JMS Providers

This section discusses the following third-party JMS providers and how they integrate with OC4J using the resource provider interface:

- [Using WebSphere MQ as a Resource Provider](#)
- [Using SonicMQ as a Resource Provider](#)
- [Using SwiftMQ as a Resource Provider](#)

Here are the operations that the resource provider interface supports:

- Look up queue and topic with


```
java:comp/resource/providerName/resourceName
```
- Send a message in EJB
- Receive a message synchronously in EJB

Note: Oracle supports only single-phase commit semantics for resource providers other than OJMS.

The context-scanning resource provider class is a generic resource provider class shipped with OCJ for use with third-party message providers.

Using WebSphere MQ as a Resource Provider

WebSphere MQ is an IBM messaging provider. This example demonstrates how to make WebSphere MQ the default resource provider for JMS connections. The WebSphere MQ resources are available in OC4J under `java:comp/resource/MQSeries/`.

Configuring WebSphere MQ

To configure WebSphere MQ, perform the following steps:

1. Install and configure WebSphere MQ on your system, then verify the installation by running any examples or tools supplied by the vendor. (See the documentation supplied with your software for instructions.)
2. Configure the resource provider. You can use either of two methods to configure the resource provider: Use Oracle Enterprise Manager 10g (as shown in "[Define the OJMS Resource Provider](#)" on page 3-27), or use the `<resource-provider>` element in `orion-application.xml`. Use either method to add WebSphere MQ as a custom resource provider. The following example demonstrates using the `<resource-provider>` element to configure WebSphere MQ. You can use the same information to configure using Oracle Enterprise Manager 10g.

```
<resource-provider
  class="com.evermind.server.deployment.ContextScanningResourceProvider"
  name="MQSeries">
<description> MQSeries resource provider </description>
<property
  name="java.naming.factory.initial"
  value="com.sun.jndi.fscontext.ReffFSContextFactory">
```

```
</property>
<property
  name="java.naming.provider.url"
  value="file:/var/mqm/JNDI-Directory">
</property>
</resource-provider>
```

3. Add the following WebSphere MQ JMS client JAR files to `J2EE_HOME/lib`:

```
com.ibm.mq.jar
com.ibm.mqbind.jar
com.ibm.mqjms.jar
mqji.properties
```

4. Add the file system JNDI JAR files `fscontext.jar` and `providerutil.jar` to `J2EE_HOME/lib`.

Using SonicMQ as a Resource Provider

SonicMQ is a messaging provider from Sonic Software Corporation. The resource provider interface furnishes support for plugging in third-party JMS implementations. This example describes how to make SonicMQ the default resource provider for JMS connections. The SonicMQ resources are available in OC4J under `java:comp/resource/SonicMQ`.

Note: SonicMQ broker does not embed a JNDI service. Instead, it relies on an external directory server to register the administered objects. Administered objects, such as queues, are created by an administrator—either using SonicMQ Explorer or programmatically—using the Sonic Management API. Oracle registers the administered objects from SonicMQ Explorer using the file system JNDI.

Configuring SonicMQ

To configure SonicMQ, perform the following steps:

1. Install and configure SonicMQ on your system, then verify the installation by running any examples or tools supplied by the vendor. (See the documentation supplied with your software for instructions.)
2. Configure the resource provider. You can use either of two methods to configure the resource provider: Use Oracle Enterprise Manager 10g (as shown in ["Define the OJMS Resource Provider"](#) on page 3-27), or use the `<resource-provider>` element in `orion-application.xml`. Use either method to add SonicMQ as a custom resource provider. The following example demonstrates using the `<resource-provider>` element to configure SonicMQ. You can use the same information to configure using Oracle Enterprise Manager 10g.

```
<resource-provider
  class="com.evermind.server.deployment.ContextScanningResourceProvider"
  name="SonicJMS">
  <description>
    SonicJMS resource provider.
  </description>
  <property name="java.naming.factory.initial"
    value="com.sun.jndi.fscontext.RefFSContextFactory">
  <property name="java.naming.provider.url"
    value="file:/private/jndi-directory/">
```



```
</resource-provider>
```

3. Add the following SonicMQ JMS client JAR files to `J2EE_HOME/lib`:

```
Sonic_client.jar
Sonic_XA.jar
```

Using SwiftMQ as a Resource Provider

SwiftMQ is a messaging provider from IIT Software. This example describes how to make SwiftMQ the default resource provider for JMS connections. The SwiftMQ resources are available in OC4J under `java:comp/resource/SwiftMQ`.

Configuring SwiftMQ

To configure SwiftMQ, perform the following steps:

1. Install and configure SwiftMQ on your system, then verify the installation by running any examples or tools supplied by the vendor. (See the documentation provided with your software for instructions.)
2. Configure the resource provider. You can use either of two methods to configure the resource provider: Use Oracle Enterprise Manager 10g (as shown in "[Define the OJMS Resource Provider](#)" on page 3-27), or use the `<resource-provider>` element in `orion-application.xml`. Use either method to add SwiftMQ as a custom resource provider. The following example demonstrates using the `<resource-provider>` element to configure SwiftMQ. You can use the same information to configure using Oracle Enterprise Manager 10g.

```
<resource-provider
  class="com.evermind.server.deployment.ContextScanningResourceProvider"
  name="SwiftMQ">
  <description>
    SwiftMQ resource provider.
  </description>
  <property name="java.naming.factory.initial"
    value="com.swiftmq.jndi.InitialContextFactoryImpl">
  <property name="java.naming.provider.url"
    value="smqp://localhost:4001">
</resource-provider>
```

3. Add the following SwiftMQ JMS client JAR files to `J2EE_HOME/lib`:

```
swiftmq.jar
```

Using Message-Driven Beans

See the Message-Driven Beans (MDB) chapter of the *Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide* for details on deploying an MDB that accesses OracleAS JMS or OJMS.

Note: The message-driven bean (MDB) transaction timeout, as defined in the transaction-timeout attribute in the orion-ejb-jar.xml file, is an optional parameter. This attribute controls the transaction timeout interval (in seconds) for any container-managed transactional MDB that uses Oracle JMS. The default is one day (86,400 seconds). The MDB transaction-timeout attribute applies only to CMT MDBs that use Oracle JMS as the JMS provider. This attribute setting has no effect on BMT MDBs or any MDBs that use OC4J JMS. (bug 3079322)

- JMS behavior with Oracle Application Server — If the transaction has not completed in this time frame, then the transaction is rolled back and the message is redelivered to the Destination object. After Oracle JMS attempts to redeliver the message (the default is five attempts), the message is moved to the exception queue. For more information, refer to the *Oracle9i Application Developer's Guide—Advanced Queuing for Release 2 (9.2)*.
- JMS behavior with OC4J — The transaction-timeout setting does not work for CMT MDBs that use OC4J JMS. The timeout is always one day and cannot be modified. When the timeout occurs, OC4J JMS redelivers the message indefinitely, until the delivery is successful. You cannot set a retry limit.

In addition, the global transaction-timeout attribute defined in the server.xml file does not have any effect on MDBs.

High Availability and Clustering for JMS

A highly available JMS server provides a guarantee that JMS requests will be serviced with no interruptions because of software or hardware failures. One way to achieve high availability is through fail-over; if one instance of the server fails, then a combination of software, hardware, and infrastructure mechanisms causes another instance of the server to take over the servicing of requests.

Table 3–8 summarizes the support for high availability in OracleAS JMS and OJMS.

Table 3–8 High Availability Summary

Feature	OJMS	OracleAS JMS
High availability	RAC + OPMPN	OPMPN
Configuration	RAC configuration, resource provider configuration	Dedicated JMS server, jms.xml configuration, opmn.xml configuration
Message store	On RAC database	In dedicated JMS server/persistence files
Failover	Same or different machine (depending on RAC)	Same or different machine within Oracle Application Server Cold Failover Cluster (Midtier)

JMS clustering provides an environment wherein JMS applications deployed in this type of environment can load balance JMS requests across multiple OC4J instances or processes. Clustering of stateless applications is trivial: The application is deployed on multiple servers, and user requests are routed to one of them.

JMS is a stateful API: Both the JMS client and the JMS server contain state about each other, which includes information about connections, sessions, and durable subscriptions. Users can configure their environment and use a few simple techniques when writing their applications to make them cluster-friendly.

The following sections discuss how both OJMS and OracleAS JMS use high availability and clustering:

- [OracleAS JMS High Availability Configuration](#)
- [OJMS High Availability Configuration](#)
- [Failover Scenarios When Using a RAC Database With OJMS](#)
- [Server Side Sample Code for Failover for Both JMS Providers](#)
- [Clustering Best Practices](#)

OracleAS JMS High Availability Configuration

OracleAS JMS clustering normally implies that an application deployed in this type of environment is able to load balance messages across multiple instances of OC4J. There is also a degree of high availability in this environment because the container processes can be distributed across multiple nodes. If any of the processes or nodes goes down, then the processes on an alternate node continue to service messages.

This section describes two JMS clustering scenarios:

- [OracleAS JMS Server Distributed Destinations](#)

In this configuration, all factories and destinations are defined on all OC4J instances. Each OC4J instance has a separate copy of each of the destinations. Each copy of the destinations is unique and is not replicated or synchronized across OC4J instances. Destinations can be persistent or in-memory. A message enqueued to one OC4J instance can be dequeued only from that OC4J instance.

This configuration is ideal for high-throughput applications where requests must be load balanced across OC4J instances. No configuration changes are required for this scenario.

- [Cold Failover Cluster](#)

This configuration is a two-node cluster. Only one node is active at any time. The second node is made active if the first node fails.

- [OracleAS Dedicated JMS Server](#)

In this configuration, a single JVM within a single OC4J instance is dedicated as the JMS server. All other OC4J instances that are hosting JMS clients forward their JMS requests to the dedicated JMS server.

This configuration is the easiest to maintain and troubleshoot and should be suitable for the majority of OracleAS JMS applications, especially those where message ordering is a requirement.

Terminology

The terms introduced here are explained in the *Oracle Application Server High Availability Guide* and the *Oracle Process Manager and Notification Server Administrator's Guide*.

- *OHS*—Oracle HTTP Server

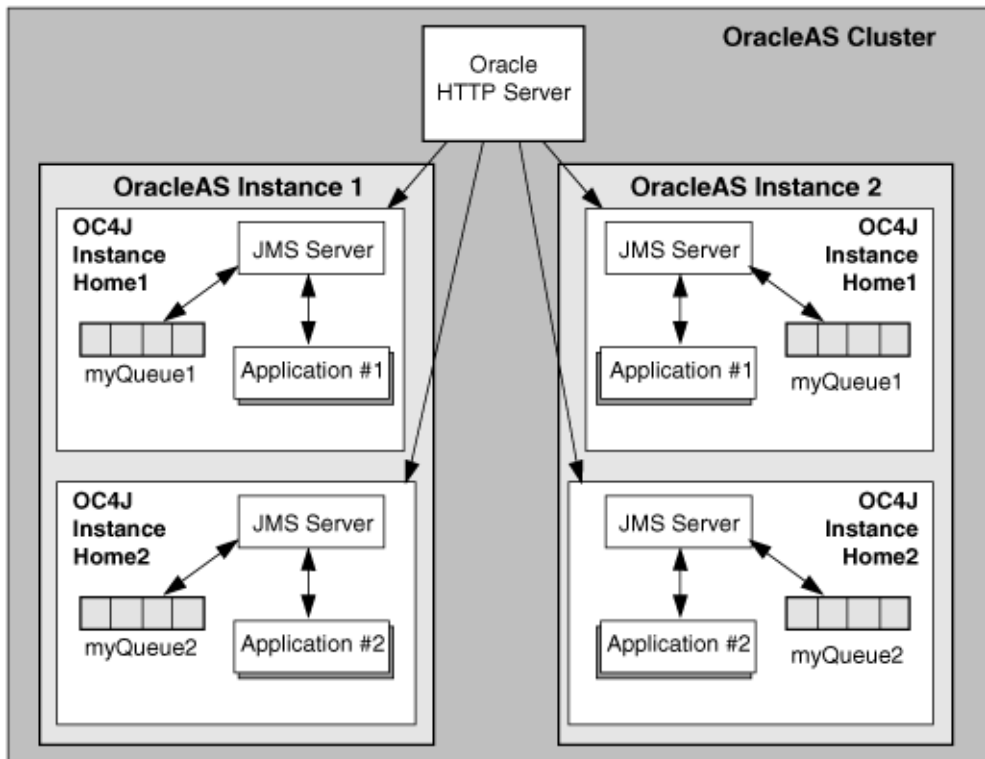
- *OracleAS Cluster*—A grouping of similarly configured Oracle Application Server instances
- *Oracle Application Server Instance*—Represents an installation of Oracle Application Server (that is, an *ORACLE_HOME*)
- *OC4J Instance*—Within an Oracle Application Server instance there can be multiple OC4J instances, and each OC4J instance has 1 to n identically-configured JVMs.
- *Factory*—Denotes a JMS connection factory
- *Destination* —Denotes a JMS destination

OracleAS JMS Server Distributed Destinations

In this configuration, OHS services HTTP requests and load balances them across the two Oracle Application Server instances in an Oracle Application Server cluster. This can scale to more than two Oracle Application Server instances. This type of deployment has several advantages:

- High throughput is achieved because applications and the JMS server are both running inside the same JVM and no interprocess communication is necessary.
- Load balancing promotes high throughput as well as high availability.
- There is no single point of failure. As long as one OC4J process is available, then requests can be processed.
- Oracle Application Server instances can be clustered without impacting JMS operations.
- Destination objects can be persistent or in-memory.

Figure 3–2 OracleAS JMS Server Distributed Destinations



O_1087

Within each Oracle Application Server instance, two OC4J instances have been defined. Each of these OC4J instances is running a separate application. In other words, OC4J instance #1 (Home1) is running Application #1 while OC4J instance #2 (Home2) is running Application #2. Remember, each OC4J instance can be configured to run multiple JVMs, allowing the application to scale across these multiple JVMs.

Within an Oracle Application Server cluster, the configuration information for each Oracle Application Server instance is identical (except for the instance-specific information such as host name, port numbers, and so on). This means that Application #1 deployed to OC4J instance #1 in Oracle Application Server instance #1 is also deployed on OC4J instance #1 in Oracle Application Server instance #2. This type of architecture allows for load balancing of messages across multiple Oracle Application Server instances—as well as high availability of the JMS application, especially if Oracle Application Server instance #2 is deployed to another node to ensure against hardware failure.

The sender and receiver of each application must be deployed together on an OC4J instance. In other words, a message enqueued to the JMS Server in one OC4J process can be dequeued only from that OC4J process.

All factories and destinations are defined on all OC4J processes. Each OC4J process has a separate copy of each of the destinations. The copies of destinations are not replicated or synchronized. So, in the diagram, Application #1 is writing to a destination called `myQueue1`. This destination physically exists in two locations (Oracle Application Server instance #1 and #2) and is managed by the respective JMS servers in each OC4J instance.

Note that this type of JMS deployment is suited only for specific types of JMS applications. Assuming that message order is not a concern, messages are enqueued onto distributed queues of the same name. Given the semantics of JMS point-to-point messaging, messages must not be duplicated across multiple queues. In the preceding case, messages are sent to whatever queue the load balancing algorithm determines, and the MDBs dequeue them as they arrive.

Cold Failover Cluster

This configuration is a two-node cluster. Only one node is active at any time. The second node is made active if the first node fails. For Cold Failover documentation, see the *Oracle Application Server 10g High Availability Guide*.

Configure

Configure both nodes identically as described in the following example. Modify the `jms.xml` file for both OC4J instances. Set the host parameter in the `jms-server` to be:

```
<jms-server host=vmt.us.oracle.com port="9127">
...
...
</jms-server>
```

When using file-based message persistence for a queue, the file must be located on a shared disk that is accessible by both nodes. The shared disk must fail over with the virtual IP when failing over from one node to the other. Configure the `persistence-file` as follows:

```
<queue name="Demo Queue" location="jms/demoQueue"
persistence-file="/path/to/shared_file_system/demoQueueFile">
  <description>A dummy queue</description>
</queue>
```

Update, Stop, and Start

On each node, use the following commands to update configuration, stop, and start:

```

$ORACLE_HOME/dcm/bin/dcmctl updateConfig -ct oc4j
$ORACLE_HOME/opmn/bin/opmnctl stopall
$ORACLE_HOME/opmn/bin/opmnctl startall
    
```

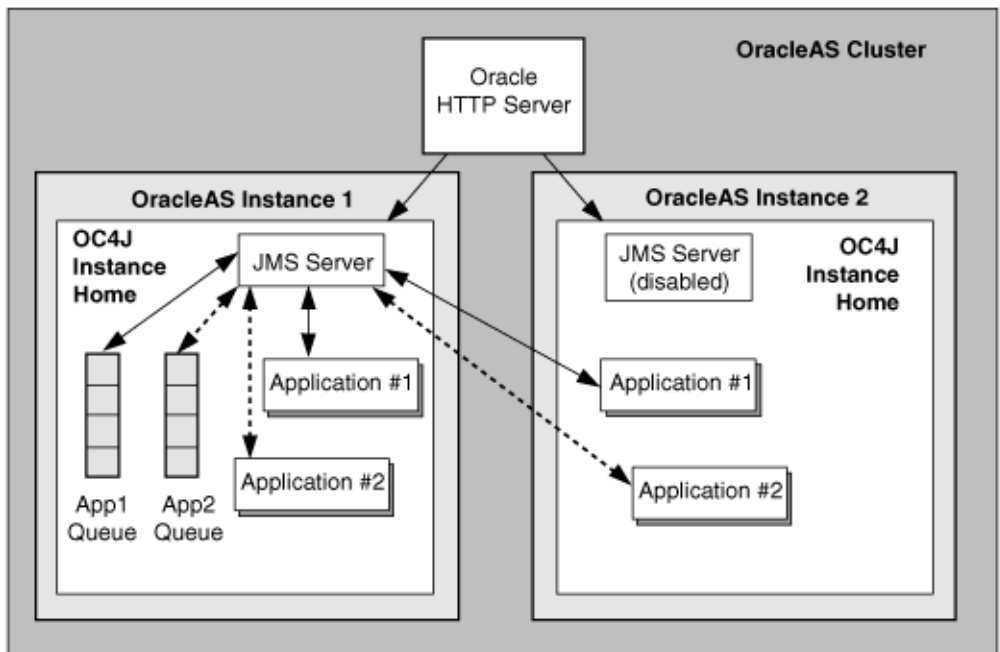
OracleAS Dedicated JMS Server

In this configuration, a single OC4J instance is configured as the dedicated JMS server within an Oracle Application Server clustered environment. This OC4J instance handles all messages, thus message ordering is always maintained. All JMS applications use this dedicated server to host their connection factories and destinations, and to service their enqueue and dequeue requests.

Only one OC4J JVM is acting as the dedicated JMS provider for all JMS applications within the cluster. This is achieved by limiting the JMS port range in the `opmn.xml` file to only one port for the dedicated OC4J instance.

Although this diagram shows the active JMS server in the OC4J Home instance, Oracle recommends that the JMS provider be hosted in its own OC4J instance. For example, although Home is the default OC4J instance running after an Oracle Application Server install, you should create a second OC4J instance with the Oracle Enterprise Manager 10g. In the `opmn.xml` file example following, we have created an OC4J instance called `JMSserver`.

Figure 3–3 OracleAS Dedicated JMS Server



After creating an OC4J instance called `JMSserver`, we must make the following two changes to the `opmn.xml` file for this Oracle Application Server instance:

1. Make sure that only one JVM is started for this OC4J instance (JMSserver).
2. Specify only one value for the JMS port range for this instance.

The single JVM in the OC4J instance ensures that other JVMs will not attempt to use the same set of persistent files.

The single port value ensures that OPMN always assigns this value to the dedicated JMS server. This port value is used to define the connection factory in the `jms.xml` file that other OC4J instances will use to connect to the dedicated JMS server.

For more information on OPMN and dynamic port assignments, see the *Oracle Process Manager and Notification Server Administrator's Guide*.

Modifying the OPMN Configuration

Note: When editing any configuration file by hand (that is, not using Oracle Enterprise Manager 10g), run the following Distributed Configuration Management (DCM) command:

```
dcmctl updateConfig
```

See the *Distributed Configuration Management Administrator's Guide* for more information.

The following XML from the `opmn.xml` file shows what changes must be made and how to find where to make these changes.

- Assuming an OC4J instance has been created through Oracle Enterprise Manager 10g called JMSserver, then the line denoted by (1) demonstrates where to locate the start of the JMSserver definition.
- The line denoted by (2) is the JMS port range that OPMN uses when assigning JMS ports to OC4J JVMs. For the desired dedicated OC4J instance that acts as your JMS provider, narrow this range down to one value. In this example, the original range was from 3701-3800. In our connection factory definitions, we know the port to use by configuring this value as 3701-3701.
- The line denoted by (3) defines the number of JVMs that will be in the JMSserver default island. By default, this value is set to 1. This value must always be 1.

```
<ias-component id="OC4J">
  (1) <process-type id="JMSserver" module-id="OC4J" status="enabled">
    <module-data>
      <category id="start-parameters">
        <data id="java-options" value="-server
          -Djava.security.policy=$ORACLE_HOME/j2ee/home/config/java2.policy
          -Djava.awt.headless=true
        "/>
      </category>
      <category id="stop-parameters">
        <data id="java-options"
          value="-Djava.security.policy=
            $ORACLE_HOME/j2ee/home/config/java2.policy
            -Djava.awt.headless=true"/>
        </category>
      </module-data>
      <start timeout="600" retry="2"/>
      <stop timeout="120"/>
    </process-type>
  </ias-component>
```

```
<restart timeout="720" retry="2"/>
<port id="ajp" range="3000-3100"/>
<port id="rmi" range="3201-3300"/>
(2) <port id="jms" range="3701-3701"/>
(3) <process-set id="default_island" numprocs="1"/>
</process-type>
</ias-component>
```

Configuring OracleAS JMS

As already described in this scenario, one of the OC4J instances is dedicated as the JMS server. Other OC4J instances and standalone JMS clients running outside OC4J must be set up to forward JMS requests to the dedicated JMS server. All connection factories and destinations are defined in the JMS server instance's `jms.xml` file. This `jms.xml` file should then be copied to all the other OC4J instances that will be communicating with the JMS server.

The connection factories configured in the `jms.xml` file on the dedicated JMS server should specify, explicitly, the host name and the port number of the server. These values, in particular the port number, should also use the single port number defined by OPMN for the dedicated server as discussed above. The same connection factory configuration should also be used in all other OC4J instances so that they all point to the dedicated JMS server for their operations.

Thus, if the dedicated JMS server runs on `host1`, port 3701, then all connection factories defined within the `jms.xml` file for each OC4J instance in the cluster should point to `host1`, port 3701—where this port is the single port available in the `opmn.xml` file used in the dedicated OC4J instance (in our example, `JMSserver`) used for the dedicated JMS server.

The destinations configured in the `jms.xml` file on the dedicated JMS server should also be configured on all other OC4J instances; the physical store for these destinations, however, is on the dedicated JMS server.

Queue Connection Factory Definition Example

The following is an example for defining a queue connection factory in the `jms.xml` file of the dedicated OracleAS JMS server.

```
<!-- Queue connection factory -->
<queue-connection-factory name="jms/MyQueueConnectionFactory"
    host="host1" port="3701"
    location="jms/MyQueueConnectionFactory"/>
```

Administrative changes (that is, add a new `Destination` object) should be made to the dedicated JMS server's `jms.xml` file. These changes should then be made in the `jms.xml` files of all other OC4J instances running JMS applications. Changes can be made either by hand or by copying the dedicated JMS server's `jms.xml` file to the other OC4J instances.

Deploying Applications

The user decides where the JMS applications will actually be deployed. Although the dedicated JMS server services JMS requests, it can also execute deployed JMS applications. JMS applications can also be deployed to other OC4J instances (that is, `Home`).

Remember, the `jms.xml` file from the dedicated JMS server must be propagated to all OC4J instances where JMS applications are to be deployed. JMS applications can also be deployed to standalone JMS clients running in separate JVMs.

High Availability

OPMN provides the failover mechanism to keep the dedicated JMS server up and running. If for some reason the JMS server fails, OPMN detects this and restarts the JVM. If a hardware failure occurs, then the only way to recover messages is to have the persisted destinations hosted on a network file system. An OC4J instance can then be brought up and configured to point to these persisted files.

See the *Oracle Process Manager and Notification Server Administrator's Guide* for more information on how OPMN manages Oracle Application Server processes.

OJMS High Availability Configuration

To enable high availability with OJMS, run the following:

- Run an Oracle database that contains the AQ queues and topics in RAC-mode. This ensures that the database is highly available.
- Run Oracle Application Server in OPMN-mode. This ensures that the application servers (and applications deployed on them) are highly available.

Each application instance in an Oracle Application Server cluster uses OC4J resource providers to point to the back-end Oracle database, which is operating in RAC-mode. JMS operations invoked on objects derived from these resource providers are directed to the real application clusters (RAC) database.

If a failure of the application occurs, then state information in the application is lost (that is, state of connections, sessions, and messages not yet committed). As the application server is restarted, the applications must re-create their JMS state appropriately and resume operations.

If network failover of a back-end database occurs, where the database is a non-RAC database, then state information in the server is lost (that is, state of transactions not yet committed). Additionally, the JMS objects (connection factories, `Destination` objects, connections, sessions, and so on) inside the application may also become invalid. The application code can see exceptions if it attempts to use these objects after the failure of the database occurs. The code throws a `JMSException` until it gets to the point where it can look up, through JNDI, all JMS administered objects, and proceed from there.

Failover Scenarios When Using a RAC Database With OJMS

An application that uses a RAC database must handle database failover scenarios. There are two types of failover scenarios, as described in [Chapter 4, "Data Sources"](#). The following sections demonstrate how to handle each failover scenario:

- [Using JMS with RAC Network Failover](#)
- [Using OJMS With Transparent Application Failover \(TAF\)](#)

Note: The RAC-enabled attribute of a data source is discussed in [Chapter 4, "Data Sources"](#). For more information on using this flag with an infrastructure database, see the *Oracle Application Server High Availability Guide*.

Using JMS with RAC Network Failover

A standalone OJMS client running against an RAC database must write code to obtain the connection again, by invoking the API

`com.evermind.sql.DbUtil.oracleFatalError()`, to determine if the connection object is invalid. It must then reestablish the database connection if necessary. The `oracleFatalError()` method detects if the SQL error thrown by the database during network failover is a fatal error. This method takes in the SQL error and the database connection, and returns `true` if the error is a fatal error. If `true`, you may wish to aggressively roll back transactions and re-create the JMS state (such as connections, session, and messages that were lost).

The following example outlines the logic:

```
getMessage(QueueSession session)
{
    try
    {
        QueueReceiver rcvr;
        Message msgRec = null;
        QueueReceiver rcvr = session.createReceiver(rcvrQueue);
        msgRec = rcvr.receive();
    }
    catch(Exception e )
    {
        if (exc instanceof JMSEException)
        {
            JMSEException jmsexc = (JMSEException) exc;
            sql_ex = (SQLException)(jmsexc.getLinkedException());

            db_conn =
                (oracle.jms.AQjmsSession)session.getDBConnection();

            if ((DbUtil.oracleFatalError(sql_ex, db_conn))
            {
                // failover logic
            }
        }
    }
}
```

Using OJMS With Transparent Application Failover (TAF)

Note: [Chapter 4, "Data Sources"](#) discusses transparent application failure (TAF).

In most cases where TAF is configured, the application does not notice that failover to another database instance has occurred. So, you need not do anything to recover from failure.

However, in some cases, OC4J throws an ORA error when a failure occurs. OJMS passes these errors to the user as a `JMSEException` with a linked SQL exception. In this case, do one or more of the following:

- As described in "[Using JMS with RAC Network Failover](#)" on page 3-47, you can use the `DbUtil.oracleFatalError` method to determine if the error is a fatal error. If it is not a fatal error, then the client recovers by sleeping for a short time and then retrying the current operation.

- You can recover from failback and transient errors caused by incomplete failover by trying to use the JMS connection after a short time. Waiting allows the database failover to recover from the failure and reinstate itself.
- In the case of transaction exceptions (such as "Transaction must roll back" (ORA-25402) or "Transaction status unknown" (ORA-25405)) you must roll back the current operation and retry all operations past the last commit. The connection is not usable until the cause of the exception is dealt with. If this retry fails, then close and re create all connections and retry all noncommitted operations.

Server Side Sample Code for Failover for Both JMS Providers

The following shows JMS application code for a queue that is tolerant to server-side failover. This example is valid for both OJMS and OracleAS JMS.

```
while (notShutdown)
{
    Context ctx = new InitialContext();

    /* create the queue connection factory */
    QueueConnectionFactory qcf = (QueueConnectionFactory)
        ctx.lookup(QCF_NAME);
    /* create the queue */
    Queue q = (Queue) ctx.lookup(Q_NAME);
    ctx.close();

    try
    {
        /*Create a queue connection, session, sender and receiver */
        QueueConnection qc = qcf.createQueueConnection();
        QueueSession qs = qc.createQueueSession(true, 0);
        QueueSender snd = qs.createSender(q);
        QueueReceiver rcv = qs.createReceiver(q);

        /* start the queue */
        qc.start();

        /* receive requests on the queue receiver and send out
           replies on the queue sender.
        while (notDone)
        {
            Message request = rcv.receive();
            Message reply = qs.createMessage();

            /* put code here to process request and construct reply */

            snd.send(reply);
            qs.commit();
        }
        /* stop the queue */
        qc.stop();
    }
    catch (JMSEException ex)
    {
        if (transientServerFailure)
        { // retry }
        else {
            notShutdown = false;
        }
    }
}
```

Clustering Best Practices

- Minimize JMS client-side state.
 - Perform work in transacted sessions.
 - Save/checkpoint intermediate program state in JMS queues/topics for full recoverability.
 - Do not depend on J2EE application state to be serializable or recoverable across JVM boundaries. Always use transient member variables for JMS objects, and write passivate/activate and serialize/deserialize functions that save and recover JMS state appropriately.
- Do not use nondurable subscriptions on topics.
 - Nondurable topic subscriptions duplicate messages per active subscriber. Clustering and load-balancing creates multiple application instances. If the application creates a nondurable subscriber, it causes the duplication of each message published to the topic. This is either inefficient or semantically invalid.
 - Use only durable subscriptions for topics. Use queues whenever possible.
- Do not keep durable subscriptions alive for extended periods of time.
 - Only one instance of a durable subscription can be active at any given time. Clustering and load-balancing creates multiple application instances. If the application creates a durable subscription, only one instance of the application in the cluster succeeds. All other instances fail with a `JMSException`.
 - Create, use, and close a durable subscription in small time/code windows, minimizing the duration when the subscription is active.
 - Write application code that accommodates failure to create durable subscription due to clustering (when some other instance of the application running in a cluster is currently in the same block of code) and program appropriate back-off strategies. Do not always treat the failure to create a durable subscription as a fatal error.

Data Sources

This chapter describes how to configure and use data sources in your Oracle Application Server Containers for J2EE (OC4J) application. A data source is a vendor-independent encapsulation of a connection to a database server. A data source instantiates an object that implements the `javax.sql.DataSource` interface.

This chapter covers the following topics:

- [Introduction](#)
- [Defining Data Sources](#)
- [Using Data Sources](#)
- [Using Two-Phase Commits and Data Sources](#)
- [Using Oracle JDBC Extensions](#)
- [Using Connection Caching Schemes](#)
- [Using the OCI JDBC Drivers](#)
- [Using DataDirect JDBC Drivers](#)
- [High Availability Support for Data Sources](#)

Introduction

A *data source* is a Java object that implements the `javax.sql.DataSource` interface. Data sources offer a portable, vendor-independent method for creating JDBC connections. Data sources are factories that return JDBC connections to a database. J2EE applications use JNDI to look up `DataSource` objects. Each JDBC 2.0 driver provides its own implementation of a `DataSource` object, which can be bound into the JNDI name space. After this data source object has been bound, you can retrieve it through a JNDI lookup. Because data sources are vendor-independent, we recommend that J2EE applications retrieve connections to data servers using data sources.

Types of Data Sources

In OC4J, Data Sources are classified as follows:

- [Emulated Data Sources](#)
- [Nonemulated Data Sources](#)
- [Native Data Sources](#)

Table 4–1 summarizes the key differences between the data source types.

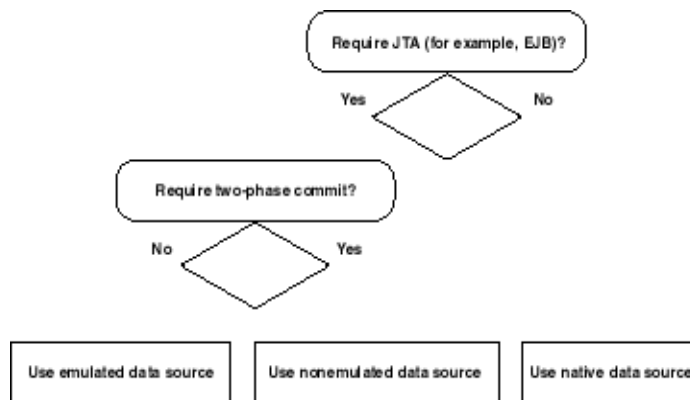
Table 4–1 Features of Emulated, Nonemulated, and Native Data Sources

	Emulated	Not Emulated
No JTA		Native Data Source <ul style="list-style-type: none"> ▪ Vendor extensions ▪ Vendor JDBC pool/cache ▪ No JTA
JTA	Emulated Data Source <ul style="list-style-type: none"> ▪ Lightweight transactions ▪ One-phase commit ▪ OC4J pool/cache 	Nonemulated Data Source <ul style="list-style-type: none"> ▪ Full transactions ▪ Two-phase commit ▪ Oracle JDBC pool/cache

: Note

If you access a nonemulated data source by the `ejb-location`, then you are using the OC4J pool and cache. If you use `OracleConnectionCacheImpl`, then you can access both OC4J and Oracle JDBC pool and cache.

Figure 4–1 summarizes the decision tree that should guide you when choosing a data source type.

Figure 4–1 Choosing a Data Source Type

The following sections describe each data source type in detail.

Emulated Data Sources

Emulated data sources are data sources that emulate the XA protocol for JTA transactions. Emulated data sources offer OC4J caching, pooling, and Oracle JDBC extensions for Oracle data sources. Historically, emulated data sources were necessary because many JDBC drivers did not provide XA capabilities. Today even though most JDBC drivers do provide XA capabilities, there are still cases in which emulated XA is preferred (such as transactions that do not require two-phase commit.)

Connections obtained from emulated data sources are extremely fast, because the connections emulate the XA API without providing full XA global transactional

support. In particular, emulated data sources do not support two-phase commit. Oracle recommends that you use emulated data sources for local transactions, or when your application uses global transactions without requiring two-phase commit. For information on the limitations of two-phase commit, see [Chapter 7, "Java Transaction API"](#).

The following is a `data-sources.xml` configuration entry for an emulated data source:

```
<data-source
  class="com.evermind.sql.DriverManagerDataSource"
  name="OracleDS"
  location="jdbc/OracleCoreDS"
  xa-location="OracleDS"
  ejb-location="jdbc/OracleDS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="scott"
  password="tiger"
  url="jdbc:oracle:thin:@//localhost:5521/oracle.regress.rdbms.dev.us.oracle.com"
  inactivity-timeout="30"
/>
```

When defining an emulated data source in `data-sources.xml`, you must provide values for the `location`, `ejb-location`, and `xa-location` attributes. However, when looking up an emulated data source through JNDI, you should look it up by the value specified with the `ejb-location` attribute. For example:

```
Context ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("jdbc/OracleDS");
// This lookup could also be done as
// DataSource ds = (DataSource) ic.lookup("java:comp/env/jdbc/OracleDS");
Connection con = ds.getConnection();
```

This connection opens a database session for `scott/tiger`.

Note : Previous releases supported the `location` and `xa-location` attributes for retrieving data source objects. These attributes are now strongly deprecated. Applications, EJBs, servlets, and JSPs should use only the JNDI name `ejb-location` in emulated data source definitions for retrieving the data source. You must specify all three values for emulated data sources, although only `ejb-location` is actually used.

If you use an emulated data source inside a global transaction, you must exercise caution. Because the `XAResource` that you enlist with the transaction manager is an emulated `XAResource`, the transaction will not be a true two-phase commit transaction. If you want true two-phase commit semantics in global transactions, then you must use a nonemulated data source. For information on the limitations of two-phase commit, see [Chapter 7, "Java Transaction API"](#).

Retrieving multiple connections from a data source using the same user name and password within a single global transaction causes the logical connections to share a single physical connection. The following code shows two connections—`conn1` and `conn2`—that share a single physical connection. They are both retrieved from the same data source object. They also authenticate with the same user name and password.

```
Context ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("jdbc/OracleCMTDS1");
```

```
Connection conn1 = ds.getConnection("scott", "tiger");
Connection conn2 = ds.getConnection("scott", "tiger");
```

Nonemulated Data Sources

Nonemulated data sources provide full (nonemulated) JTA services, including two-phase commit capabilities for global transactions. Nonemulated data sources offer pooling, caching, distributed transactions capabilities, and vendor JDBC extensions (currently, only Oracle JDBC extensions). For information on the limitations of two-phase commit, see [Chapter 7, "Java Transaction API"](#).

Oracle recommends that you use nonemulated data sources for distributed database communications, recovery, and reliability. Nonemulated data sources share physical connections for logical connections to the same database for the same user.

Note: You must use a Java-enabled database to run a nonemulated data source. When you use a nonemulated data source and a non-Java-enabled database, deploying any MDB application (AQJMS) generates an exception, thrown to OC4J stdout. If you switch to an emulated data source or a Java-enabled database, deployment proceeds correctly.

Note: When an application using a nonemulated data source is undeployed, the physical database connection is not removed from OC4J until OC4J is restarted.

The following is a `data-sources.xml` configuration entry for a nonemulated data source:

```
<data-source
  class="com.evermind.sql.OrionCMTDataSource"
  location="jdbc/OracleDS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="scott"
  password="tiger"
  url="jdbc:oracle:thin:@//localhost:5521/oracle.regress.rdbms.dev.us.oracle.com"
</data-source>
```

JNDI lookups should be performed using the value of the `location` attribute.

Here are the expected attribute definitions:

- `location` is the JNDI name to which this data source is bound within the JNDI name space. Use `location` in the JNDI lookup for retrieving this data source.
- `url`, `username`, and `password` identify the database and default user name and password to use when connections are retrieved with this data source.
- `class` defines what type of data source class to bind in the name space.

Native Data Sources

Native data sources are JDBC-vendor supplied implementations of the `DataSource`. They expose vendor's JDBC driver capabilities including caching, pooling, and vendor specific extensions. Exercise caution when using native data sources, because OC4J

cannot enlist them inside global transactions, and they can be used by EJBs or other components requiring global transaction semantics.

Native data source implementations can be used directly without an emulator. OC4J supports the use of native data sources directly and benefits from their vendor-specific pooling, caching, extensions, and properties. However, native data sources do not provide JTA services (such as begin, commit, and rollback)

The following is a `data-sources.xml` configuration entry for a native data source:

```
<data-source
  class="com.my.DataSourceImplementationClass"
  name="NativeDS"
  location="jdbc/NativeDS"
  username="user"
  password="pswd"
  url="jdbc:myDataSourceURL"
</data-source>
```

JNDI lookups can be performed only through the value of the `location` attribute.

Mixing Data Sources

A single application can use several different types of data sources.

If your application mixes data sources, note the following issues:

- Only emulated and nonemulated data sources support JTA transactions.

You cannot enlist connections obtained from native data sources in a JTA transaction.

- Only nonemulated data sources support true two-phase commit (emulated data sources emulate two-phase commit).

To enlist multiple connections in a two-phase commit transaction, all connections must use nonemulated data sources. For information on the limitations of two-phase commit, see [Chapter 7, "Java Transaction API"](#).

- If you have opened a JTA transaction and want to obtain a connection that does not participate in the transaction, then use a native data source to obtain the connection.
- If your application does not use JTA transactions, you can obtain connections from any data source.
- If your application has opened a `javax.transaction.UserTransaction`, all future transaction work must be performed through that object.

If you try to invoke the connection's `rollback()` or `commit()` methods, then you will receive the following `SQLException`:

```
calling commit() [or rollback()] is not allowed on a container-managed
transactions Connection
```

The following example explains what happens:

```
Context ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("JDBC/OracleCMTDS1"); // Using JTA DataSources
Connection conn1 = ds.getConnection("scott", "tiger");
javax.transaction.UserTransaction ut =
    (javax.transaction.UserTransaction)ic.lookup("java:comp/UserTransaction");
ut.begin();
conn1.query();
```

```
conn1.commit(); // not allowed, returns error: calling commit[or rollback] is not allowed
                // on a container-managed transaction connection
```

Defining Data Sources

You define OC4J data sources in the `data-sources.xml` file.

The `data-sources.xml` file installed with OC4J includes predefined, default data sources that may be sufficient for your needs. If not, you must define your own.

[Table 4–2](#) summarizes the configuration requirements for each type of data source.

Table 4–2 Data Source Configuration Summary

Configuration	Nonemulated	Emulated	Native
Data source class	OrionCMTDataSource	DriverManagerDataSource	OracleConnection - CacheImpl
Connection-driver	N/A	vendor specific OracleDriver for Oracle extensions	N/A
JNDI Context specification	location	location ejb-location xa-location	location
JNDI Context lookup	location	ejb-location	location
URL	Oracle driver URL	Vendor-specific Oracle: Thin or OCI (TAF with OCI)	Vendor-specific Oracle: Thin or OCI (TAF with OCI)
Additional configuration	Oracle database commit coordinator Database link for two-phase commit coordinator	None	Cache scheme

Note: For documentation of `OrionCMTDataSource` and its elements, see <http://www.orionserver.com/docs/api/orion/com/evermind/sql/OrionCMTDataSource.html>

[Table 4–3](#) summarizes the characteristics for each type of data source.

Table 4–3 Data Source Characteristics

Characteristic	Nonemulated	Emulated	Native
Pool and cache support	Oracle JDBC driver pool	OC4J connection pool	vendor specific Oracle
Vendor extension support	Oracle only	Oracle only	vendor specific Oracle
JTA support	Full XA (one- or two-phase commit)	Emulated XA (one-phase commit)	Not supported
J2CA support	No	Yes	Yes

Note: If you access a nonemulated data source by the `ejb-location`, then you are using the OC4J pool and cache. If you use `OracleConnectionCacheImpl`, then you can access both OC4J and Oracle JDBC pool and cache.

To define a new data source object:

1. Select a location for the `data-sources.xml` file (see "[Configuration Files](#)" on page 4-7).
2. Understand data source attributes (see "[Data Source Attributes](#)" on page 4-8).
3. Define a data source either by using the Oracle Enterprise Manager 10g (See "[Defining Data Sources in Oracle Enterprise Manager 10g](#)" on page 4-10) or by manually editing configuration files. (See "[Defining Data Sources in the XML Configuration File](#)" on page 4-11).

Configuration Files

One main configuration file establishes data sources at the OC4J server level: `J2EE_HOME/config/data-sources.xml`.

Each application also has a separate JNDI name space. The files `web.xml`, `ejb-jar.xml`, `orion-ejb-jar.xml`, and `orion-web.xml` contain entries that you can use to map application JNDI names to data sources, as the next section describes.

Defining Location of the Data Source XML Configuration File

Your application can know about the data sources defined in this file only if the `application.xml` file knows about it. The `path` attribute in the `<data-sources>` tag in the `application.xml` file must contain the name and path to your `data-sources.xml` file, as follows:

```
<data-sources path="data-sources.xml" />
```

The `path` attribute of the `<data-sources>` tag contains a full path name for the `data-sources.xml` file. The path can be absolute, or it can be relative to where the `application.xml` is located. Both the `application.xml` and `data-sources.xml` files are located in the `J2EE_HOME/config/application.xml` directory. Thus, the path contains only the name of the `data-sources.xml` file.

Application-Specific Data Source XML Configuration File

Each application can define its own `data-sources.xml` file in its EAR file. This is done by having the reference to the `data-sources.xml` file in the `orion-application.xml` file packaged in the EAR file.

To configure this:

1. Locate the `data-sources.xml` and `orion-application.xml` files in your application's META-INF directory.
2. Edit the `orion-application.xml` file to add a `<data-sources>` tag as follows:

```
<orion-application>
  <data-sources path="./data-sources.xml"/>
</orion-application>
```

Data Source Attributes

A data source can take many attributes. Some are required; most are optional. The required attributes are marked below. The attributes are specified in a `<data-source>` tag.

[Table 4-4](#) lists and describes the data source attributes.

In addition to the `data-source` attributes described in [Table 4-4](#), you can also add property subnodes to a `data-source`. These are used to configure generic properties on a data source object (following Java Bean conventions.) A property node has a name and value attribute used to specify the name and value of a data source bean property.

All OC4J data source attributes are applicable to the infrastructure database as well. For more information on the infrastructure database, see *Oracle High Availability Architecture and Best Practices*.

Table 4-4 Data Source Attributes

Attribute Name	Description	Default Value
class	Names the class that implements the data source. For nonemulated, this can be <code>com.evermind.sql.OrionCMTDataSource</code> . For emulated, this should be <code>com.evermind.sql.DriverManagerDataSource</code> . (This value is required.)	N/A
location	The JNDI logical name for the data source object. OC4J binds the class instance into the application JNDI name space with this name. This JNDI lookup name is used for nonemulated data sources. See also Table 4-2, "Data Source Configuration Summary" on page 4-6.	N/A
name	The data source name. Must be unique within the application.	None
connection-driver	The JDBC-driver class name for this data source, used by some data sources that deal with <code>java.sql.Connection</code> . For most data sources, the driver is <code>oracle.jdbc.driver.OracleDriver</code> . This attribute applies only to emulated data sources for which the <code>class</code> attribute is <code>com.evermind.sql.DriverManagerDataSource</code> .	None
username	Default user name used when getting data source connections.	None

Table 4-4 (Cont.) Data Source Attributes

Attribute Name	Description	Default Value
password	Default password used when getting data source connections. See also "Password Indirection" on page 4-12.	None
URL	The URL for database connections.	None
xa-location	The logical name of an XA data source. Applies to emulated data sources only. See also Table 4-2, "Data Source Configuration Summary" on page 4-6.	None
ejb-location	Use this attribute for JTA single-phase commit transactions or for looking up emulated data sources. If you use it to retrieve the data source, you can map the returned connection to <code>oracle.jdbc.OracleConnection</code> . See also Table 4-2, "Data Source Configuration Summary" on page 4-6.	None
stmt-cache-size	A performance tuning attribute set to a non-zero value to enable JDBC statement caching and to define the maximum number of statements cached. Enabled to avoid the overhead of repeated cursor creation, and statement parsing and creation. Applicable only for emulated data sources for which <code>connection-driver</code> is <code>oracle.jdbc.driver.OracleDriver</code> and class is <code>com.evermind.sql.DriverManagerDataSource</code> .	0 (disabled)
inactivity-timeout	Time (in seconds) to cache an unused connection before closing it.	60 seconds
connection-retry-interval	Time (in seconds) to wait before retrying a failed connection attempt.	1 second
max-connections	The maximum number of open connections for a pooled data source.	Depends on the data source type
min-connections	The minimum number of open connections for a pooled data source. OC4J does not open these connections until the <code>DataSource.getConnection</code> method is invoked.	0
wait-timeout	The number of seconds to wait for a free connection if the pool has reached <code>max-connections</code> used.	60
max-connect-attempts	The number of times to retry making a connection. Useful when the network or environment is unstable for any reason that makes connection attempts fail.	3
clean-available-connections-threshold	This optional attribute specifies the threshold (in seconds) for when a cleanup of available connections will occur. For example, if a connection is bad, the available connections are cleaned up. If another connection is bad (that is, it throws an exception), and if the threshold time has elapsed, then the available connections are cleaned up again. If the threshold time has not elapsed, then the available connections are not cleaned up again.	30
rac-enabled	This optional attribute specifies whether the system is enabled for Real Application Clusters (RAC). For information on using this flag with an infrastructure database, see <i>Oracle High Availability Architecture and Best Practices</i> . For information on using this flag with a user database, see "Using DataDirect JDBC Drivers" on page 4-23 and "High Availability Support for Data Sources" on page 4-26. If the data source points to an RAC database, set this property to <code>true</code> . This enables OC4J to manage its connection pool in a way that performs better during RAC instance failures.	false
schema	This optional attribute specifies the <code>database-schema</code> associated with a data source. It is especially useful when using CMP with additional data types or third-party databases. For information on using this attribute, see "Associating a Database Schema with a Data Source" on page 4-14.	None

The following example shows the use of the `clean-available-connections-threshold` and `rac-enabled` attributes:

```
<data-source
  class="com.evermind.sql.OrionCMTDataSource"
  name="NEDS1"
  location="jdbc/NELoc1"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  min-connections="5"
  max-connections="10"
  clean-available-connections-threshold="35"
  rac-enabled="true"
  username="scott"
  password="tiger"
  url="jdbc:oracle:thin:@//localhost:5521/oracle.regress.rdbms.dev.us.oracle.com"
  inactivity-timeout="30"
  max-connect-attempts="5"
/>
```

For each data source you define, OC4J may create and bind within JNDI up to four data sources: one each for `location`, `ejb-location`, `xa-location`, and `pool-location`. The type of data source selected is determined by the values associated with `data-sources.xml` attributes `class`, `connection-driver`, and `url`, and the JNDI context in which the data source object is created and looked up. For more information about data source types, see ["Types of Data Sources"](#) on page 4-1.

Defining Data Sources in Oracle Enterprise Manager 10g

You can define any type of data source with the Oracle Enterprise Manager 10g.

The Data Sources Primer chapter of the Oracle Application Server Containers for J2EE User's Guide describes how to define data sources.

See the Oracle Application Server Containers for J2EE User's Guide to find out how to use the Administrative tools. See the *Oracle Enterprise Manager Administrator's Guide* for information on Oracle Enterprise Manager 10g.

This section presents a brief overview of these procedures.

Use the Oracle Enterprise Manager 10g and drill down to the Data Source page. OC4J parses the `data-sources.xml` file when it starts, instantiates data source objects, and binds them into the server JNDI name space. When you add a new data source specification, you must restart the OC4J server to make the new data source available for lookup.

To define emulated data sources, follow the same steps as for defining nonemulated data sources, up to the step in which you define the JNDI location. In the procedure for defining nonemulated data sources, the screen shot shows one field, **Location**, to be filled out. For defining an emulated data source, fill out the three fields **Location**, **XA-Location**, and **EJB-Location**.

Note: Previous releases supported the `location` and `xa-location` attributes for retrieving data source objects. These attributes are now strongly deprecated; applications, EJBs, servlets, and JSPs should use only the JNDI name `ejb-location` in emulated data source definitions for retrieving the data source. You must specify all three values must be specified for emulated data sources, but only `ejb-location` is actually used.

Defining Data Sources in the XML Configuration File

The `$J2EE_HOME/config/data-sources.xml` file is preinstalled with a default data source. For most uses, this default is all you need. However, you can also add your own customized data source definitions.

The default data source is an emulated data source.

For more information about data source types, see "[Types of Data Sources](#)" on page 4-1.

The following is a simple emulated data source definition that you can modify for most applications:

```
<data-source
  class="com.evermind.sql.DriverManagerDataSource"
  name="OracleDS"
  location="jdbc/OracleCoreDS"
  xa-location="OracleDS"
  ejb-location="jdbc/OracleDS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="scott"
  password="tiger"
  url="jdbc:oracle:thin:@//localhost:5521/oracle.regress.rdbms.dev.us.oracle.com"
  inactivity-timeout="30"
/>
```

See "[Data Source Attributes](#)" on page 4-8 for details on all data source attributes.

Fatal Error Code Enhancement

For each data source defined in `data-sources.xml`, you can define fatal error codes that indicate that the back-end database with which the data source communicates is no longer accessible. When OC4J detects one of these error codes (stated when a `SQLException` is thrown by the JDBC driver), OC4J will clean its connection pool. That is, it closes all connections in the connection pool. For Oracle, the predefined fatal error codes are: 3113, 3114, 1033, 1034, 1089, and 1090.

Use the following procedure to add additional fatal error codes for Oracle.

Use the `<fatal-error-codes>` element, which is a subtag of the `<data-source>` element. The `<fatal-error-codes>` element uses the child element `<error-code>` to define one fatal error code. You can define 0 - n `<error-code>` elements for each `<fatal-error-codes>` element. For example, for fatal error codes 10, 20, and 30, the data source definition would look like this:

```
<data-source
  class="com.evermind.sql.DriverManagerDataSource"
  name="ds"
  location="jdbc/ds"
  xa-location="jdbc/xa/ds"
  ejb-location="jdbc/ejb/ds"
```

```
@ connection-driver="oracle.jdbc.driver.OracleDriver"
username="scott"
@ password="tiger"

@ url="jdbc:oracle:thin:@//localhost:1521/oracle.regress.rdbms.dev.us.oracle.com">

    <fatal-error-codes>
        <error-code code='10' />
        <error-code code='20' />
        <error-code code='30' />
    </fatal-error-codes>

</data-source>
```

Password Indirection

The `data-sources.xml` file requires passwords for authentication. Embedding these passwords into deployment and configuration files poses a security risk, especially if the permissions on this file allow it to be read by any user. To avoid this problem, OC4J supports password indirection.

An indirect password is made up of a special indirection symbol (`->`) and a user name (or user name and realm). When OC4J encounters an indirect password, it uses its privileged access to retrieve the password associated with the specified user from the security store provided by a user manager.

For more information on creating users and passwords, and working with a user manager, see the section on password management in the *Oracle Application Server Containers for J2EE Security Guide*.

For example, the sample code under "[Emulated Data Sources](#)" on page 4-2 contains the following line:

```
password="tiger"
```

You could replace that with the indirection symbol (`->`) and a user name (`scott`) as follows:

```
password="->scott"
```

This assumes that a user named `scott` with the password `tiger` has been created in a user manager.

Because OC4J has privileged access to the security store, it can retrieve the password (`tiger`) associated with this user (`scott`).

There are two ways to configure password indirection:

- [Configuring an Indirect Password with Oracle Enterprise Manager 10g](#)
- [Configuring an Indirect Password Manually](#)

Configuring an Indirect Password with Oracle Enterprise Manager 10g

To configure an indirect password using the Oracle Enterprise Manager 10g:

1. Log into the Oracle Enterprise Manager 10g
2. Select a target of type OC4J.

3. Select **Administer**. The Oracle Enterprise Manager 10g for Oracle Application Server home page is displayed.
4. Select **Administration**.
5. Select **Data Sources**. A list of data sources is displayed.
6. Click in the **Select** column to select a data source.
7. Click **Edit**. The Edit Data Source page is displayed as shown in [Figure 4-2](#).

Figure 4-2 Edit Data Source Page

Cleartext passwords may pose a security risk, especially if the permissions on the data-sources.xml configuration file allows it to be read by any user. You can specify an indirect password to avoid this risk. An indirect password is used to do a look up in the User Manager to get the password.

Username

Use Cleartext Password

 Password

Use Indirect Password

 Indirect Password

example: Scott, customers/Scott

8. In the **Username and Password** area, click **Use Indirect Password**, and enter the appropriate value in the **Indirect Password** field.
9. Click **Apply**.

Configuring an Indirect Password Manually

To configure an indirect password for a data source manually:

1. Edit the appropriate OC4J XML configuration or deployment file:
 - data-sources.xml—password attribute of <data-source> element
 - ra.xml — <res-password> element
 - rmi.xml— password attribute of <cluster> element
 - application.xml— password attributes of <resource-provider> and <commit-coordinator> elements
 - jms.xml— <password> element
 - internal-settings.xml— <sep-property> element, attributes name=" keystore-password" and name=" truststore-password"
2. To make any of these passwords indirect, replace the literal password string with a string containing "->", followed either by the username or by the realm and username separated by a slash ("/").

For example: <data-source password="->Scott">

This causes the User Manager to look up the user name "Scott" and use the password stored for that user.

Associating a Database Schema with a Data Source

The data source identifies a database instance. The data source `schema` attribute allows you to associate a data source with a `database-schema.xml` file that you can customize for its particular database.

When using container-managed persistence (CMP), the container is responsible for creating the database schema necessary to persist a bean. Associating a data source with a `database-schema.xml` file allows you to influence what SQL is ultimately generated by the container. This can help you solve problems such as accommodating additional data types supported in your application (such as `java.math.BigDecimal`) but not in your database.

The `database-schema.xml` File

A `database-schema.xml` file contains a `database-schema` element as shown in [Example 4-1](#). It is made up of the attributes listed in [Table 4-5](#).

Example 4-1 The `database-schema` Element

```
<database-schema case-sensitive="true" max-table-name-length="30"
name="MyDatabase" not-null="not null" null="null" primary-key="primary key">
  <type-mapping type="java.math.BigDecimal" name="number(20,8)" />
  <disallowed-field name="order" />
</database-schema>
```

Table 4-5 `database-schema.xml` File Attributes

Attribute	Description
<code>case-sensitive</code>	Specifies whether this database treats names as case sensitive (<code>true</code>) or not (<code>false</code>). This applies to names specified by <code>disallowed-field</code> subelements.
<code>max-table-name-length</code>	This optional attribute specifies the maximum length for table names for this database. Names longer than this value will be truncated.
<code>name</code>	The name of this database.
<code>not-null</code>	Specifies the keyword used by this database to indicate a not-null constraint.
<code>null</code>	Specifies the keyword used by this database to indicate a null constraint.
<code>primary-key</code>	Specifies the keyword used by this database to indicate a primary-key constraint.

The `<database-schema>` element may contain any number of the following subelements:

- `<<type-mapping>>`
- `<<disallowed-field>>`

`<type-mapping>` This subelement is used to map a Java type to the corresponding type appropriate for this database instance. It contains two attributes:

- `name`: the name of the database type
- `type`: the name of the Java type

<disallowed-field> This subelement identifies a name that you must not use because it is a reserved word in this database instance. It contains one attribute:

- name: the name of the reserved word

Example Configuration

This example shows how to map a data type supported in your application (`java.math.BigDecimal`) to a data type supported by the underlying database.

1. Define the mapping for `java.math.BigDecimal` in your `database-schemas/oracle.xml` file as follows:

```
<type-mapping type="java.math.BigDecimal" name="number(20,8)" />
```

2. Use this schema in your `<data-source>` element as follows:

```
<data-source
  class="com.evermind.sql.DriverManagerDataSource"
  name="OracleDS"
  ejb-location="jdbc/OracleDS"
  schema="database-schemas/oracle.xml"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="scott"
  password="tiger"
  url="jdbc:oracle:thin:@//localhost:5521/oracle.regress.rdbms.dev.us.oracle.com"
  clean-available-connections-threshold="30"
  rac-enabled="false"
  inactivity-timeout="30"
/>
```

3. Use this `<data-source>` for your EJBs:

```
<orion-ejb-jar>
  <enterprise-beans>
    <entity-deployment name="BigDecimalTest" data-source="jdbc/OracleDS" />
  </enterprise-beans>
</orion-ejb-jar>
```

4. Deploy your `ejb` and `OC4J` creates the appropriate tables.

Using Data Sources

The following sections describe how to use data sources in your application:

- [Portable Data Source Lookup](#)
- [Retrieving a Connection from a Data Source](#)
- [Retrieving Connections with a Nonemulated Data Source](#)
- [Connection Retrieval Error Conditions](#)

For information on data source methods, refer to your J2EE API documentation.

Portable Data Source Lookup

When the `OC4J` server starts, the data sources in the `data-sources.xml` file in the `j2ee/home/config` directory are added to the `OC4J` JNDI tree. When you look up a data source using JNDI, specify the JNDI lookup as follows:

```
DataSource ds = ic.lookup("jdbc/OracleCMTDS1");
```

The OC4J server looks in its own internal JNDI tree for this data source.

However, we recommend—and it is much more portable—for an application to look up a data source in the *application* JNDI tree, using the portable `java:comp/env` mechanism. Place an entry pointing to the data source in the application `web.xml` or `ejb-jar.xml` files, using the `<resource-ref>` tag. For example:

```
<resource-ref>
  <res-ref-name>jdbc/OracleDS</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

where `<res-ref-name>` can be one of the following:

- The actual JNDI name—such as `jdbc/OracleDS`—that is defined in the `data-sources.xml` file. In this situation, no mapping is necessary. The preceding code example demonstrates this. The `<res-ref-name>` is the same as the JNDI name bound in the `data-sources.xml` file.

Retrieve this data source without using `java:comp/env`, as shown by the following JNDI lookup:

```
InitialContext ic = new InitialContext();
DataSource ds = ic.lookup("jdbc/OracleDS");
```

- A logical name that is mapped to the actual JNDI name in the OC4J-specific files, `orion-web.xml` or `orion-ejb-jar.xml`. The OC4J-specific XML files then define a mapping from the logical name in the `web.xml` or `ejb-jar.xml` file to the actual JNDI name that is defined in the `data-sources.xml` file.

Example 4-2 Mapping Logical JNDI Name to Actual JNDI Name

The following code demonstrates the second of the two preceding options. If you want to choose a logical name of `"jdbc/OracleMappedDS"` to be used within your code for the JNDI retrieval, then place the following in your `web.xml` or `ejb-jar.xml` files:

```
<resource-ref>
  <res-ref-name>jdbc/OracleMappedDS</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

For the actual JNDI name to be found, you must have a `<resource-ref-mapping>` element that maps the `jdbc/OracleMappedDS` to the actual JNDI name in the `data-sources.xml` file. If you are using the default emulated data source, then the `ejb-location` will be defined with `jdbc/OracleDS` as the actual JNDI name. For example:

```
<resource-ref-mapping name="jdbc/OracleMappedDS" location="jdbc/OracleDS" />
```

You can then look up the data source in the application JNDI name space using the Java statements:

```
InitialContext ic = new InitialContext();
DataSource ds = ic.lookup("jdbc/OracleMappedDS");
```

Retrieving a Connection from a Data Source

One way to modify data in your database is to retrieve a JDBC connection and use JDBC statements. We recommend that you, instead, use data source objects in your JDBC operations.

Note: Data sources always return logical connections.

Perform the following steps to modify data within your database:

1. Retrieve the `DataSource` object through a JNDI lookup on the data source definition in the `data-sources.xml` file.

The lookup is performed on the logical name of the default data source, which is an emulated data source defined in the `ejb-location` tag in the `data-sources.xml` file.

You must always cast or narrow the object that JNDI returns to the `DataSource`, because the `JNDI.lookup()` method returns a `Java object`.

2. Create a connection to the database represented by the `DataSource` object.

After you have the connection, you can construct and execute JDBC statements against this database that is specified by the data source.

The following code represents the preceding steps:

```
Context ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("jdbc/OracleDS");
Connection conn = ds.getConnection();
```

Use the following methods of the `DataSource` object in your application code to retrieve a connection to your database:

- `getConnection()`;

The user name and password are those that are defined in the data source definition.

- `getConnection(String username, String password)`;

This user name and password overrides the user name and password that are defined in the data source definition.

If the data source refers to an Oracle database, then you can cast the connection object that is returned on the `getConnection` method to `oracle.jdbc.OracleConnection`, and use all the Oracle extensions. See ["Using Oracle JDBC Extensions"](#) on page 4-21 for details.

The following example illustrates this:

```
oracle.jdbc.OracleConnection conn =
    (oracle.jdbc.OracleConnection) ds.getConnection();
```

After you retrieve a connection, you can execute SQL statements against the database through JDBC.

Refer to ["Retrieving Connections with a Nonemulated Data Source"](#) on page 4-18 for information on handling common connection retrieval error conditions.

Retrieving Connections with a Nonemulated Data Source

The physical behavior of a nonemulated data source object changes, depending on whether you retrieve a connection from the data source outside of or within a global transaction. The following sections discuss these differences:

- [Retrieving a Connection Outside a Global Transaction](#)
- [Retrieving a Connection Within a Global Transaction](#)

Retrieving a Connection Outside a Global Transaction

If you retrieve a connection from a nonemulated data source and you are not involved in a global transaction, then every `getConnection` method returns a logical handle. When the connection is used for work, a physical connection is created for each connection created. Thus, if you create two connections outside of a global transaction, then both connections use a separate physical connection. When you close each connection, it is returned to a pool to be used by the next connection retrieval.

Retrieving a Connection Within a Global Transaction

If you retrieve a connection from a nonemulated data source and you are involved in a global JTA transaction, then all physical connections retrieved from the same `DataSource` object by the same user within the transaction share the same physical connection.

For example, if you start a transaction and retrieve two connections from the `jdbc/OracleCMTDS1 DataSource` with the `scott` user, then both connections share the physical connection. In the following example, both `conn1` and `conn2` share the same physical connection.

```
Context ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("jdbc/OracleCMTDS1");
txn.begin(); //start txn
Connection conn1 = ds.getConnection("scott", "tiger");
Connection conn2 = ds.getConnection("scott", "tiger");
```

However, separate physical connections are retrieved for connections that are retrieved from separate `DataSource` objects. The following example shows that both `conn1` and `conn2` are retrieved from different `DataSource` objects: `jdbc/OracleCMTDS1` and `jdbc/OracleCMTDS2`. Both `conn1` and `conn2` will exist upon a separate physical connection.

```
Context ic = new InitialContext();
DataSource ds1 = (DataSource) ic.lookup("jdbc/OracleCMTDS1");
DataSource ds2 = (DataSource) ic.lookup("jdbc/OracleCMTDS2");
txn.begin(); //start txn
Connection conn1 = ds1.getConnection();
Connection conn2 = ds2.getConnection();
```

Connection Retrieval Error Conditions

The following mistakes can create an error condition:

- [Using Different User Names for Two Connections to a Single Data Source](#)
- [Improperly configured OCI JDBC Driver](#)

Using Different User Names for Two Connections to a Single Data Source

When you retrieve a connection from a `DataSource` object with a user name and password, this user name and password are used on all subsequent connection retrievals within the same transaction. This is true for all data source types.

For example, suppose an application retrieves a connection from the `jdbc/OracleCMTDS1` data source with the `scott` user name. When the application retrieves a second connection from the same data source with a different user name, such as `adams`, the second user name (`adams`) is ignored. Instead, the original user name (`scott`) is used.

```
Context ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("jdbc/OracleCMTDS1");
txn.begin(); //start txn
Connection conn1 = ds.getConnection("scott", "tiger"); //uses scott/tiger
Connection conn2 = ds.getConnection("adams", "woods"); //uses scott/tiger also
```

Thus, you cannot authenticate using two different users to the same data source. If you try to access the tables as `"adams/woods"`, you enter into an error condition.

Improperly configured OCI JDBC Driver

If you are using the OCI JDBC driver, ensure that you have configured it according to the recommendations in ["Using the OCI JDBC Drivers"](#) on page 4-22.

Using Two-Phase Commits and Data Sources

The Oracle two-phase commit coordinator is a DTC (distributed transaction coordinator) engine that performs two-phase commits with appropriate recovery. The two-phase commit engine is responsible for ensuring that when the transaction ends, all changes to all databases are either totally committed or fully rolled back. The two-phase commit engine can be one of the databases that participates in the global transaction, or it can be a separate database. If multiple databases or multiple sessions in the same database participate in a transaction, then you must specify a two-phase commit coordinator. Otherwise, you cannot commit the transaction.

Specify a commit coordinator in one of the following ways:

- Use the global `application.xml` in the `J2EE_HOME/config` directory to specify one commit coordinator for all applications.
- Override this commit coordinator for an individual application in the application's `orion-application.xml` file.

For example:

```
<commit-coordinator>
  <commit-class class="com.evermind.server.OracleTwoPhaseCommitDriver" />
  <property name="datasource" value="jdbc/OracleCommitDS" />
  <property name="username" value="system" />
  <property name="password" value="manager" />
</commit-coordinator>
```

Notes:

- The password attribute of the `<commit-coordinator>` element supports password indirection. For more information, see the section on password management in the *Oracle Application Server Containers for J2EE Security Guide*.
- Two-phase commits may be configured only for nonemulated data sources. For more information on data source types, see ["Types of Data Sources"](#) on page 4-1

If you specify a user name and password in the global `application.xml` file, then these values override the values in the `datasource.xml` file. If these values are null, then the user name and password in the `datasource.xml` file are used to connect to the commit coordinator.

The user name and password used to connect to the commit coordinator (for example, System) must have "force any transaction" privilege. By default, during installation, the `commit-coordinator` is specified in the global `application.xml` file, with the user name and password set as null.

Each data source participating in a two-phase commit should specify `dblink` information in the `OrionCMTDataSource` data source file. This `dblink` should be the name of the `dblink` that was created in the commit coordinator database to connect to this database.

For example, if `db1` is the database for the commit coordinator and `db2` and `db3` are participating in the global transactions, then you create `link2` and `link3` in the `db1` database as shown in the following example.

```
connect commit_user/commit_user
create database link link2 using "inst1_db2"; // link from db1 to db2
create database link link3 using "inst1_db3"; // link from db1 to db3;
```

Next, define a data source called `jdbc/OracleCommitDS` in the `application.xml` file:

```
<data-source
  class="com.evermind.sql.OrionCMTDataSource"
  name="OracleCommitDS"
  location="jdbc/OracleCommitDS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="system"
  password="manager"
  url="jdbc:oracle:thin:@//localhost:5521/db1.regress.rdbms.dev.us.oracle.com"
  inactivity-timeout="30"/>
```

Here is the data source description of `db2` that participates in the global transaction. Note that `link2`, which was created in `db1`, is specified as a property here:

```
<data-source
  class="com.evermind.sql.OrionCMTDataSource"
  name="OracleDB2"
  location="jdbc/OracleDB2"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="scott"
  password="tiger"
  url="jdbc:oracle:thin:@//localhost:5521/db2.regress.rdbms.dev.us.oracle.com"
  inactivity-timeout="30">
  <property name="dblink"
```



```

        value="LINK2.REGRESS.RDBMS.EXAMPLE.COM" />
</data-source>

```

Here is the data source description of db3 that participates in the global transaction. Note that link3, which is created in db1, is specified as a property here:

```

<data-source
  class="com.evermind.sql.OrionCMTDataSource"
  name="OracleDB3"
  location="jdbc/OracleDB3"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="scott"
  password="tiger"
  url="jdbc:oracle:thin:@//localhost:5521/db3.regress.rdbms.dev.us.oracle.com"
  inactivity-timeout="30">
  <property name="dblink"
    value="LINK3.REGRESS.RDBMS.EXAMPLE.COM" />
</data-source>

```

For information on the limitations of two-phase commit, see [Chapter 7, "Java Transaction API"](#).

Using Oracle JDBC Extensions

To use Oracle JDBC extensions, cast the returned connection to `oracle.jdbc.OracleConnection`, as follows:

```

Context ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("jdbc/OracleCMTDS1");
oracle.jdbc.OracleConnection conn =
    (oracle.jdbc.OracleConnection) ds.getConnection();

```

You can use any of the Oracle extensions on the returned connection, `conn`.

```

// you can create oracle.jdbc.* objects using this connection
oracle.jdbc.Statement orclStmt =
    (oracle.jdbc.OracleStatement) conn.createStatement();
// assume table is varray_table
oracle.jdbc.OracleResultSet rs =
    orclStmt.executeQuery("SELECT * FROM " + tableName);
while (rs.next())
{
    oracle.sql.ARRAY array = rs.getARRAY(1);
    ...
}

```

Using Connection Caching Schemes

You can define the connection caching scheme to use within the data source definition. There are three types of connection caching schemes: `DYNAMIC_SCHEME`, `FIXED_WAIT_SCHEME`, and `FIXED_RETURN_NULL_SCHEME`. For a description of these schemes, see the Connection Pooling and Caching chapter of the *Oracle9i JDBC Developer's Guide and Reference*, found at the following location:

<http://st-doc.us.oracle.com/9.0/920/java.920/a96654/toc.htm>

To specify a caching scheme, specify an integer or string value for a `<property>` element named `cacheScheme`. [Table 4-6](#) shows the supported values.

Table 4–6 Connection Caching Schemes

Value	Cache Scheme
1	DYNAMIC_SCHEME
2	FIXED_WAIT_SCHEME
3	FIXED_RETURN_NULL_SCHEME

Note: The connection cache scheme discussion in this section applies only to native data sources. It does not apply to any other data source.

The following example is a data source using the `DYNAMIC_SCHEME`.

```
<data-source
  class="oracle.jdbc.pool.OracleConnectionCacheImpl"
  name="OracleDS"
  location="jdbc/pool/OracleCache"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="scott"
  password="tiger"
  url="jdbc:oracle:thin:@//hostname:TTC port number/SERVICE"
  inactivity-timeout="30">
  <property name="cacheScheme" value="1" />
</data-source>
```

In this example, for the `<property name>` element, you could also specify **value="DYNAMIC_SCHEME"**.

When you create a data source in `data-sources.xml`, be aware of the following: When `class` is set to `oracle.jdbc.pool.OracleConnectionCacheImpl`, you must not specify the `ejb-location`, `xa-location`, and `pooled-location` attributes. Specify only the `location` attribute. Accessing the data source using any other attribute with JNDI causes unpredictable cleanup of cached connections in the event that the database goes down.

Using the OCI JDBC Drivers

The examples of Oracle data source definitions in this chapter use the Oracle JDBC Thin driver. However, you can use the Oracle JDBC OCI driver as well. Do the following before you start the OC4J server:

- Install the Oracle Client on the same system on which OC4J is installed.
- Set the `ORACLE_HOME` variable.
- Set `LD_LIBRARY_PATH` (or the equivalent environment variable for your OS) to `$ORACLE_HOME/lib`.
- Set `TNS_ADMIN` to a valid Oracle administration directory with a valid `tnsnames.ora` file.

The URL to use in the `url` attribute of the `<data-source>` element definition can have any of these forms:

- `jdbc:oracle:oci:@`

This TNS entry is for a database on the same system as the client, and the client connects to the database in IPC mode.

- `jdbc:oracle:oci:@TNS_service_name`
The TNS service name is an entry in the instance `tnsnames.ora` file.
- `jdbc:oracle:oci:@full_TNS_listener_description`
For more TNS information, see the *Oracle10i Net Services Administrator's Guide*

Notes on Oracle JDBC-OCI driver upgrade in the Oracle Application Server

It is not possible to upgrade to an arbitrary Oracle JDBC-OCI driver version due to client library compatibility constraints. Upgrading to OCI driver versions with matching Oracle Client libraries that are installed within the Oracle Application Server 10g (10.1.2) is supported. For example, Oracle JDBC 10.1.x drivers are supported, but the Oracle JDBC 9.2.x drivers are not.

Where the use of JDBC-OCI within the Oracle Application Server is supported, it is also necessary for the `opmn.xml` entry for each OC4J instance to propagate appropriate `ORACLE_HOME` and library path values to its startup environment.

The environment variable `ORACLE_HOME` is common to all platforms, but the name of the environment variable that specifies the library path is different depending on the operating systems:

- `LD_LIBRARY_PATH` for Solaris
- `SLIB_PATH` for AIX
- `SHLIB_PATH` for HP-UX
- `PATH` for Windows

The generic syntax for specifying the library paths in `opmn.xml` looks like this:

```
<prop name="<LIB_PATH_VARIABLE>" value="<LIB_PATH_VARIABLE_VALUE>" />
```

where `<LIB_PATH_VARIABLE>` should be replaced with the appropriate platform-specific variable name that specifies the library path, and

```
<LIB_PATH_VARIABLE_VALUE>
```

should be replaced with that variable's value.

Here is an example, assuming the Solaris OS:

```
<process-type id="OC4J_SECURITY" module-id="OC4J">
  <environment>
    <variable id="ORACLE_HOME"
value="/u01/app/oracle/product/inf10120" />
    <variable
      id="LD_LIBRARY_PATH"
      value="/u01/app/oracle/product/inf10120/lib"
    />
  </environment>
  ...
```

Using DataDirect JDBC Drivers

When your application must connect to heterogeneous databases, use DataDirect JDBC drivers. DataDirect JDBC drivers are not meant to be used with an Oracle database but for connecting to non-Oracle databases, such as Microsoft, SQLServer,

Sybase, and DB2. If you want to use DataDirect drivers with OC4J, then add corresponding entries for each database in the `data-sources.xml` file.

Installing and Setting Up DataDirect JDBC Drivers

Install the DataDirect JDBC drivers as described in the *DataDirect Connect for JDBC User's Guide and Reference*

After you have installed the drivers, follow these instructions to set them up.

Note: In the following instructions, note these definitions:

- `OC4J_INSTALL`: In a standalone OC4J environment, the directory into which you unzip the file `oc4j_extended.zip`. In an Oracle Application Server, `OC4J_INSTALL` is `ORACLE_HOME`.
 - In both a standalone OC4J environment and an Oracle Application Server, `DDJD_INSTALL` is the directory into which you unzip the content of the DataDirect JDBC drivers.
 - In a standalone OC4J environment, `INSTANCE_NAME` is `home`.
 - In an Oracle Application Server, `INSTANCE_NAME` is the OC4J instance into which you install the DataDirect JDBC drivers.
-
-

1. Unzip the content of the DataDirect JDBC drivers to the directory `DDJD_INSTALL`.
2. Create the directory `OC4J_INSTALL/j2ee/INSTANCE_NAME/applib` if it does not already exist.
3. Copy the DataDirect JDBC drivers in `DDJD_INSTALL/lib` to the `OC4J_INSTALL/j2ee/INSTANCE_NAME/applib` directory.
4. Verify that the file `application.xml` contains a library entry that references the `j2ee/home/applib` location, as follows:

```
<library path="../../INSTANCE_NAME/applib" />
```
5. Add data sources to the file `data-source.xml` as described in "[Example DataDirect Data Source Entries](#)" on page 4-24.

Example DataDirect Data Source Entries

This section shows an example data source entry for each of the following non-Oracle databases:

- [SQLServer](#)
- [DB2](#)
- [Sybase](#)

You can also use vendor-specific data sources in the class attribute directly. That is, it is not necessary to use an OC4J-specific data source in the class attribute.

For more detailed information, refer to the *DataDirect Connect for JDBC User's Guide and Reference*.

Note: OC4J does not work with non-Oracle data sources in the non-emulated case. That is, you cannot use a non-Oracle data source in a two-phase commit transaction.

SQLServer

The following is an example of a data source entry for SQLServer.

```
<data-source
  class="com.evermind.sql.DriverManagerDataSource"
  name="MerantDS"
  location="jdbc/MerantCoreSSDS"
  xa-location="jdbc/xa/MerantSSXADS"
  ejb-location="jdbc/MerantSSDS"
  connection-driver="com.oracle.ias.jdbc.sqlserver.SQLServerDriver"
  username="test"
  password="secret"
  url="jdbc:oracle:sqlserver://hostname:port;User=test;Password=secret"
  inactivity-timeout="30"
/>
```

DB2

Here is a data source configuration sample for a DB2 database:

```
<data-source
  class="com.evermind.sql.DriverManagerDataSource"
  name="MerantDS"
  location="jdbc/MerantDB2DS"
  xa-location="jdbc/xa/MerantDB2XADS"
  ejb-location="jdbc/MerantDB2DS"
  connection-driver="com.oracle.ias.jdbc.db2.DB2Driver"
  username="test"
  password="secret"
  url="jdbc:oracle:db2://hostname:port;LocationName=jdbc;CollectionId=default;"
  inactivity-timeout="30"
/>
```

Sybase

Here is a data source configuration sample for a Sybase database:

```
<data-source
  class="com.evermind.sql.DriverManagerDataSource"
  name="MerantDS"
  location="jdbc/MerantCoreSybaseDS"
  xa-location="jdbc/xa/MerantSybaseXADS"
  ejb-location="jdbc/MerantSybaseDS"
  connection-driver="com.oracle.ias.jdbc.sybase.SybaseDriver"
  username="test"
  password="secret"
  url="jdbc:oracle:sybase://hostname:port;User=test;Password=secret"
  inactivity-timeout="30"
/>
```

High Availability Support for Data Sources

This section discusses high availability (HA) support for data sources.

A high availability (HA) architecture must encompass redundancy across all components, achieve fast client failover for all types of outages, provide consistent high performance, and provide protection from user errors, corruptions, and site disasters, while being easy to deploy, manage, and scale.

Oracle Maximum Availability Architecture (MAA)

The Oracle Maximum Availability Architecture (MAA) provides recommendations and configuration instructions to help you choose and implement an Oracle platform availability architecture that best fits your availability requirements.

The main MAA recommendations are:

- Use redundant middle-tier or application tier (Oracle Application Server), network, and storage infrastructure.
- Use Oracle Data Guard to protect from human errors and data failures, and to recover from site failures.
- Use Real Application Clusters (RAC) at each site to protect from host and instance failures.
- Use sound operational best practices (such as fast-start check pointing to control the amount of time required to recover from an instance failure).

For more information about MAA, see

<http://www.oracle.com/technology/deploy/availability/htdocs/maa.htm>

Oracle Data Guard

Oracle Data Guard is software integrated with the Oracle database that maintains a real-time copy of a production database, called a standby database, and keeps this instance synchronized with its redundant mate. Oracle Data Guard manages the two databases using log transport services, managed recovery, switchover, and failover features.

Real Application Clusters (RAC)

RAC uses two or more nodes or computers, each running an Oracle instance that accesses a single database residing on shared-disk storage. In a RAC environment, all active instances can concurrently execute transactions against the shared database. RAC automatically coordinates each instance's access to the shared data to provide data consistency and data integrity.

RAC depends on two types of failover mechanisms:

- Network failover implemented in the network layer.
- Transparent Application Failover (TAF) implemented on top of the network layer.

Network Failover

Network failover is the default failover and is the only type of failover available when using the JDBC Thin driver. Network failure ensures that newer database connections created after a database instance in an RAC cluster goes down are created against a backup or surviving database instance in that cluster, even though the TNS alias that was used to create the newer database connection was for the database instance that

went down. When network failover is the only available failover mechanism, then existing connections are not automatically reconnected to surviving RAC instances. These existing connections are no longer usable, and you will get ORA-03113 exceptions if you try to use them. Ongoing database operations (including AQ operations) can fail with a wide variety of exceptions when failover occurs in a RAC cluster configured to perform only network failover.

TAF Failover

TAF failover is available only when using the JDBC OCI driver. To enable it, you must set the `FAILOVER_MODE` as part of the `CONNECT_DATA` portion of the TNS alias used to create the JDBC connection.

TAF is a runtime failover for high-availability environments, such as RAC and Data Guard, that refers to the failover and re-establishment of application-to-service connections. It enables client applications to automatically reconnect to the database if the connection fails, and optionally resume a `SELECT` statement in progress. This reconnect happens automatically from within the Oracle Call Interface (OCI) library.

TAF provides a best effort failover mechanism for ongoing operations on a database connection created against a database instance that is part of a RAC cluster. It also attempts to ensure that existing connections that are not in use at failover time are reconnected to a backup or surviving database instance. However, TAF is not always able to replay transactional operations that occur past the last committed transaction. When this happens, it usually throws an ORA-25408 ("cannot safely replay call") error. It is then your application's responsibility to explicitly roll back the current transaction before the database connection can be used again. Your application must also replay all the operations past the last committed transaction to return to the same state as that before the failover occurred.

TAF protects or fails over:

- Database connections
- User session states
- Prepared statements
- Active cursors (`SELECT` statements) that began returning results at the time of failure

TAF neither protects nor fails over:

- Applications not using OCI8 or higher
- Server-side program variables, such as PL/SQL package states
- Active Update transactions (see "[Acknowledging TAF Exceptions](#)" on page 4-31)

High Availability (HA) Support in OC4J

You can integrate OC4J with RAC, Data Guard, and TAF as part of your HA architecture.

The following sections describe configuration issues specific to OC4J that relate directly to HA. Use this information in conjunction with MAA recommendations and procedures.

The following sections discuss OC4J HA configuration issues:

- [Configuring Network Failover with OC4J](#)
- [Configuring Transparent Application Failover \(TAF\) with OC4J](#)

- [Connection Pooling](#)
- [Acknowledging TAF Exceptions](#)
- [SQL Exception Handling](#)

Configuring Network Failover with OC4J

To configure OC4J to use network failover:

1. Configure a network failover-enabled data source in `data-sources.xml`.

For example:

```
<data-source
  class="com.evermind.sql.DriverManagerDataSource"
  name="OracleDS"
  location="jdbc/OracleCoreDS"
  xa-location="jdbc/xa/OracleXADS"
  ejb-location="jdbc/OracleDS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="scott"
  password="tiger"
  url="jdbc:oracle:thin:@(DESCRIPTION=
    (LOAD_BALANCE=on)
    (ADDRESS=(PROTOCOL=TCP) (HOST=host1) (PORT=1521))
    (ADDRESS=(PROTOCOL=TCP) (HOST=host2) (PORT=1521))
    (CONNECT_DATA=(SERVICE_NAME=service_name)))"
  inactivity-timeout="300"
  connection-retry-interval="2"
  max-connect-attempts="60"
  max-connections="60"
  min-connections="12"
/>
```

In this example, note the `url` element. As long as two or more hosts are specified, the JDBC client randomly chooses one of the alternatives if the current host is unreachable.

For details on data source configuration, see "[Defining Data Sources](#)" on page 4-6.

Configuring Transparent Application Failover (TAF) with OC4J

To configure OC4J for use with TAF:

1. Configure a TAF descriptor as described in "[Configuring a TAF Descriptor \(tnsnames.ora\)](#)" on page 4-29.
2. Configure a TAF-enabled data source in `data-sources.xml`. For example:

```
<data-source
  class="com.evermind.sql.DriverManagerDataSource"
  name="OracleDS"
  location="jdbc/OracleCoreDS"
  xa-location="jdbc/xa/OracleXADS"
  ejb-location="jdbc/OracleDS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="scott"
  password="tiger"
  url="jdbc:oracle:oci8:@(description=(load_balance=on) (failover=on)
    (address=(protocol=tcp) (host=db-node1) (port=1521))
    (address=(protocol=tcp) (host=db-node2) (port=1521))
    (address=(protocol=tcp) (host=db-node3) (port=1521)))
```



```

        (address=(protocol=tcp) (host=db-node4) (port=1521))
        (connect_data=
          (service_name=db.us.oracle.com)
          (failover_mode=(type=select) (method=basic) (retries=20) (delay=15)))"
      rac-enabled="true"
      inactivity-timeout="300"
      connection-retry-interval="2"
      max-connect-attempts="60"
      max-connections="60"
      min-connections="12"
    />

```

In this example, note that the `url` element `failover` is on and `failover_mode` is defined. As long as two or more hosts are specified, the JDBC client randomly chooses one of the alternatives if the current host is unreachable. For a description of `failover_mode` options, see [Table 4-7, "TAF Configuration Options"](#) on page 4-29.

For details on data source configuration, see ["Defining Data Sources"](#) on page 4-6.

3. Configure Oracle JMS as the Resource Provider for JMS in the `orion-application.xml` file. For example:

```

<resource-provider
  class="oracle.jms.OjmsContext" name="cartojms1">
  <description> OJMS/AQ </description>
  <property name="datasource" value="jdbc/CartEmulatedDS"></property>
</resource-provider>

```

Note: Only data sources configured to use the JDBC OCI client can be configured for use with TAF.

Configuring a TAF Descriptor (`tnsnames.ora`)

TAF is configured using Net8 parameters in the `tnsnames.ora` file.

TAF can be configured by including a `FAILOVER_MODE` parameter under the `CONNECT_DATA` section of a connect descriptor. TAF supports the subparameters described in [Table 4-7](#).

Table 4-7 TAF Configuration Options

Subparameter	Description
BACKUP	Specify a different net service name for backup connections. A backup should be specified when using the <code>PRECONNECT METHOD</code> to pre-establish connections.
TYPE	Specify the type of fail over. Three types of Oracle Net failover functionality are available by default to Oracle Call Interface (OCI) applications: <ul style="list-style-type: none"> ■ SESSION: Set to failover the session. If a user's connection is lost, then a new session is automatically created for the user on the backup. This type of failover does not attempt to recover selects. ■ SELECT: Set to enable users with open cursors to continue fetching on them after failure. However, this mode involves overhead on the client side in normal select operations. ■ NONE: This is the default. No failover functionality is used. This can also be explicitly specified to prevent failover from happening.

Table 4–7 (Cont.) TAF Configuration Options

Subparameter	Description
METHOD	Determines how fast failover occurs from the primary node to the backup node: <ul style="list-style-type: none"> ▪ BASIC: Set to establish connections at failover time. This option requires almost no work on the backup server until failover time. ▪ PRECONNECT: Set to pre-established connections. This provides faster failover, but requires that the backup instance be able to support all connections from every supported instance.
RETRIES	Specify the number of times to attempt to connect after a failover. If DELAY is specified, RETRIES defaults to five retry attempts. Note: If a callback function is registered, then this subparameter is ignored.
DELAY	Specify the amount of time, in seconds, to wait between connect attempts. If RETRIES is specified, DELAY defaults to one second. Note: If a callback function is registered, then this subparameter is ignored.

In the following example, Oracle Net connects randomly to one of the protocol addresses on `sales1-server` or `sales2-server`. If the instance fails after the connection, then the TAF application fails over to the listener on another node.

```
sales.us.acme.com=
  (DESCRIPTION=
    (LOAD_BALANCE=on)
    (FAILOVER=on)
    (ADDRESS=(PROTOCOL=tcp)(HOST=sales1-server)(PORT=1521))
    (ADDRESS=(PROTOCOL=tcp)(HOST=sales2-server)(PORT=1521))
    (CONNECT_DATA=
      (SERVICE_NAME=sales.us.acme.com)
      (FAILOVER_MODE=
        (TYPE=session)
        (METHOD=basic)
        (RETRIES=20)
        (DELAY=15))))
```

For more information on configuring TAF, refer to the "*Oracle9i Net Services Administrator's Guide*".

Connection Pooling

If you have a transaction spanning two beans and each bean gets a JDBC connection to the same database but different instances, then on commit, OC4J issues a simple commit (instead of a Two-Phase Commit), which makes the transaction suspect. If your application encounters such transactions, use either TAF or connection pooling, but not both.

In case of an instance failure, dead connections are cleaned from both the OC4J connection pool and from the JDBC type 2 connection pool.

If a database goes down and `getConnection()` is called, and if connection pooling is used, then the pool is cleaned up. The caller must catch the exception on the `getConnection()` call, and retry. In some cases, the OC4J container does the retries.

OC4J cleans up a connection pool when the connection is detected to be bad. That is, if `getConnection()` throws a `SQLException` with error code 3113 or 3114.

When an exception occurs while using a user connection handle, it is useful for OC4J to detect whether the exception is due to a database connection error or to a database

operational error. The most common error codes thrown by the database when a connection error occurs are 3113 and 3114. These errors are returned typically for in-flight connections that get dropped. In addition, new connection attempts may receive error codes 1033, 1034, 1089 and 1090.

Fast-connection cleanup is implemented in both non-RAC and RAC environments.

In a non-RAC environment, when `ajava.sql.SQLException` is thrown, all unallocated connections are removed from the pool.

In a RAC environment, when `ajava.sql.SQLException` is thrown, first the states of all unallocated connections are checked. Connections that are alive are left alone. Otherwise, they are removed from the pool.

Acknowledging TAF Exceptions

Active Update transactions are rolled back at the time of failure because TAF cannot preserve active transactions after failover. TAF requires an acknowledgement from the application that a failure has occurred through a rollback command. In other words, the application receives an error message until a ROLLBACK is submitted.

A common failure scenario is as follows:

1. A JDBC Connection fails or is switched over by TAF.
2. TAF issues an exception.
3. TAF waits for an acknowledgement from the application in the form of a ROLLBACK.
4. The application rolls back the transaction and replays it.

Using Oracle Call Interface (OCI) callbacks and failover events, your application can customize TAF operations to automatically provide the required acknowledgement.

Your application (J2EE components) can capture the failure status of an Oracle instance and customize TAF by providing a function that the OCI library will automatically call during failover, using OCI callback capabilities. [Table 4–8](#) describes the failover events defined in the OCI API.

Table 4–8 OCI API Failover Events

Symbol	Value	Meaning
FO_BEGIN	1	A lost connection has been detected and failover is starting.
FO_END	2	A successful completion of failover.
FO_ABORT	3	An unsuccessful failover with no option of retrying.
FO_REAUTH	4	A user handle has been re-authenticated.
FO_ERROR	5	A failover was temporarily unsuccessful but the application has the opportunity to handle the error and retry.
FO_RETRY	6	Retry failover.
FO_EVENT_UNKNOWN	7	A bad or unknown failover event.

For more information, see the *Oracle Call Interface Programmer's Guide*.

SQL Exception Handling

Depending on the driver type used, SQL Exceptions will have different error codes and transaction replay may or may not be supported.

These error codes are obtained by making a `getErrorCode()` call on the `java.sql.SQLException` thrown to the caller.

[Table 4–9](#) summarizes these issues by driver type.

Table 4–9 SQL Exceptions and Driver Type

Driver	Error Code	Servlet Layer	Session Bean (CMT, BMT)	Entity Bean (CMP)
Thin JDBC	17410	Replay works.	Replay works (ignore "No _ activetransaction" error).	Replay not supported.
OCI	3113, 3114	Replay works.	Replay not supported.	Replay not supported.
OCI/TAF		After application sends acknowledgement to TAF (see "Acknowledging TAF Exceptions" on page 4-31), replay on surviving node works.	After application sends acknowledgement to TAF (see "Acknowledging TAF Exceptions" on page 4-31), replay on surviving node works.	If application sends acknowledgement to TAF (see "Acknowledging TAF Exceptions" on page 4-31), then OC4J proceeds transparently.

Oracle Remote Method Invocation

This chapter describes Oracle Application Server Containers for J2EE (OC4J) support for allowing EJBs to invoke one another across OC4J containers using the proprietary Remote Method Invocation (RMI)/Oracle RMI (ORMI) protocol.

This chapter covers the following topics:

- [Introduction to RMI/ORMI](#)
- [Configuring OC4J for RMI](#)
- [Configuring ORMI Tunneling through HTTP](#)

Introduction to RMI/ORMI

Java Remote Method Invocation (RMI) enables you to create distributed Java-based to Java-based applications, in which the methods of remote Java objects can be invoked from other Java virtual machines (JVMs), possibly on different hosts.

By default, OC4J EJBs exchange RMI calls over the Oracle Remote Method Invocation (ORMI) protocol, an Oracle proprietary protocol optimized for use with OC4J.

Alternatively, you can convert an EJB to use RMI/IIOP, making it possible for EJBs to invoke one another across different EJB containers as described in [Chapter 6, "J2EE Interoperability"](#).

Note: For the OC4J 10g Release 2 (10.1.2) implementation, load balancing and failover are supported only for ORMI, not IIOP.

ORMI Enhancements

ORMI is enhanced for OC4J and provides the following features:

- [Increased RMI Message Throughput](#)
- ["Enhanced Threading Support"](#)
- ["Co-Located Object Support"](#)

Increased RMI Message Throughput

Using ORMI, OC4J can process at a very high transaction rate. This is reflected in Oracle's SpecJ Application Server benchmarks at:

<http://www.spec.org/>

One way ORMI achieves this performance is by using messages that are much smaller than IIOP messages. Smaller messages take less bandwidth to send and receive, and less processing time to encode and decode. ORMI message size is further reduced by optimizing how much state information is exchanged between client and server. Using ORMI, some state is cached on the server so that it does not need to be transmitted in every RMI call. This does not violate the RMI requirement to be stateless because in the event of a failover, the client code resends all the state information required by the new server.

Enhanced Threading Support

ORMI is tightly coupled with the OC4J threading model to take full advantage of its queuing, pooling, and staging capabilities.

ORMI uses one thread per client. For multi-threaded clients, OC4J multiplexes each call through one connection. However, OC4J does not serialize them, so multiple threads do not block each other.

This feature ensures that each client (single-threaded or multi-threaded) has one connection to the remote server.

Co-Located Object Support

For co-located objects, RMI/ORMI detects the co-located scenario and avoids the extra, unnecessary socket call.

The same is true when the JNDI registry is co-located.

Client-Side Requirements

To access EJBs, do the following on the client-side:

1. Download the `oc4j_client.zip` file from
<http://www.oracle.com/technology/software/products/ias/devuse.html>
2. Unzip it into a client-side directory (for example, `d:\oc4jclient`)
3. Add `d:\oc4jclient\oc4jclient.jar` to your CLASSPATH

The `oc4j_client.zip` file contains all the JAR files required by the client (including `oc4jclient.jar` and `optic.jar`). These JAR files contain the classes necessary for client interaction. You must add only `oc4jclient.jar` to your CLASSPATH, because all other JAR files required by the client are referenced in the `oc4jclient.jar` manifest classpath.

If you download this file into a browser, then you must grant certain permissions as described in the "Granting Permissions" section of the Security chapter in the Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide.

Configuring OC4J for RMI

You can configure OC4J for RMI in one of two ways:

- [Configuring RMI Using Oracle Enterprise Manager 10g](#)
- [Configuring RMI Manually](#)

Oracle recommends that you configure OC4J using the Oracle Enterprise Manager 10g.

After OC4J is configured for RMI, you must specify RMI properties as described in "[RMI Configuration Files](#)" on page 5-7.

Configuring RMI Using Oracle Enterprise Manager 10g

Oracle recommends that you configure OC4J to use RMI by using Oracle Enterprise Manager 10g as follows:

1. Navigate to an OC4J instance in which you want to allow access to applications through RMI.

Figure 5–1 shows an OC4J instance called home.

Figure 5–1 Oracle Enterprise Manager 10g System Components

System Components

Enable/Disable Components Create OC4J Instance

Start Stop Restart Delete OC4J Instance

Select All | Select None

Select	Name	Status	Start Time	CPU Usage (%)	Memory Usage (MB)
<input type="checkbox"/>	home	↑	May 29, 2003 7:03:58 PM	Unavailable	94.92
<input type="checkbox"/>	HTTP_Server	↓			
<input type="checkbox"/>	JServ	↓			
<input type="checkbox"/>	OC4J_EM	↑	May 28, 2003 1:15:25 PM	0.05	157.68
<input type="checkbox"/>	OID	↓			
<input type="checkbox"/>	WebCache	↓			

TIP This table contains only the enabled components of the application server. Only components that have the checkbox enabled can be started or stopped.

Related Links: [Process Management](#)

1. Click the OC4J instance name.
2. Click the **Administration** tab.
3. Click **Server Properties**.
4. By default, RMI is disabled in an Oracle Application Server environment. To enable RMI, set a unique RMI port (or port range) for each OC4J instance by entering the value in the RMI Ports field, as shown in Figure 5–2.

Figure 5–2 Oracle Enterprise Manager 10g Server Properties Port Configuration

Islands

Island ID	Number of Processes
default_island	1

Add Another Row

Related Links: [Virtual Machine](#), [Metrics](#)

Ports

RMI Ports: 3101-3200

JMS Ports: 3201-3300

ΔJP Ports: 3000-3100

RMI-IIOP Ports

IIOP Ports: 3401-3500

IIOP SSL (Server only):

IIOP SSL (Server and Client):

1. Click **Apply**.
2. Click the **Back** button on your browser.
3. Click **Replication Properties**.
4. Check the **Replicate State** field as shown in Figure 5–3.

The remaining attributes on the **EJB Applications** screen are ignored if **Replicate State** is not checked.

Figure 5–3 Oracle Enterprise Manager 10g Replication Properties

EJB Applications

TIP EJB applications replicate state between all OC4J processes in the OC4J instance.

Replicate State

Multicast Host (IP)

Multicast Port

Username

Password

RMI Server Host

This is usually the name of the machine where the OC4J instance is running.

1. Configure the **RMI Server Host** field as shown in [Figure 5–3](#).

Enter a particular host name or IP address from which your server will accept RMI requests. The OC4J server accepts only RMI requests from this particular host.

Note: The other attributes on the Replication Properties window apply only to EJB clustering. For details, see the section "Configure the Multicast Address for EJB Clustering" in the *Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide*.

2. Click **Apply**.

Configuring RMI Manually

Oracle recommends that you configure OC4J using the Oracle Enterprise Manager 10g as described in "[Configuring RMI Using Oracle Enterprise Manager 10g](#)" on page 5-3. If you choose to manually configure RMI, you must:

1. Edit the property file `server.xml`. See "[Editing server.xml](#)" on page 5-5.
2. Choose the configuration files appropriate for your environment:
 - In an OC4J standalone environment, edit the `rmi.xml` file (see "[Editing rmi.xml](#)" on page 5-5) only.
 - In an Oracle Application Server environment, edit both the `rmi.xml` file (see "[Editing rmi.xml](#)" on page 5-5) and the `opmn.xml` file (see "[Editing opmn.xml](#)" on page 5-6).

Note: In an Oracle Application Server environment, `opmn` selects an RMI port for each OC4J instance from the range of RMI ports defined in the `opmn.xml` file (see "[Editing opmn.xml](#)" on page 5-6); the `rmi.xml` file `rmi-server` element `port` attribute is ignored.

Manual changes to configuration files in an Oracle Application Server environment are not applied until you synchronize the configuration repository by running the following on the Oracle Application Server command line: `dcmctl updateConfig`

Editing server.xml

Your `server.xml` file must specify the path name of the RMI configuration file in the `<rmi-config>` element. Here is the syntax:

```
<rmi-config path="RMI_PATH" />
```

The usual `RMI_PATH` is `./rmi.xml`; you can name the file whatever you like.

In an Oracle Application Server environment only, apply changes by running the following on the Oracle Application Server command line:

```
dcmctl updateConfig
```

Editing rmi.xml

Edit the `rmi.xml` file to specify which host, port, and user name and password to use to connect to (and accept connections from) remote RMI servers by configuring an `rmi-server` element.

To configure the `rmi.xml` file:

1. Add an `rmi-server` element for this local RMI server.

For example:

```
<rmi-server host="hostname" port="port">
</rmi-server>
```

Here are the user-replaceable attributes of the `<rmi-server>` element:

- `hostname`: the host name or IP address from which the RMI server accepts RMI requests. If you omit this attribute, the RMI server accepts RMI requests from any host.
- `port`: the port number on which the RMI server listens for RMI requests.

Note: In an OC4J standalone environment, if you omit this attribute, it defaults to 23791.

In an Oracle Application Server environment, `opmn` selects an RMI port for each OC4J instance from the range of RMI ports defined in the `opmn.xml` file (see "Editing `opmn.xml`" on page 5-6); the `rmi-server` element `port` attribute is ignored.

2. Configure the `rmi-server` element with zero or more `server` elements that each specify a remote (point-to-point) RMI server that your application can contact over RMI.

For example:

```
<rmi-server host="hostname" port="port">
  <server host="serverhostname" username="username" port="serverport"
    password="password" />
</rmi-server>
```

The `host` attribute is required; the remaining attributes are optional. Here are the user-replaceable attributes of the `server` element:

- `serverhostname`: the host name or IP address on which the remote RMI server listens for RMI requests
- `username`: the user name of a valid principal on the remote RMI server

- *serverport*: the port number on which the remote RMI server listens for RMI requests
 - *password*: the password used by the principal *username*
3. Configure the `rmi-server` element with zero or more `log` elements that each specify a file to which RMI-specific notifications are written.

For example, using the `file` element:

```
<rmi-server host="hostname" port="port">
  <log>
    <file path="logfilepathname" />
  </log>
</rmi-server>
```

Or using the `odl` element:

```
<rmi-server host="hostname" port="port">
  <log>
    <odl path="odlpathname" max-file-size="size" max-num-files="num"/>
  </log>
</rmi-server>
```

You can use either the `file` element or the `odl` element (but not both).

Here are the user-replaceable attributes of the `log` element:

- *odlpathname*: the path and folder name of the log folder for this area. You can use an absolute path or a path relative to the `J2EE_HOME/config` directory. This denotes where the RMI log files will reside.
- *size*: the maximum size in bytes of each individual log file.
- *num*: the maximum number of log files.
- *logfilepathname*: the path name of a log file (*logfilepathname*) to which the server writes all RMI requests.

The `<odl>` element is new in the OC4J 10g Release 2 (10.1.2) implementation. The ODL log entries are each written out in XML format in its respective log file. The log files have a maximum limit. When the limit is reached, the log files are overwritten.

When you enable ODL logging, each message goes into its respective log file, named `logN.xml`, where *N* is a number starting at 1. The first log message starts the log file `log1.xml`. When the log file size maximum is reached, the second log file, named `log2.xml`, is opened to continue the logging. When the last log file is full, the first log file, `log1.xml`, is erased and a new one is opened for the new messages. Thus, your log files are constantly rolling over and do not encroach on your disk space.

For more information about ODL logging, see the Oracle Application Server Containers for J2EE User's Guide.

4. In an Oracle Application Server environment only, apply changes by running the following on the Oracle Application Server command line:

```
dcmctl updateConfig
```

Editing `opmn.xml`

In an Oracle Application Server environment, edit the `opmn.xml` file to specify the port range on which this local RMI server listens for RMI requests.

To configure the `opmn.xml` file:

1. Configure the `rmi` port range using the `port id="rmi"` element as shown in the following example `opmn.xml` file excerpt:

```
<ias-component id="OC4J">
  <process-type id="home" module-id="OC4J">
    <port id="ajp" range="3301-3400" />
    <port id="rmi" range="3101-3200" />
    <port id="jms" range="3201-3300" />
    <process-set id="default-island" numprocs="1"/>
  </process-type>
</ias-component>
```

For more information on configuring the `opmn.xml` file, see the Oracle Application Server Administrator's Guide.

2. Apply changes by running the following on the Oracle Application Server command line:

```
dcmdctl updateConfig
```

RMI Configuration Files

Before EJBs can communicate, you must configure the parameters in the configuration files listed in [Table 5-1](#).

Table 5-1 RMI Configuration Files

Context	File	Description
Server	<code>server.xml</code>	The <code><sep-config></code> element in this file specifies the path name, normally <code>internal-settings.xml</code> , for the server extension provider properties. Example: <code><sep-config path="./internal-settings.xml"></code>
Application	<code>jndi.properties</code>	This file specifies the URL of the initial naming context used by the client. See " JNDI Properties for RMI " on page 5-7.

JNDI Properties for RMI

This section summarizes JNDI properties specific to RMI/ORMI. For details, see "Accessing the EJB" in the EJB Primer chapter in Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide.

The following RMI/ORMI properties are controlled by the `jndi.properties` file:

- `java.naming.provider.url` (see "[Naming Provider URL](#)" on page 5-7)
- `java.naming.factory.initial` (see "[Context Factory Usage](#)" on page 5-9)

Naming Provider URL

Use the following syntax to set the `java.naming.provider.url`:

```
<prefix>://<host>:<port>:<oc4j_instance>/<application-name>
```

[Table 5-2](#) describes arguments used in this syntax.

Table 5–2 Naming Provider URL

Variable	Description
prefix	<p>Use <code>opmn:ormi</code> for Oracle Application Server applications.</p> <p>Use <code>ormi</code> for standalone OC4J applications.</p> <p>Use <code>http:ormi</code> for applications that use HTTP tunneling (see "Configuring ORMI Tunneling through HTTP" on page 5-11).</p> <p>Use <code>corbaname</code> for applications that must interoperate with non-OC4J containers (see "The corbaname URL" on page 6-10).</p>
host	<p>For Oracle Application Server applications, the name of the OPMN host as defined in the <code>opmn.xml</code> file. Although OPMN is often located on the same node as the OC4J instance, specify the host name in case it is located on another machine.</p> <p>For standalone OC4J applications, the port number defined by the <code>rmi.xml</code> file <code>rmi-server</code> element <code>host</code> attribute.</p>
port	<p>In Oracle Application Server 10g Release 2 (10.1.2), when the <code>opmn:ormi</code> prefix is used, specify the <code>request</code> port on which the <code>opmn</code> process is listening, and the <code>opmn</code> process will forward RMI requests to the RMI port that it selected for the appropriate OC4J instance (see "Using the opmn Request Port" on page 5-8). If omitted, the default <code>request</code> port value 6003 is used.</p> <p>In Oracle Application Server implementations before 10g Release 2 (10.1.2), when the <code>ormi</code> prefix is used, you must specify the RMI port that <code>opmn</code> selected for your OC4J instance (see "Using opmnctl to Show the Selected RMI Port" on page 5-9).</p> <p>For standalone OC4J applications, when the <code>ormi</code> prefix is used, you must specify the port number defined by the <code>rmi.xml</code> file <code>rmi-server</code> element <code>port</code> attribute.</p> <p>For applications that use HTTP tunneling and use the <code>http:ormi</code> prefix, see "Configuring ORMI Tunneling through HTTP" on page 5-11 for information on what port to specify.</p> <p>For applications that must interoperate with non-OC4J containers and use the <code>corbaname</code> prefix, see "The corbaname URL" on page 6-10 for information on what port to specify.</p>
oc4j_instance	<p>For Oracle Application Server applications, the name of the OC4J instance as defined in the Enterprise Manager.</p> <p>For standalone OC4J applications, this is not applicable.</p>
application-name	The name of your application.

For example:

```
java.naming.provider.url=opmn:ormi://localhost:oc4j_inst1/ejbsamples
```

Using the opmn Request Port

In Oracle Application Server 10g Release 2 (10.1.2), you can specify the port defined for the `request` attribute of the `notification-server` element's `port` element configured in the `opmn.xml` file (default: 6003). When `opmn` receives an RMI request on its `request` port, it forwards the RMI request to the RMI port that it selected for the appropriate OC4J instance.

For example, consider the following `opmn.xml` file excerpt:

```
<notification-server>
  <port local="6100" remote="6200" request="6004"/>
  <log-file path="$ORACLE_HOME/opmn/logs/ons.log" level="4">
```

```

        rotation-size="1500000"/>
        <ssl enabled="true" wallet-file="$ORACLE_HOME/opmn/conf/ssl.wlt/default"/>
</notification-server>

```

In this example, the port defined for the request attribute of the notification-server element's port element is 6004, so you would use 6004 as the port in your JNDI naming provider URL.

For an example of how this URL is used, see ["OC4J in Oracle Application Server Since 9.0.4"](#) on page 5-11.

Using opmnctl to Show the Selected RMI Port To determine what RMI port has been selected by opmn for each OC4J instance, use the following command on the host on which opmn is running:

```
opmnctl status -l
```

This outputs a table of data with one row per OC4J instance.

For example (some columns are omitted for clarity):

```

Processes in Instance: core817.dsunrdb22.us.oracle.com
-----+-----+-----+ ... +-----
ias-component | process-type | pid | ... | ports
-----+-----+-----+ ... +-----
WebCache      | WebCacheAdmin | 28821 | ... | administration:4000
WebCache      | WebCache      | 28820 | ... |
statistics:4002,invalidation:4001,http:7777
OC4J          | home         | 2012 | ... | iiop:3401,jms:3701,rmi:3201,ajp:3000
HTTP_Server   | HTTP_Server  | 28818 | ... | http2:7200,http1:7778,http:7200
dcm-daemon    | dcm-daemon   | 28811 | ... | N/A
LogLoader     | logloaderd   | N/A  | ... | N/A

```

The ports column of this table lists the ports selected by opmn. For example:

```
iiop:3401,jms:3701,rmi:3201,ajp:3000
```

In this example, opmn has selected port 3201 for RMI on the OC4J instance, with process id 2012, so you would use 3201 as the port in your JNDI naming provider URL for this OC4J instance.

Context Factory Usage

The initial context factory creates the initial context class for the client.

Set the java.naming.factory.initial property to one of the following:

- com.evermind.server.ApplicationClientInitialContextFactory
- com.evermind.server.ApplicationInitialContextFactory
- com.evermind.server.RMIInitialContextFactory.

The ApplicationClientInitialContextFactory is used when looking up remote objects from standalone application clients. It uses the refs and ref-mappings found in application-client.xml and orion-application-client.xml. It is the default initial context factory when the initial context is instantiated in a Java application.

Note: If you access an EJB in an application from an EJB in a different application, then you cannot use the `RMIInitialContextFactory` object. In this scenario, you must use a parent-child relationship between these applications, and you must use the default initial context factory object.

The `RMIInitialContextFactory` is used when looking up remote objects between different containers using the ORMI protocol.

The type of initial context factory that you use depends on who the client is:

- If the client is a pure Java client outside the OC4J container, then use the `ApplicationClientInitialContextFactory` class.
- If the client is an EJB or servlet client within the OC4J container, then use the `ApplicationInitialContextFactory` class. This is the default class; thus, each time you create a new `InitialContext` without specifying an initial context factory class, your client uses the `ApplicationInitialContextFactory` class.
- If the client is an administrative class that is going to manipulate or traverse the JNDI tree, then use the `RMIInitialContextFactory` class.
- If the client is going to use DNS load balancing, then use the `RMIInitialContextFactory` class.

Example Lookups

This section provides examples of how to look up an EJB in:

- [OC4J Standalone](#)
- [OC4J in Oracle Application Server: Releases Before 9.0.4](#)
- [OC4J in Oracle Application Server Since 9.0.4](#)

OC4J Standalone

The following example shows how to look up an EJB called `MyCart` in the J2EE application `ejbsamples` deployed in a standalone OC4J instance. The application is located on a node named `localhost`, configured to listen on RMI port 23792:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.evermind.server.rmi.RMIInitialContextFactory");
env.put(Context.SECURITY_PRINCIPAL, "admin");
env.put(Context.SECURITY_CREDENTIALS, "welcome");
env.put(Context.PROVIDER_URL, "ormi://localhost:23792/ejbsamples");

Context context = new InitialContext(env);
Object homeObject = context.lookup("MyCart");
CartHome home = (CartHome) PortableRemoteObject.narrow(homeObject, CartHome.class);
```

OC4J in Oracle Application Server: Releases Before 9.0.4

In an OC4J instance in an Oracle Application Server environment, RMI ports are assigned dynamically, and `JAZNUserManager` is the default user manager.

In Oracle Application Server releases before 10g Release 2 (10.1.2), if you are accessing an EJB in Oracle Application Server, you have to know the RMI ports assigned by

opmn. If you have only one JVM for your OC4J instance, then you have to limit the port ranges for RMIs to a specific number, for example: 3101-3101.

The following example shows how to look up an EJB called `MyCart` in the J2EE application `ejbsamples` in an Oracle Application Server environment in releases before 10g Release 2 (10.1.2). The application is located on a node named `localhost` configured to listen on RMI port 3101:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.evermind.server.rmi.RMIInitialContextFactory");
env.put(Context.SECURITY_PRINCIPAL, "jazn.com/admin");
env.put(Context.SECURITY_CREDENTIALS, "welcome");
env.put(Context.PROVIDER_URL, "ormi://localhost:3101/ejbsamples");

Context context = new InitialContext(env);
Object homeObject = context.lookup("MyCart");
CartHome home = (CartHome)PortableRemoteObject.narrow(homeObject, CartHome.class);
```

OC4J in Oracle Application Server Since 9.0.4

You can use the following type of lookup in the URL to look up an EJB in an Oracle Application Server environment without needing to know the RMI port assigned to your OC4J instance.

The following example shows how to look up the EJB named `MyCart` in the J2EE application `ejbsamples` in an Oracle Application Server 10g Release 2 (10.1.2) environment. The EJB application is located on a node named `localhost`. The differences between this invocation and the standalone invocation are: the `opmn` prefix to `ormi`, the specification of the OC4J instance name `oc4j_inst1` to which the EJB application is deployed, and no requirement to specify the RMI port:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.evermind.server.rmi.RMIInitialContextFactory");
env.put(Context.SECURITY_PRINCIPAL, "jazn.com/admin");
env.put(Context.SECURITY_CREDENTIALS, "welcome");
env.put(Context.PROVIDER_URL, "opmn:ormi://localhost:oc4j_inst1/ejbsamples");

Context context = new InitialContext(env);
Object homeObject = context.lookup("MyCart");
CartHome home = (CartHome)PortableRemoteObject.narrow(homeObject, CartHome.class);
```

Configuring ORMI Tunneling through HTTP

When EJBs communicate across firewalls, they can use tunneling to transmit RMI across HTTP. This tunneling is supported only with RMI/ORMI; you cannot perform HTTP tunneling with RMI/IIOP.

To configure OC4J to support RMI tunneling, do the following:

1. Verify that the following entries are in `global-web-application.xml` (the default installation has these entries preconfigured):

```
<servlet>
  <servlet-name>rmi</servlet-name>
  <servlet-class>com.evermind.server.rmi.RMIHttpTunnelServlet
</servlet-class>
</servlet>
<servlet>
  <servlet-name>rmip</servlet-name>
  <servlet-class>com.evermind.server.rmi.RMIHttpTunnelProxyServlet
</servlet-class>
```

```
</servlet>
```

2. Modify the JNDI provider URL (see ["JNDI Properties for RMI"](#) on page 5-7). The JNDI provider URL for accessing the OC4J EJB server takes the form:

```
ormi://hostname:ormi_port/appName
```

- To direct tunneling requests to the home instance of OC4J in an Oracle Application Server or standalone environment, set the URL to:

```
http:ormi://hostname:http_port/appName
```

- To direct tunneling requests to an instance mapped in an OC4J mount point in an Oracle Application Server environment only, configure `oc4j_mount` (see ["Configuring an OC4J Mount Point"](#) on page 5-12) and set the URL to:

```
http:ormi://hostname:http_port/appName@oc4j_mount
```

Note: `http_port` is the HTTP port, *not* the ORMI port (if omitted, it defaults to 80), and `appName` is the name of the application, *not* the application context defined in `web-site.xml`.

3. If your HTTP traffic goes through a proxy server, specify the `proxyHost` and (optionally) `proxyPort` in the command line that starts the EJB client:

```
-Dhttp.proxyHost=proxy_host -Dhttp.proxyPort=proxy_port
```

Note: If omitted, `proxy_port` defaults to 80.

Configuring an OC4J Mount Point

An OC4J mount point is used to direct tunneling requests to an OC4J instance other than the home instance of OC4J. To route incoming requests, Oracle HTTP Server (OHS) utilizes a list of application-specific mount points that map the URL supplied in a request with the OC4J instance that will service the request. The list of application mount points is defined in the `mod_oc4j` configuration file, `mod_oc4j.conf`. Mount points are typically defined at the time an application is deployed.

Note: For this to work, you must have a web module deployed or you must use a valid context root in the context instance. Using a placeholder web module has the advantage of removing any dependency on an existing web module that might be removed or modified, thereby affecting the client connection.

Here is an example of such a mapping:

```
Oc4jMount /xyz inst1  
Oc4jMount /xyz/* inst1
```

In this example, the OC4J instance `inst1` receives all requests with URLs that start with `/xyz`.

The first part of ["Configuring ORMI Tunneling through HTTP"](#) on page 5-11 shows the following URL:

```
http:ormi://hostname:http_port/appName@oc4j_mount
```


In this URL:

- `http_port` is the HTTP port, *not* the ORMI port.
- `appName` is the name of the application (defined by `default-web-site.xml` attribute name). It is not the application context (defined by `default-web-site.xml` attribute `application`).

Make sure that the context root for `appName` in the `default-web-site.xml` file is the same as that in the `mod-oc4j.conf` file. In this example, the application name is `demoApp` and its context root is `/xyz`. The actual line in the `default-web-site.xml` file for this application would be:

```
<default-web-app application="default" name="demoApp" root="/xyz" />
```

- `oc4j_mount` is the OC4J instance to which requests are to be routed.

Therefore, in this example, to direct tunneling requests from `defaultWebApp` to OC4J instance `inst1`, the URL would be:

```
http:ormi://hostname:http_port/defaultWebApp@xyz
```

The `mod-oc4j.conf` file is a component of the Oracle HTTP server. For more information, see the *Oracle HTTP Server Administrator's Guide*.

The `default-web-site.xml` file is an OC4J configuration file. For more information, see the *Oracle Application Server Containers for J2EE Servlet Developer's Guide*.

Caution.: When you change these configuration files manually, the changes do not take effect until you restart the OC4J instance. If you make the changes in the Application Server Control Console, then the changes take effect immediately. In this case, it is not necessary to restart the OC4J instance.

Configuring ORMI over HTTP in a Clustered Application

Here are the steps used to configure ORMI over HTTP in a clustered application in conjunction with the information already provided in this chapter.

1. Create a placeholder web application. This only needs to be a WAR file that contains a `WEB-INF` directory with a basic `web.xml` file. No actual JSP, servlet, or other files are needed.

2. Add the empty `distributable` tag to the `web.xml` file. For example,

```
<description> A placeholder web.xml file to be used to implement HTTP tunnel
ORMI connection for the EJB application.</description>
  <distributable />
  ...
```

3. Include the web module in the `application.xml` file.

```
<module>
  <web>
    <web-uri>placeholder.war</web-uri>
    <context-root>/tunnel</context-root>
  </web.>
</module>
```

This assumes that the WAR file is `placeholder.war` and the context root is `/tunnel`.

4. Configure the OC4J instances for replication as documented in the Oracle Application Server High Availability Guide.

Configure both the Web Application and the EJB Application section.

5. Package and deploy your new EAR file with the placeholder WAR file.

If using a manually-configured cluster, then you must deploy to each OC4J instance.

6. Make sure the client uses the following URL:

```
http:ormi://host.domain:http_port/<appname>@<contextroot>
```

where `<contextroot>` is the tunnel for this example and `<appname>` is the name provide when you deploy the EAR file for this application.

Failover will now work as expected.

Note:

If the application already contains a web module, then you can configure this module to be distributable and use its context root using these same steps. However, this method will prevent any changes to the web module from affecting the client configuration. For example, if the developer decides to change the context root you must also implement the change in the client connection information.

J2EE Interoperability

This chapter describes Oracle Application Server Containers for J2EE (OC4J) support for allowing EJBs to invoke one another across different containers using the standard Remote Method Invocation (RMI)/Internet Inter-Orb Protocol (IIOP).

This chapter covers the following topics:

- [Introduction to RMI/IIOP](#)
- [Switching to Interoperable Transport](#)
- [Configuring OC4J for Interoperability](#)

Introduction to RMI/IIOP

Java Remote Method Invocation (RMI) enables you to create distributed Java-based to Java-based applications, in which the methods of remote Java objects can be invoked from other Java virtual machines (JVMs), possibly on different hosts.

Version 2.0 of the EJB specification adds features that make it easy for EJB-based applications to invoke one another across different containers. You can make your existing EJB interoperable without changing a line of code: simply edit the bean's properties and redeploy. "[Switching to Interoperable Transport](#)" on page 6-3 discusses redeployment details.

EJB interoperability consists of the following:

- Transport interoperability, through CORBA IIOP (Internet Inter-ORB Protocol, where ORB is Object Request Broker)
- Naming interoperability, through the CORBA CosNaming Service (CORBA Object Service Naming, part of the OMG CORBA Object Service specification)
- Security interoperability, through Common Secure Interoperability Version 2 (CSIv2)
- Transaction interoperability, through the CORBA Transaction Service (OTS)

OC4J furnishes transport, naming, and security interoperability.

Transport

By default, OC4J EJBs use RMI/Oracle Remote Method Invocation (ORMI), a proprietary protocol, to communicate as described in [Chapter 5, "Oracle Remote Method Invocation"](#).

Note: For the OC4J 10g Release 2 (10.1.2) implementation, load balancing and failover are supported only for ORMI, not IIOP.

In OC4J, you can easily convert an EJB to use RMI/IIOP, making it possible for EJBs to invoke one another across different EJB containers. This chapter describes configuring and using RMI/IIOP.

Naming

OC4J supports the CORBA CosNaming service. OC4J can publish EJBHome object references in a CosNaming service and provides a JNDI CosNaming implementation that allows applications to look up JNDI names using CORBA. You can write your applications using either the JNDI or CosNaming APIs.

Security

OC4J supports Common Secure Interoperability Version 2 (CSIv2), which specifies different conformance levels; OC4J complies with the EJB specification, which requires conformance level 0.

Transactions

The EJB2.0 specification stipulates an optional transactional interoperability feature. Conforming implementations must choose one of the following:

- Transactionally interoperable: transactions are supported between beans that are hosted in different J2EE containers.
- Transactionally noninteroperable: transactions are supported only among beans in the same container.

The current release of OC4J is transactionally noninteroperable, so when a transaction spans EJB containers, OC4J raises a specified exception.

Client-Side Requirements

To access EJBs, you must do the following on the client-side:

1. Download the `oc4j_client.zip` file from
<http://www.oracle.com/technology/software/products/ias/devuse.html>
2. Unzip it into a client-side directory. (for example, `d:\oc4jclient`).
3. Add `d:\oc4jclient\oc4jclient.jar` to your CLASSPATH.

The `oc4j_client.zip` file contains all the JAR files required by the client (including `oc4jclient.jar` and `optic.jar`). These JAR files contain the classes necessary for client interaction. You need to add only `oc4jclient.jar` to your CLASSPATH, because all other JAR files required by the client are referenced in the `oc4jclient.jar` manifest classpath.

If you download this file into a browser, you must grant certain permissions as described in the "Granting Permissions" section of the Security chapter in the Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide.

The rmic.jar Compiler

To invoke or be invoked by CORBA objects, RMI objects must have corresponding stubs, skeletons, and Internet Description Language (IDL). Use the `rmic.jar` compiler to generate stubs and skeletons from Java classes or to generate IDL, as described in "[Configuring OC4J for RMI](#)" on page 5-2.

For use with RMI/IIOP, be sure to compile using the `-iiop` option.

Switching to Interoperable Transport

In OC4J, EJBs use RMI/ORMI, a proprietary protocol, to communicate (as described in [Chapter 5, "Oracle Remote Method Invocation"](#)). You can convert an EJB to use RMI/IIOP, making it possible for EJBs to invoke one another across EJB containers.

Note: RMI/IIOP support is based on the CORBA 2.3.1 specification. Applications that were compiled using earlier releases of CORBA may not work correctly.

The following four sections provide details on making the conversions.

Simple Interoperability in a Standalone Environment

Follow these steps to convert an EJB to use RMI/IIOP in a standalone environment:

1. Restart OC4J with the `-DGenerateIIOP=true` flag.
2. Deploy your application using `admin.jar`. You must obtain the client's stub JAR file, using the `-iiopClientJar` switch. Here is an example:

```
java -jar $J2EE_HOME/admin.jar ormi://localhost admin welcome -deploy -file
filename
-deploymentName application_name -iiopClientJar stub_jar_filename
```

Note: You must use the `-iiopClientJar` switch to enable interoperability (IIOP) for the application you are deploying. In OC4J, interoperability is enabled on a per-application basis.

3. Change the client's `classpath` to include the stub JAR file that was obtained during deployment, by running `admin.jar` with the `-iiopClientJar` switch.

A copy of the stub JAR file that was generated by OC4J can also be found in the server's deployment directory at:

```
application_deployment_directory/appname/ejb_module/module_iiopClient.jar
```

4. Edit the client's JNDI property `java.naming.provider.url` to use a `corbaname` URL instead of an `ormi` URL. For details on the `corbaname` URL, see "[The corbaname URL](#)" on page 6-10.

Note: IIOP stub and tie class code generation occurs at deployment time, unlike ORMI stub generation, which occurs at runtime. This is why you must add the JAR file to the `classpath` yourself. If you run in the server, a list of generated classes required by the server and IIOP stubs is made available automatically.

5. (Optional) To make the bean accessible to CORBA applications, run `rmic.jar` to generate IDL describing its interfaces. See "[Configuring OC4J for Interoperability](#)" on page 6-12 for a discussion of command-line options.

Advanced Interoperability in a Standalone Environment

This section expands upon the preceding section, describing how to convert an EJB to use RMI/IIOP in a standalone environment.

1. Specify CSIv2 security policies for the bean in `orion_ejb_jar.xml` and in `internal_settings.xml`. See the *Oracle Application Server Containers for J2EE Security Guide* for more information on these security-related topics.
2. Restart OC4J with the `-DGenerateIIOP=true` flag.
3. Deploy your application using `admin.jar`. You must obtain the client's stub JAR file using the `-iiopClientJar` switch. Here is an example:

```
java -jar $J2EE_HOME/admin.jar ormi://localhost admin welcome -deploy -file
filename
-deployment_name application_name -iiopClientJar stub_jar_filename
```

Note: You must use the `-iiopClientJar` switch to enable interoperability (IIOP) for the application that you are deploying. In OC4J, interoperability is enabled on a per-application basis.

4. Change the client's `classpath` to include the stub JAR file that was obtained during deployment, by running `admin.jar` with the `-iiopClientJar` switch.

You can also find a copy of the stub JAR file that was generated by OC4J in the server's deployment directory at:

```
application_deployment_directory/appname/ejb_module/module_iiopClient.jar
```

5. Edit the client's JNDI property `java.naming.provider.url` to use a `corbaname` URL instead of an `ormi` URL. For details on the `corbaname` URL, see "[The corbaname URL](#)" on page 6-10.

Note: IIOP stub and tie class code generation occurs at deployment time, unlike ORMI stub generation, which occurs at runtime. This is why you must add the JAR file to the `classpath` yourself. If you run in the server, a list of generated classes required by the server and IIOP stubs is made available automatically.

6. (Optional) To make the bean accessible to CORBA applications, run `rmic.jar` to generate IDL describing its interfaces. See "[Configuring OC4J for Interoperability](#)" on page 6-12 for a discussion of command-line options.

Simple Interoperability in Oracle Application Server Environment

You can access an EJB using RMI/IIOP in an Oracle Application Server environment in two ways:

- [Configuring for Interoperability Using Oracle Enterprise Manager 10g](#)
- [Configuring for Interoperability Manually](#)

Configuring for Interoperability Using Oracle Enterprise Manager 10g

You can configure an EJB to be accessible by means of RMI/IIOP in an Oracle Application Server environment by using Oracle Enterprise Manager 10g. Follow these steps:

1. Navigate to an OC4J instance in which you want to allow access to applications through RMI/IIOP. [Figure 6–1](#) shows an OC4J instance named home.

Figure 6–1 Oracle Enterprise Manager 10g System Components

System Components

[Enable/Disable Components](#) [Create OC4J Instance](#)
[Start](#) [Stop](#) [Restart](#) [Delete OC4J Instance](#)

Select All | Select None

Select	Name	Status	Start Time	CPU Usage (%)	Memory Usage (MB)
<input type="checkbox"/>	home	↑	May 29, 2003 7:03:58 PM	Unavailable	94.92
<input type="checkbox"/>	HTTP_Server	↓			
<input type="checkbox"/>	JServ	↓			
<input type="checkbox"/>	OC4J_EM	↑	May 28, 2003 1:15:25 PM	0.05	157.68
<input type="checkbox"/>	OID	↓			
<input type="checkbox"/>	WebCache	↓			

TIP This table contains only the enabled components of the application server. Only components that have the checkbox enabled can be started or stopped.

Related Links [Process Management](#)

1. Click Server Properties in the Administration section for this OC4J instance. [Figure 6–2](#) illustrates this.

Figure 6–2 Oracle Enterprise Manager 10g Server Properties

Applications

[Deploy EAR file](#) [Deploy WAR file](#)

Select	Name	Path	Parent Application	Active Requests	Request Processing Time (seconds)	Active EJB Methods
No applications deployed						

Administration

<p>Instance Properties</p> <ul style="list-style-type: none"> Server Properties Website Properties JSP Container Properties Replication Properties Advanced Properties 	<p>Application Defaults</p> <ul style="list-style-type: none"> Data Sources Security JMS Providers Global Web Module
--	---

By default, RMI/IIOP is disabled in an Oracle Application Server environment. To enable RMI/IIOP, ensure that a unique IIOP port (or port range) exists for each OC4J

instance by entering the value in the IIOP ports field, as shown in [Figure 6–3](#). Then click Apply.

Figure 6–3 Oracle Enterprise Manager 10g Port Configuration

Islands		Related Links
Island ID	Number of Processes	Virtual Machine Metrics
default_island	1	
Add Another Row		

Ports	
RMI Ports	3101-3200
JMS Ports	3201-3300
AJP Ports	3000-3100

RMI-IIOP Ports	
IIOP Ports	3401-3500
IIOP SSL (Server only)	
IIOP SSL (Server and Client)	

Use the Oracle Enterprise Manager 10g deployment wizard to deploy your application.

Enable generation of client IIOP stubs for this application by selecting Generate IIOP stubs, as shown in [Figure 6–4](#).

Figure 6–4 Oracle Enterprise Manager 10g Stub Generation

ORACLE
Enterprise Manager for Oracle9iAS

URL Mappings for Web Modules IIOP Stub Generation User Manager Security Role Mappings Review

Deploy Application: IIOP Stub Generation

This application contains EJB's. Please confirm if you wish to generate IIOP stubs.

Generate IIOP Stubs

Cancel Back Step 2 of 5 Next Finish

Copyright © 1996, 2003, Oracle. All rights reserved.
About Oracle Enterprise Manager Oracle9iAS Console

Finish deploying your application using the Oracle Enterprise Manager 10g deployment wizard.

Configuring for Interoperability Manually

Follow these steps to manually configure an EJB for remote access by RMI/IIOP in an Oracle Application Server environment:

1. By default, RMI/IIOP is disabled in an Oracle Application Server environment. To enable RMI/IIOP, confirm in OPMN's configuration file, `J2EE_HOME/opmn/conf/opmn.xml`, that a unique IIOP port (or port range) exists for each OC4J instance to be managed by OPMN.

Note: You must specify an IIOP port (or port range) for each OC4J instance in which interoperability is to be enabled. Failure to do so causes OC4J to not configure an IIOP listener, thus automatically disabling interoperability, regardless of the configuration in the `internal-settings.xml` file of OC4J.

Here is an example:

```
<ias-component id="OC4J">
  <process-type id="home" module-id="OC4J">
    <port id="ajp" range="3000-3100"/>
    <port id="rmi" range="23791-23799"/>
    <port id="jms" range="3201-3300"/>
    <port id="iiop" range="3401-3500"/>
    <process-set id="default_island" numprocs="1"/>
  </process-type>
</ias-component>
```

2. If you modify any configuration file manually, you must update the configuration with `dcmctl`. Use the following command:

```
dcmctl updateConfig
```

3. Using `opmnctl` or Oracle Enterprise Manager 10g, restart all OC4J instances that are managed by OPMN.

For information on `opmnctl`, use the following command:

```
opmnctl help
```

To stop and restart OPMN and all OPMN-managed processes, first use the following command:

```
opmnctl stopall
```

Then use this command:

```
opmnctl startall
```

For information on Oracle Enterprise Manager 10g, see the *Oracle Application Server Containers for J2EE User's Guide*.

4. Deploy your application using `dcmctl`, specifying the `-enableIIOP` option. Here is an example:

```
dcmctl deployApplication -f filename -a application_name -enableIIOP
```

5. Change the client's `classpath` to include the stub JAR file that was generated by OC4J. This file is normally found in the server's deployment directory:

```
application_deployment_directory/appname/ejb_module/module_iiopClient.jar
```

6. Edit the client's JNDI property `java.naming.provider.url` to use an OPMN or corbaname URL instead of an ormi URL. For details on the corbaname URL, see "[The corbaname URL](#)" on page 6-10. For details on the OPMN URL, see "[The OPMN URL](#)" on page 6-10.

Note: IIOP stub and tie class code generation occurs at deployment time, unlike ORMI stub generation, which occurs at runtime. This is why you must add the JAR file to the `classpath` yourself. If you run in the server, a list of generated classes required by the server and IIOP stubs is made available automatically.

- (Optional) To make the bean accessible to CORBA applications, run `rmic.jar` to generate IDL describing its interfaces. See "[Configuring OC4J for Interoperability](#)" on page 6-12 for a discussion of command-line options.

Advanced Interoperability in Oracle Application Server Environment

You can access an EJB using RMI/IIOP in an Oracle Application Server environment in two ways:


- Configuring for Interoperability Using Oracle Enterprise Manager 10g
- Configuring for Interoperability Manually

Configuring for Interoperability Using Oracle Enterprise Manager 10g

The advanced configuring for interoperability using Oracle Enterprise Manager 10g differs from the simple configuring described under "[Configuring for Interoperability Using Oracle Enterprise Manager 10g](#)" on page 6-5 only in the specification of ports. That is, you must specify an `iiop`, `iiops1`, and `iiops2` port (or port range) for each OC4J instance in which interoperability with CSIv2 is to be enabled. Failure to do so causes OC4J to not configure an IIOP listener, thus automatically disabling interoperability, regardless of the configuration in the `internal-settings.xml` file of OC4J. [Figure 6-5](#) shows this.

Figure 6-5 Oracle Enterprise Manager 10g Port Specifications

Multiple VM Configuration

 **TIP** If OC4J is running, newly added islands and associated processes will be automatically started.

Islands		Related Links	Virtual Machine Metrics
Island ID	Number of Processes		
default_island	1		
Add Another Row			

Ports

RMI Ports	3101-3200
JMS Ports	3201-3300
ΔJP Ports	3000-3100

RMI-IIOP Ports

IIOP Ports	3401-3500
IIOP SSL (Server only)	3501-3600
IIOP SSL (Server and Client)	3601-3700

Configuring for Interoperability Manually

This section expands upon the preceding section, describing how to convert an EJB to use RMI/IIOP in an Oracle Application Server environment.

- Specify CSIv2 security policies for the bean in `orion_ejb_jar.xml` and in `internal_settings.xml`. See the *Oracle Application Server Containers for J2EE Security Guide* for more information on these security-related topics.

- By default, RMI/IIOP is disabled in an Oracle Application Server environment. To enable RMI/IIOP, confirm in the OPMN configuration file, `J2EE_HOME/opmn/conf/opmn.xml`, that a unique `iiop`, `iiops1`, and `iiops2` port (or port range) exists for each OC4J instance to be managed by OPMN. These are the port meanings:

`iiop`—standard IIOP port

`iiops1`—IIOP/SSL port used for server-side authentication only

`iiops2`—IIOP/SSL port used for both client and server authentication

Note: You must specify an `iiop`, `iiops1`, and `iiops2` port (or port range) for each OC4J instance in which interoperability with CSIv2 is to be enabled. Failure to do so causes OC4J to not configure an IIOP listener, thus automatically disabling interoperability, regardless of the configuration in the `internal-settings.xml` file of OC4J.

Here is an example:

```
<ias-component id="OC4J">
  <process-type id="home" module-id="OC4J">
    <port id="ajp" range="3000-3100"/>
    <port id="rmi" range="23791-23799"/>
    <port id="jms" range="3201-3300"/>
    <port id="iiop" range="3401-3500"/>
    <port id="iiops1" range="3501-3600"/>
    <port id="iiops2" range="3601-3700"/>
    <process-set id="default_island" numprocs="1"/>
  </process-type>
</ias-component>
```

Note: If you choose to configure your client's JNDI property `java.naming.provider.url` to use an OPMN URL, then your client cannot connect to `iiops1` or `iiops2` ports because OPMN-allocated ports are not reported to OC4J.

- Using `opmnctl` or Oracle Enterprise Manager 10g, restart all OC4J instances that are managed by OPMN.

For information on `opmnctl`, use the following command:

```
opmnctl help
```

To stop and restart OPMN and all OPMN-managed processes, first use the following command:

```
opmnctl stopall
```

Then use this command:

```
opmnctl startall
```

For information on Oracle Enterprise Manager 10g, see the *Oracle Application Server Containers for J2EE User's Guide*.

4. Deploy your application using `dcmctl`, specifying the `-enableIIOP` option. Here is an example:

```
dcmctl deployApplication -f filename -a application_name -enableIIOP
```

5. Change the client's `classpath` to include the stub JAR file that was generated by OC4J. This is normally found in the server's deployment directory:

```
application_deployment_directory/appname/ejb_module/module_iiopClient.jar
```

6. Edit the client's JNDI property `java.naming.provider.url` to use an OPMN or corbaname URL instead of an ormi URL. For details on the corbaname URL, see "[The corbaname URL](#)" on page 6-10. For details on the OPMN URL, see "[The OPMN URL](#)" on page 6-10.

Note: IIOP stub and tie class code generation occurs at deployment time, unlike ORMI stub generation, which occurs at runtime. This is why you must add the JAR file to the `classpath` yourself. If you run in the server, a list of generated classes required by the server and IIOP stubs is made available automatically.

7. (Optional) To make the bean accessible to CORBA applications, run `rmic.jar` to generate IDL describing its interfaces. See "[Configuring OC4J for Interoperability](#)" on page 6-12 for a discussion of command-line options.

The corbaname URL

To interoperate, an EJB must look up other beans using `CosNaming`. This means that the URL for looking up the root `NamingContext` must use the corbaname URL scheme instead of the ormi URL scheme. This section discusses the corbaname subset that EJB developers use most often. For a full discussion of the corbaname scheme, see section 2.5.3 of the CORBA Naming Service specification. The corbaname scheme is based on the corbaloc scheme, which section 13.6.10.1 of the CORBA specification discusses.

The most common form of the corbaname URL scheme is:

```
corbaname::host[:port]
```

This corbaname URL specifies a conventional DNS host name or IP address, and a port number. For example,

```
corbaname::example.com:8000
```

A corbaname URL can also specify a naming context by following the host and port by `#` and `NamingContext` in string representation. The `CosNaming` service on the specified host is responsible for interpreting the naming context.

```
corbaname::host[:port]#namingcontext
```

For example:

```
corbaname::example.com:8000#Myapp
```

The OPMN URL

This section describes OPMN URL details that are specific to RMI/IIOP. For general information about the OPMN URL, see "[JNDI Properties for RMI](#)" on page 5-7.

In an Oracle Application Server environment, IIOP ports for all OC4J processes within each Oracle Application Server instance are dynamically managed by OPMN. Because of this, it may not be possible for clients to know the ports on which OC4J processes are actively listening for IIOP requests. To enable clients to successfully make RMI/IIOP requests in an Oracle Application Server environment without having to know the IIOP ports for all active OC4J processes, modify the `jndi.naming.provider.url` property (in the client's `jndi.properties` file) with a URL of the following format:

```
opmn:corbaname::opmn_host[:opmn_port]:]:OC4J_instance_name#naming_context
```

For example:

```
opmn:corbaname::dlsun74:6003:home#stateless
```

Notes:

- For the OC4J 10g Release 2 (10.1.2) implementation, load balancing and failover are supported only for ORMI, not IIOP.
 - If you use an OPMN URL, your client cannot connect to `iiops1` or `iiops2` (`ssl-port` or `ssl-client-server-auth-port`) ports.
-
-

Exception Mapping

When EJBs are invoked over IIOP, OC4J must map system exceptions to CORBA exceptions. [Table 6–1](#) lists the exception mappings.

Table 6–1 Java-CORBA Exception Mappings

OC4J System Exception	CORBA System Exception
<code>javax.transaction.TransactionRolledbackException</code>	TRANSACTION_ROLLEDBACK
<code>javax.transaction.TransactionRequiredException</code>	TRANSACTION_REQUIRED
<code>javax.transaction.InvalidTransactionException</code>	INVALID_TRANSACTION
<code>java.rmi.NoSuchObjectException</code>	OBJECT_NOT_EXIST
<code>java.rmi.AccessException</code>	NO_PERMISSION
<code>java.rmi.MarshalException</code>	MARSHAL
<code>java.rmi.RemoteException</code>	UNKNOWN

Invoking OC4J-Hosted Beans from a Non-OC4J Container

EJBs that are not hosted in OC4J must add the file `oc4j_interop.jar` to the classpath to invoke OC4J-hosted EJBs. OC4J expects the other container to make the `HandleDelegate` object available in the JNDI name space at `java:comp/HandleDelegate`. The `oc4j_interop.jar` file contains the standard portable implementations of home and remote handles, and metadata objects.

Configuring OC4J for Interoperability

To add interoperability support to your EJB, you must specify interoperability properties. Some of these properties are specified when starting OC4J and others in bean properties that are specified in deployment files.

The following security-related topics are now covered in the *Oracle Application Server Containers for J2EE Security Guide*:

- EJB Server Security Properties (internal-settings.xml)
- CSiv2 Security Properties
- CSiv2 Security Properties (internal-settings.xml)
- CSiv2 Security Properties (ejb_sec.properties)
- CSiv2 Security Properties (orion-ejb-jar.xml)
- EJB Client Security Properties (ejb_sec.properties)

Interoperability OC4J Flags

The following OC4J startup flags support RMI interoperability:

- `-DGenerateIIOP=true` generates new stubs and skeletons whenever you redeploy an application.
- `-Diiop.debug=true` generates deployment-time debugging messages, most of which have to do with code generation.
- `-Diiop.runtime.debug=true` generates runtime debugging messages.

Interoperability Configuration Files

Before EJBs can communicate, you must configure the parameters in the configuration files listed in [Table 6–2](#).

Table 6–2 Interoperability Configuration Files

Context	File	Description
Server	server.xml	The <code><sep-config></code> element in this file specifies the path name, normally <code>internal-settings.xml</code> , for the server extension provider properties. For example: <code><sep-config path="./internal-settings.xml"></code>
	internal-settings.xml	This file specifies server extension provider properties that are specific to RMI/IIOP. See the <i>Oracle Application Server Containers for J2EE Security Guide</i> for more information on these security-related topics.
Application	orion-ejb-jar.xml	The <code><ior-security-config></code> subentity of the <code><session-deployment></code> and <code><entity-deployment></code> entities specifies Common Secure Interoperability Version 2 (CSiv2) security properties for the server. See the <i>Oracle Application Server Containers for J2EE Security Guide</i> for more information on these security-related topics.

Table 6–2 (Cont.) Interoperability Configuration Files

Context	File	Description
	<code>ejb_sec.properties</code>	This file specifies client-side security properties for an EJB. See the <i>Oracle Application Server Containers for J2EE Security Guide</i> for more information on these security-related topics.
	<code>jndi.properties</code>	This file specifies the URL of the initial naming context used by the client. See " JNDI Properties for Interoperability (jndi.properties) " on page 6-13 for details.

JNDI Properties for Interoperability (`jndi.properties`)

The following RMI/IIOP properties are controlled by the client's `jndi.properties` file:

- `java.naming.provider.url` may be an OPMN or a corbaname URL for the bean to be interoperable. For details on corbaname URLs, see "[The corbaname URL](#)" on page 6-10. For details on the OPMN URL, see "[The OPMN URL](#)" on page 6-10.
- `contextFactory` can be either `ApplicationClientInitialContextFactory` or the class `IIOPInitialContextFactory`.

If your application has an `application-client.xml` file, then leave `contextFactory` set to `ApplicationClientInitialContextFactory`. If your application does not have an `application-client.xml` file, then change `contextFactory` to `IIOPInitialContextFactory`.

Context Factory Usage

`com.evermind.server.ApplicationClientInitialContextFactory` is used when looking up remote objects from standalone application clients. It uses the `refs` and `ref-mappings` found in `application-client.xml` and `orion-application-client.xml`. It is the default initial context factory when the initial context is instantiated in a Java application.

`com.oracle.iop.server.IIOPInitialContextFactory` is used when looking up remote objects between different containers using the IIOP protocol.

Enabling IIOP in OC4J

This tutorial describes the steps necessary to enable IIOP applications in OC4J. After completing this exercise you should be able to:

- Access a remote EJB over IIOP
- Secure EJB invocations with IIOP over SSL
- Secure corbaname lookups by remote clients with IIOP over SSL

To minimize deployment and configuration changes, the tutorial uses the demo "helloworld" EJB application available through the installation and on Oracle's OTN site:

http://www.oracle.com/technology/sample_code/tech/java/ejb_corba/index.html

Building and deploying the `helloworld` application with a default OC4J installation results in an application that is only accessible over ORMI. To enable IIOP for a given application you must make the following changes to OC4J's server configuration and the client application. The required changes include:

- Configure the `IIOPServerExtensionProvider`
- Change the `java.naming.provider.url`
- Deploy the application using the `-iiopClientJar` argument

The following sections describe the steps in more detail.

Getting Started

Expand the EJB demos to your development environment. The `helloworld` application should be under `<install-dir>/demo/ejb/helloworld` with the following structure:

```
|-----dist
|-----etc
|         |-----application-client.xml
|         |-----application.xml
|         |-----ejb-jar.xml
|         |-----jndi.properties
|-----src
|         |-----ejb
|         |         |-----client
|         |         |         |-----HelloClient.java
|         |         |-----helloworld-ejb
|         |         |         |-----Hello.java
|         |         |         |-----HelloBean.java
|         |         |         |-----HelloHome.java
|         |         |         |-----HelloLocal.java
|         |         |         |-----HelloLocalHome.java
|-----build.xml
```

Applications other than the `helloworld` example can be ignored for the remainder of the tutorial, but should be unaffected by changes required for enabling IIOP. This tutorial installs the demos to the root partition, so the application is under `/demo/ejb/helloworld`.

The supplied Ant build file provides targets for compiling the `src`, building the `jar` and `ear`, and running the client application. This tutorial assumes you are familiar with Ant build files. If you are unfamiliar with Ant, please see Apache's Ant documentation site: <http://ant.apache.org/manual/index.html>

Configure IIOP in OC4J

Edit the `server.xml` file as follows:

```
<install-dir>j2ee/home/config/internal-settings.xml
```

Make sure the `server.xml` file contains the following line:

```
<sep-config path="./internal-settings.xml" />
```

If the line is missing, or commented out, remove the comments or add the line below the following line:


```
<.rmi-config path="./rmi.xml" />
```

This configures the `IIOPServerExtensionProvider` for OC4J.

Now edit the `internal-settings.xml` file as follows to configure your IIOP settings:

```
<install-dir>j2ee/home/config/internal-settings.xml
```

Make sure the file contains the following settings:

```
<server-extension-provider name="IIOP"
  class="com.oracle.iop.server.IIOPServerExtensionProvider">
  <sep-property name="port" value="5555" />
  <sep-property name="host" value="localhost" />
  <sep-property name="ssl" value="false" />
  <sep-property name="trusted-clients" value="*" />
</server-extension-provider>
```

If necessary, you can modify the host and port to match your environment. If your file contains entries for SSL, temporarily comment them out:

```
<!--
  <sep-property name="ssl-port" value="5556" />
  <sep-property name="ssl-client-server-auth-port" value="5557" />
  <sep-property name="keystore" value="keystore.jks" />
  <sep-property name="keystore-password" value="->pwForSSL" />
  <sep-property name="truststore" value="truststore.jks" />
  <sep-property name="truststore-password" value="->pwForSSL" />
-->
```

Now OC4J is configured for IIOP. The final step to enable IIOP on the server side is to start OC4J with the JVM argument: `-DgenerateIIOP=true`. This can be done on the command line for OC4J standalone, and in the `${ORACLE_HOME}/opmn/opmn.xml` file for Oracle Application Server installs.

Configure the JNDI provider URL

Edit the `jndi.properties` file as follows for the helloworld application:

```
<install-dir>/demo/ejb/helloworld/etc/jndi.properties
```

```
java.naming.factory.initial=com.evermind.server.ApplicationClientInitialContextFactory
java.naming.provider.url=corbaname:iiop:localhost:5555#helloworld
#java.naming.provider.url=ormi://localhost:23791/helloworld
java.naming.security.principal=admin
java.naming.security.credentials=welcome
```

Comment out the line containing the ORMI provider URL, and add a line matching the corbaname provider URL in the example.

Build and deploy the application

From the `<install-dir>/demo/ejb/helloworld` directory, run the default ant target to build the application:

```
<install-dir>/demo/ejb/helloworld > ant
```

Start OC4J if you have not done so already, then execute the following deploy command:

```
java -jar ${J2EE_HOME}/admin.jar ormi://localhost:23791 admin welcome -deploy
```

```
-file dist/helloworld.ear -deploymentName helloworld -iiopClientJar
dist/helloworld_iiop_client.jar
```

This deploys the `helloworld` application and generates the client EJB JAR containing the client IIOP stubs in `dest/helloworld_iiop_client.jar`.

Run the application

Edit the `<install-dir>/demo/ejb/common.xml` file, and make sure the environment settings for `ORACLE_HOME`, `JAVA_HOME`, and `J2EE_HOME` match your environment.

Execute `> ant run`

You should see the appropriate "Hello ..." response from the client application. To verify that communication is running over IIOP, you can set the following JMV arg on both the client and the server:

```
-Diiop.runtime.debug=true
```

Enable IIOP over SSL on the server

Edit the `internal-settings.xml` file and uncomment or add the SSL settings (indicated by the **bold** lines in the following example):

```
<server-extension-provider name="IIOP"
  class="com.oracle.iio.server.IIOPServerExtensionProvider">
  <sep-property name="port" value="5555" />
  <sep-property name="host" value="localhost" />
  <sep-property name="ssl" value="true" />
  <sep-property name="trusted-clients" value="*" />
  <sep-property name="ssl-port" value="5556" />
  <sep-property name="ssl-client-server-auth-port" value="5557" />
  <sep-property name="keystore" value="keystore.jks" />
  <sep-property name="keystore-password" value="yourPWD" />
  <sep-property name="truststore" value="truststore.jks" />
  <sep-property name="truststore-password" value=" yourPWD " />
</server-extension-provider>
```

If necessary, you can modify the host and port to match your environment. The keystore and truststore files may be the same physical file. The names above are only for illustration. If you do not have a keystore file, you can use the following Sun example for using the `keytool`:

<http://java.sun.com/docs/books/tutorial/security/1.2/summary/tools.html>

Add the absolute path and filename to the keystore and truststore properties in the example.

Enable SSL on the client

Edit the `jndi.properties` file for the `helloworld` application:

```
<install-dir>/demo/ejb/helloworld/etc/jndi.properties
java.naming.factory.initial=com.evermind.server.ApplicationClientInitialContextFactory
java.naming.provider.url=corbaname:iiop:localhost:5556#helloworld
java.naming.security.principal=admin
java.naming.security.credentials=welcome
```

Change the port in the provider URL to be the SSL port from `internal-settings.xml`.

Create a file called `ejb_sec.properties` for the `helloworld` application:

```
oc4j.iiop.trustedServers=*
nameservice.useSSL=true
oc4j.iiop.trustStoreLoc=<path to server's keystore>
oc4j.iiop.trustStorePass=<password for server's keystore file>
```

This file communicates the applications security requirements to the OC4J client bootstrap classes. The properties in this example indicate that SSL should be used for EJB lookups, and that all servers supporting SSL should be trusted. The `truststore` setting is a quick way of using the same keystore configured for OC4J instead of exporting the certificate in the servers keystore and importing it into a second `truststore` file.

Run the application with IIOP over SSL

Execute `> ant run`

You should see the appropriate "Hello . . ." response from the client application. To verify that communication is running with IIOP over SSL, set the `-Diiop.runtime.debug=true` for both the client and the server.

Java Transaction API

This chapter describes the Oracle Application Server Containers for J2EE (OC4J) Java Transaction API (JTA). This chapter covers the following topics:

- [Introduction](#)
- [Single-Phase Commit](#)
- [Two-Phase Commit](#)
- [Configuring Timeouts](#)
- [Recovery for CMP Beans when a Database Instance Fails](#)
- [Using Transactions With MDBs](#)

Introduction

Applications deployed in the application server can demarcate transactions using Java Transaction API (JTA) 10.1.

For example, Enterprise Java Beans (EJBs) with bean-managed transactions, servlets, or Java objects that are deployed in the OC4J container can begin and end (demarcate) a transaction.

This chapter discusses the method for using JTA in OC4J. It does not cover JTA concepts—you must understand how to use and program global transactions before reading this chapter. See the Sun Microsystems Web site for more information:
<http://java.sun.com/products/jta>

Code examples are available for download from the OTN OC4J sample code site:

http://www.oracle.com/technology/sample_code/tech/java/oc4j/htdocs/oc4jsamplecode/oc4j-demo-ejb.html

JTA involves demarcating transactions and enlisting resources.

Demarcating Transactions

Your application demarcates transactions. Enterprise Java Beans use JTA 1.0.1 for managing transactions through either bean-managed or container-managed transactions.

- Bean-managed transactions are programmatically demarcated within your bean implementation. The transaction boundaries are completely controlled by the application.
- Container-managed transactions are controlled by the container. That is, the container either joins an existing transaction or starts a new transaction for the

application—as defined within the deployment descriptor—and ends the newly created transaction when the bean method completes. It is not necessary for your implementation to provide code for managing the transaction.

Note: Not all data sources support JTA transactions. (See ["Using Data Sources"](#) on page 4-15 for details.)

Enlisting Resources

The complexity of your transaction is determined by how many resources your application enlists with the transaction.

- [Single-Phase Commit](#) (1pc): If only a single resource (database) is enlisted in the transaction, then you can use single-phase commit.
- [Two-Phase Commit](#) (2pc): If more than one resource is enlisted, then you must use two-phase commit, which is more difficult to configure.

Single-Phase Commit

Single-phase commit (1pc) is a transaction that involves only a single resource. JTA transactions consist of enlisting resources and demarcating transactions.

Enlisting a Single Resource

To enlist the single resource in the single-phase commit, perform the following two steps:

- [Configure the Data Source](#)
- [Retrieve the Data Source Connection](#)

Configure the Data Source

Use an emulated data source for a single phase commit. Refer to [Chapter 4, "Data Sources"](#), for information on emulated and nonemulated data source types.

If you can, use the default data source (`data-sources.xml`) that comes with a standard OC4J installation for the single-phase commit JTA transaction. After modifying this data source `url` attribute with your database URL information, retrieve the data source in your code using a JNDI lookup with the JNDI name configured in the `ejb-location` attribute. Configure a data source for each database involved in the transaction.

```
<data-source
  class="com.evermind.sql.DriverManagerDataSource"
  name="OracleDS"
  location="jdbc/OracleCoreDS"
  xa-location="jdbc/xa/OracleXADS"
  ejb-location="jdbc/OracleDS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="scott"
  password="tiger"
  url="jdbc:oracle:thin:@//localhost:1521/ORCL"
  inactivity-timeout="30"
/>
```

In the preceding example, *myhost*, *myport*, and *mySID* are entries that you must change. You must edit the example to provide meaningful values for *myhost*, *myport*, and *mySID*.

For information about the expected attribute definitions, see [Chapter 4, "Data Sources"](#).

Retrieve the Data Source Connection

Before executing any SQL statements against tables in the database, you must retrieve a connection to that database. For these updates to be included in the JTA transaction, perform the following two steps:

- [Perform JNDI Lookup](#)
- [Retrieve a Connection](#)

Perform JNDI Lookup

After the transaction has begun, look up the data source from the JNDI name space. Here are the two methods for retrieving the data source:

- [Perform JNDI Lookup on Data Source Definition](#)
- [Perform JNDI Lookup Using Environment](#)

Perform JNDI Lookup on Data Source Definition You can perform a lookup on the JNDI name bound to the data source definition in the `data-sources.xml` file and retrieve a connection, as follows:

```
Context ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("jdbc/OracleDS");
Connection conn = ds.getConnection();
```

Perform JNDI Lookup Using Environment You can perform a lookup on a logical name that is defined in the environment of the bean container. For more information, see [Chapter 4, "Data Sources"](#). Define the logical name in the J2EE deployment descriptor in `ejb-jar.xml` or `web.xml` as follows:

```
<resource-ref>
  <res-ref-name>jdbc/OracleMappedDS</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

Map the `<res-ref-name>` in the OC4J-specific deployment descriptor (for example, `orion-ejb-jar.xml`) to the JNDI name that is bound in the `data-sources.xml` file as follows, where `"jdbc/OracleDS"` is the JNDI name defined in the `data-sources.xml` file:

```
<resource-ref-mapping name="jdbc/OracleMappedDS" location="jdbc/OracleDS" />
```

Then retrieve the data source using the environment JNDI lookup and create a connection, as shown in the following:

```
InitialContext ic = new InitialContext();
DataSource ds = ic.lookup("java:comp/env/jdbc/OracleMappedDS");
Connection conn = ds.getConnection();
```

Then start preparing and executing JDBC statements against the database.

Retrieve a Connection

Retrieve a connection off this data source object using the `getConnection` method. You can do this in one of two ways:

- Use `ds.getConnection()`. Use the method with no arguments.
- Use `ds.getConnection(username, password)`. Use the method supplying a user name and password.

Use the method with no arguments when the data source definition contains the user name and password that you want.

Use the other method when the data source definition does not contain a user name and password, or when you want to use a user name and password that is different from what is specified in the data source.

Demarcating the Transaction

With JTA, you can demarcate the transaction yourself by specifying that the bean is bean-managed transactional, or designate that the container should demarcate the transaction by specifying that the bean is container-managed transactional. Container-managed transaction is available to all EJBs. However, the bean-managed transactions are available for session beans and MDBs.

Note: The client cannot demarcate the transaction. Propagation of the transaction context cannot cross OC4J instances. Thus, neither a remote client nor a remote EJB can initiate or join the transaction.

Specify the type of demarcation in the bean deployment descriptor. [Example 7-1](#) shows a session bean that is declared as container-managed transactional by defining the `<transaction-type>` element as `Container`. To configure the bean to use bean-managed transactional demarcation, define this element to be `Bean`.

Example 7-1 Session Bean Declared as Container-Managed Transactional

```
</session>
<description>no description</description>
<ejb-name>myEmployee</ejb-name>
<home>cmtxn.ejb.EmployeeHome</home>
<remote>cmtxn.ejb.Employee</remote>
<ejb-class>cmtxn.ejb.EmployeeBean</ejb-class>
<session-type>Stateful</session-type>
<transaction-type>Container</transaction-type>
<resource-ref>
  <res-ref-name>jdbc/OracleMappedDS</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Application</res-auth>
</resource-ref>
</session>
```

Container-Managed Transactional Demarcation

If you define your bean to use CMTs, then you must specify how the container manages the JTA transaction for this bean in the `<trans-attribute>` element in the deployment descriptor (shown in [Example 7-2](#)). [Table 7-1](#) briefly describes the transaction attribute types that you should specify in the deployment descriptor.

Table 7-1 Transaction Attributes

Transaction Attribute	Description
NotSupported	The bean is not involved in a transaction. If the bean invoker calls the bean while involved in a transaction, then the invoker's transaction is suspended, the bean executes, and when the bean returns, the invoker's transaction is resumed.
Required	The bean must be involved in a transaction. If the invoker is involved in a transaction, then the bean uses the invoker's transaction. If the invoker is not involved in a transaction, then the container starts a new transaction for the bean. This attribute is the default.
Supports	Whatever transactional state that the invoker is involved in is used for the bean. If the invoker has begun a transaction, then the invoker's transaction context is used by the bean. If the invoker is not involved in a transaction, then neither is the bean.
RequiresNew	Whether or not the invoker is involved in a transaction, this bean starts a new transaction that exists only for itself. If the invoker calls while involved in a transaction, then the invoker's transaction is suspended until the bean completes.
Mandatory	The invoker must be involved in a transaction before invoking this bean. The bean uses the invoker's transaction context.
Never	The bean is not involved in a transaction. Furthermore, the invoker cannot be involved in a transaction when calling the bean. If the invoker is involved in a transaction, then a <code>RemoteException</code> is thrown.

Note: The default transaction attribute (<trans-attribute> element) for each type of entity bean is as follows:

- For CMP 2.0 entity beans, the default is `Required`.
 - For MDBs, the default is `NotSupported`.
 - For all other entity beans, the default is `Supports`.
-

[Example 7-2](#) shows the <container-transaction> portion of the deployment descriptor. It demonstrates how this bean specifies the `RequiresNew` transaction attribute for all (*) methods of the `myEmployee` EJB.

Example 7-2 <container-transaction> in Deployment Descriptor

```
<assembly-descriptor>
  <container-transaction>
    <description>no description</description>
    <method>
      <ejb-name>myEmployee</ejb-name>
      <method-name>*</method-name>
    </method>
    <trans-attribute>RequiresNew</trans-attribute>
```

```
</container-transaction>  
</assembly-descriptor>
```

No bean implementation is necessary to start, commit, or roll back the transaction. The container handles all these functions based on the transaction attribute that is specified in the deployment descriptor.

Bean-Managed Transactions

If you declare the bean as bean-managed transactional (BMT) within the `<transaction-type>`, then the bean implementation must demarcate the start, commit, or rollback for the global transaction. In addition, you must be careful to retrieve the data source connection after you start the transaction, not before.

Programmatic Transaction Demarcation For programmatic transaction demarcation, the bean developer can use either the JTA user transaction interface or the JDBC connection interface methods. The bean developer must explicitly start and commit or roll back transactions within the timeout interval.

Web components (JSP, servlets) can use programmatic transaction demarcation. Stateless and stateful session beans can use it; entity beans cannot, and thus must use declarative transaction demarcation.

Client-Side Transaction Demarcation This form of transaction demarcation is not required by the J2EE specification, and is not recommended for performance and latency reasons. OC4J does not support client-side transaction demarcation.

JTA Transactions

The Web component or bean writer must explicitly issue begin, commit, and rollback methods of the `UserTransaction` interface, as follows:

```
Context initCtx = new Initial Context();  
ut = (UserTransaction) initCtx.lookup("java:comp/UserTransaction");  
...  
ut.begin();  
// Commit the transaction started in ejbCreate.  
Try {  
    ut.commit();  
} catch (Exception ex) { ... }
```

JDBC Transactions

The `java.sql.Connection` class provides commit and rollback methods. JDBC transactions implicitly begin with the first SQL statement that follows the most recent commit, rollback, or connect statement.

Two-Phase Commit

The main focus of JTA is to declaratively or programmatically start and end simple and global transactions. When a global transaction is completed, all changes are either committed or rolled back. The difficulty in implementing a two-phase commit transaction is in the configuration details. For two-phase commit, you must use only a nonemulated data source. For more information on nonemulated data sources, refer to "[Nonemulated Data Sources](#)" on page 4-4.

Figure 7–1 contains an example of a two-phase commit engine, `jdbc/OracleCommitDS`, coordinating two databases in the global transaction—`jdbc/OracleDS1` and `jdbc/OracleDS2`. Refer to this example when configuring your JTA two-phase commit environment.

Configuring Two-Phase Commit Engine

When a global transaction multiple databases, the changes to these resources must all be committed or rolled back at the same time. That is, when the transaction ends, the transaction manager contacts a coordinator—also known as a two-phase commit engine—to either commit or roll back all changes to all included databases. The two-phase commit engine is an Oracle9i Database Server database that you must configure with the following:

- Fully-qualified database links from itself to each of the databases involved in the transaction. When the transaction ends, the two-phase commit engine communicates with the included databases over their fully qualified database links.
- A user that is designated to create sessions to each database involved and is given the responsibility of performing the commit or rollback. The user that performs the communication must be created on all involved databases and be given the appropriate privileges.

To facilitate this coordination, perform the following database and OC4J configuration steps shown in the next two subsections.

Database Configuration Steps

Designate and configure an Oracle9i Database Server database as the two-phase commit engine with the following steps:

1. Create the user (for example, `COORDUSR`) on the two-phase commit engine that facilitates the transaction, and perform the following three actions:
 - a. The user must open a session from the two-phase commit engine to each of the involved databases.
 - b. Grant the user the `CONNECT`, `RESOURCE`, `CREATE SESSION` privileges to be able to connect to each of these databases. The `FORCE ANY TRANSACTION` privilege allows the user to commit or roll back the transaction.
 - c. Create this user and grant these permissions on all databases involved in the transaction.

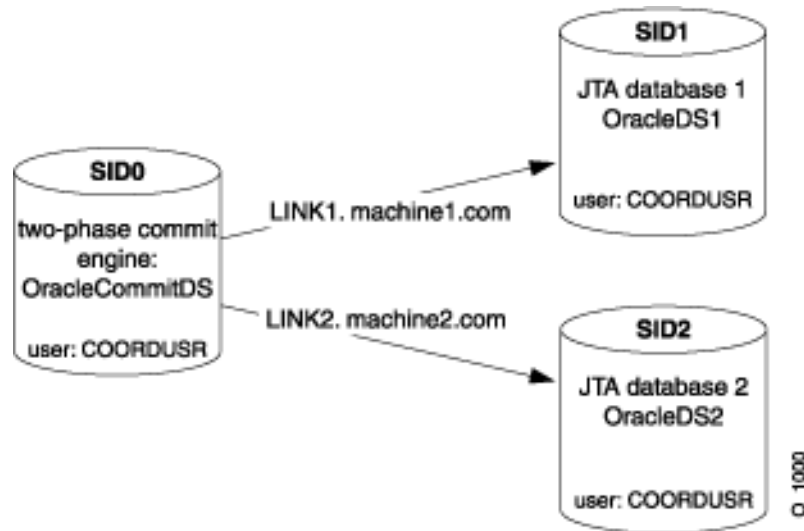
For example, if the user that is needed for completing the transaction is `COORDUSR`, do the following on the two-phase commit engine and *each* database involved in the transaction:

```
CONNECT SYSTEM/MANAGER;
CREATE USER COORDUSR IDENTIFIED BY COORDUSR;
GRANT CONNECT, RESOURCE, CREATE SESSION TO COORDUSR;
GRANT FORCE ANY TRANSACTION TO COORDUSR;
```

2. Configure fully qualified public database links (using the `CREATE PUBLIC DATABASE LINK` command) from the two-phase commit engine to each database that can be involved in the global transaction. This step is necessary for the two-phase commit engine to communicate with each database at the end of the transaction. The `COORDUSR` must be able to connect to all participating databases using these links.

Figure 7–1 shows two databases involved in the transaction. The database link from the two-phase commit engine to each database is provided on each `OrionCMTDataSource` definition in a `<property>` element in the `data-sources.xml` file. See the next step for the "dblink" `<property>` element.

Figure 7–1 Two-Phase Commit Diagram



OC4J Configuration Steps

1. To configure two-phase commit coordination: First, define the database that is to act as the two-phase commit engine, then configure it as follows:

Define a nonemulated data source, using `OrionCMTDataSource`, for the two-phase commit engine database in the `data-sources.xml` file. The following code defines the two-phase commit engine `OrionCMTDataSource` in the `data-sources.xml` file.

```
<data-source
  class="com.evermind.sql.OrionCMTDataSource"
  name="OracleCommitDS"
  location="jdbc/OracleCommitDS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="coordusr"
  password="coordpwd"
  url="jdbc:oracle:thin:@//localhost:1521/ORCL"
  inactivity-timeout="30"
/>
```

Refer to the two-phase commit engine data source in the global `application.xml` file, which resides in the `config` directory.

Configure the two-phase commit engine as follows:

```
<commit-coordinator>
  <commit-class class="com.evermind.server.OracleTwoPhaseCommitDriver" />
  <property name="datasource" value="jdbc/OracleCommitDS" />
  <property name="username" value="coordusr" />
  <property name="password" value="coordpwd" />
</commit-coordinator>
```

Note: The password attribute of the `<commit-coordinator>` element supports password indirection. For more information, refer to the *Oracle Application Server Containers for J2EE Security Guide*.

The parameters are as follows:

Specify the JNDI name of "jdbc/OracleCommitDS" for the `OrionCMTDataSource` that is defined in the `data-sources.xml` file. This identifies the data source to use as the two-phase commit engine.

Specify the two-phase commit engine user name and password. This step is optional, because you could also specify it in the data source configuration. These are the user name and password to use as the login authorization to the two-phase commit engine. This user must have the `FORCE ANY TRANSACTION` database privilege, or all session users must be identical to the user that is the commit coordinator.

Specify the `<commit-class>`. This class is always `OracleTwoPhaseCommitDriver` for two-phase commit engines.

The JNDI name for the `OrionCMTDataSource` is identified in the `<property>` element whose name is "datasource".

The user name is identified in the `<property>` element "username".

The password is identified in the `<property>` element "password".

2. To configure databases that will participate in a global transaction, configure nonemulated data source objects of the type `OrionCMTDataSource` for each database involved in the transaction with the following information:
 - a. The JNDI bound name for the object.
 - b. The URL for creating a connection to the database.
 - c. The fully qualified database link from the two-phase commit engine to this database (for example, `LINK1.machine1.COM`). This is provided in a `<property>` element within the data source definition in the `data-sources.xml` file.

The following `OrionCMTDataSource` objects specify the two databases involved in the global transaction. Notice that each of them has a `<property>` element named "dblink" that denotes the database link from the two-phase commit engine to itself.

```
<data-source
  class="com.evermind.sql.OrionCMTDataSource"
  name="OracleCMTDS1"
  location="jdbc/OracleDS1"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="scott"
  password="tiger"
  url="jdbc:oracle:thin:@//localhost:1521/db1.ORCL"
  inactivity-timeout="30">
  <property name="dblink"
    value="LINK1.machine1.COM" />
</data-source>
```

```
<data-source
  class="com.evermind.sql.OrionCMTDataSource"
  name="OracleCMTDS2"
```

```

location="jdbc/OracleDS2"
connection-driver="oracle.jdbc.driver.OracleDriver"
username="scott"
password="tiger"
url="jdbc:oracle:thin:@//localhost:1521/db2.ORCL"
inactivity-timeout="30">
<property name="dblink"
value="LINK2.machine2.COM"/>
</data-source>

```

Note: If you change the two-phase commit engine, then you must update all database links—both within the new two-phase commit engine as well as within the `OrionCMTDataSource` `<property>` definitions.

After the two-phase commit engine and all the databases involved in the transaction are configured, you can start and stop a transaction in the same manner as the single-phase commit. See ["Single-Phase Commit"](#) on page 7-2 for more information.

Limitations of Two-Phase Commit Engine

The following `data-sources.xml` configuration is supported for two-phase commit in the OC4J release:

```

<data-source
class="com.evermind.sql.OrionCMTDataSource"
location="jdbc/OracleDS"
connection-driver="oracle.jdbc.driver.OracleDriver"
username="scott"
password="tiger"
url="jdbc:oracle:thin:@//localhost:1521/ORCL
/>

```

Two-phase commit works only with a nonemulated data source configuration, as shown in the preceding code example. The URLs of all participating nonemulated data sources must point to an Oracle database instance. Only multiple Oracle resources participating in a global transaction have ACID (atomicity, consistency, isolation, durability) semantics after the commit. In summary, two-phase commit is supported only with Oracle database resources, but full recovery is always supported.

In the emulated configuration, two-phase commit may appear to work, but, because there is no recovery, it is not supported. The ACID properties of the transaction are not guaranteed and may cause problems for an application.

The JTA two-phase commit (2pc) function does not work with Oracle Database version 9.2. Instead, use Oracle Database version 9.2.0.4 or higher to enable the 2pc functionality.

Configuring Timeouts

You can configure timeouts in the `server.xml` file in the `<transaction-config>` element, which has a `timeout` attribute. This attribute specifies the maximum amount of time (in milliseconds) that a transaction can take to finish before it is rolled back due to a timeout. The default value is 30000. This timeout is a default timeout for all transactions that are started in OC4J. You can change the value by using the dynamic API `UserTransaction.setTimeout(milliseconds)`.

The server DTD defines the <transaction-config> element as follows:

```
<!ELEMENT transaction-config (#PCDATA)>
<!ATTLIST transaction-config timeout CDATA #IMPLIED>
```

Recovery for CMP Beans when a Database Instance Fails

You should be aware of any failure of the back-end database—especially if the CMP bean is acting within a transaction. If the database instance fails, then you may have to retry the operations that you were attempting during the moment of failure. The following sections detail how to implement recovery whether the CMP bean is within a container-managed transaction or a bean-managed transaction:

- [Connection Recovery for CMP Beans That Use Container-Managed Transactions](#)
- [Connection Recovery for CMP Beans That Use Bean-Managed Transactions](#)

Connection Recovery for CMP Beans That Use Container-Managed Transactions

If you define your CMP bean with container-managed transactions, then you can set a retry count and interval for re-establishing the transaction. Then if the database instance fails and your connection goes down while interacting within a transaction, the EJB container automatically retrieves a new connection to the database (within the specified interval) until the count is reached and re-executes the operations within the TRY block where the failure occurred.

To set the automatic retry count and interval, set the following optional attributes in the <entity-deployment> element in the CMP bean `orion-ejb-jar.xml` file:

- `max-tx-retries`—This parameter specifies the number of times to retry a transaction that was rolled back due to system-level failures. The default is 0.
- `tx-retry-wait`—This parameter specifies the time to wait in seconds between retrying the transaction. The default is 60 seconds.

Connection Recovery for CMP Beans That Use Bean-Managed Transactions

The EJB container does not manage bean-managed transactional CMP beans or EJB clients. Therefore, when they receive an exception denoting that the JDBC connection has failed, each must understand whether the method within the transaction can be retried.

To determine whether this is a retry scenario, provide the database connection and the SQL exception as parameters in the `DbUtil.oracleFatalError()` method, which determines if you can get a new connection and retry your operations. If this method returns true, then create a new connection to continue the transaction.

The following code demonstrates how to execute the `DbUtil.oracleFatalError()` method.

```
if ((DbUtil.oracleFatalError(sql_ex, db_conn))
{
    //retrieve the database connection again.
    //re-execute operations in the try block where the failure occurred.
}
```

Using Transactions With MDBs

Transactions, both BMT and CMT are supported within MDBs. The default transaction attribute (`trans-attribute`) for MDBs is `NOT_SUPPORTED`.

In accordance with the specification, MDBs support only the `REQUIRED` and `NOT_SUPPORTED` attributes. If you specify another attribute, such as `SUPPORTS`, then the default attribute `NOT_SUPPORTED` is used. An error is not thrown in this situation.

You can define a transaction timeout, as defined in the `transaction-timeout` attribute, in the `<message-driven-deployment>` element of the `ejb-jar.xml` file. This attribute controls the transaction timeout interval (in seconds) for any container-managed transactional MDB. The default is one day or 86,400 seconds. If the transaction has not completed in this time frame, then the transaction is rolled back.

Transaction Behavior for MDBs Using OC4J JMS

If you have heterogeneous or multiple resources involved in a single transaction, then two-phase commit is not supported. For example, if an MDB communicates to a CMP bean, which uses the database for persistence, and receives messages from a client through OC4J JMS, then this MDB includes two resources: The database and OC4J JMS. In this case, two-phase commit is not supported.

If you have no two-phase commit support, then there is no guarantee that when a transaction commits, all systems committed correctly. The same is true for rollbacks. You are not guaranteed ACID-quality global transactions without a two-phase commit engine.

Transaction Behavior for MDBs Using Oracle JMS

Oracle JMS uses a back-end Oracle database as the queue and topic facilitator. Because Oracle JMS uses database tables for the queues and topics, you may need to grant two-phase commit database privileges for your user.

OC4J optimizes one-phase commit for you so that it is not necessary to use two-phase commit unless you have two databases (or more than one data source) involved in the transaction. If you do use two-phase commit, it is fully supported within Oracle JMS.

You should be aware of any failure of the back-end database—especially if the MDB bean is acting within a transaction. If the database instance fails, then you may have to retry the operations that you were attempting during the moment of failure.

The following sections detail how to implement recovery whether the MDB bean is within a container-managed transaction or a bean-managed transaction:

- [Connection Recovery for CMP Beans That Use Container-Managed Transactions](#)
- [Connection Recovery for CMP Beans That Use Bean-Managed Transactions](#)

MDBs that Use Container-Managed Transactions

If you define your MDB with container-managed transactions you can set a retry count and interval for re-establishing the JMS session. Then, if your transaction fails while interacting with a database, the container automatically retries (within the specified interval) until the count is reached. To set the automatic retry count and interval, set the following optional attributes in the `<message-driven-deployment>` element in the MDB `orion-ejb-jar.xml` file:

- `dequeue-retry-count`—Specifies how often the listener thread tries to re-acquire the JMS session over a new database connection once database failover has occurred. The default is "0."
- `dequeue-retry-interval`—Specifies the interval between retries. The default is 60 seconds.

MDBs that Use Bean-Managed Transactions and JMS Clients

The container does not manage bean-managed transactional MDBs or JMS clients. Thus, when they receive an exception denoting that the JDBC connection has failed, each must understand if this is a scenario where the method within the transaction can be retried. To determine if this is a retry scenario, input the database connection and the SQL exception as parameters in the `DbUtil.oracleFatalError()` method.

You must retrieve the database connection from the JMS session object, and the SQL exception from the returned JMS exception, as follows:

1. Retrieve the underlying SQL exception from the JMS exception.
2. Retrieve the underlying database connection from the JMS session.
3. Execute the `DbUtil.oracleFatalError()` method to determine if the exception indicates an error that you can retry. If this method returns true, then create a new JMS connection, session, and possible sender to continue the JMS activity.

The following code demonstrates how to process the JMS exception, `jmsexc`, to pull out the SQL exception, `sql_ex`. In addition, the database connection, `db_conn`, is retrieved from the JMS session, `session`. The SQL exception and database connection are input parameters for the `DbUtil.oracleFatalError` method.

```
try
{
    ..
}
catch(Exception e )
{
    if (exc instanceof JMSEException)
    {
        JMSEException jmsexc = (JMSEException) exc;
        sql_ex = (SQLException)(jmsexc.getLinkedException());
        db_conn = (oracle.jms.AQjmsSession)session.getDBConnection();

        if ((DbUtil.oracleFatalError(sql_ex, db_conn))
        {
            // Since the DBUtil function returned true, regain the JMS objects
            // Look up the Queue Connection Factory.
            QueueConnectionFactory qcf = (QueueConnectionFactory)
                ctx.lookup ("java:comp/resource/" + resProvider +
                    "/QueueConnectionFactories/myQCF");

            // Lookup the Queue.
            Queue queue = (Queue) ctx.lookup ("java:comp/resource/" + resProvider +
                "/Queues/rpTestQueue");

            // Retrieve a connection and a session on top of the connection.
            // Create queue connection using the connection factory.
            QueueConnection qconn = qcf.createQueueConnection();

            // We're receiving msgs, so start the connection.
            qconn.start();

            // Create a session over the queue connection.
            QueueSession qsess = qconn.createQueueSession(false,
                Session.AUTO_ACKNOWLEDGE);

            //Since this is for a queue, create a sender on top of the session.
            //This is used to send out the message over the queue.
```

```
        QueueSender snd = sess.createSender (q);  
    }  
}
```

J2EE Connector Architecture (J2CA)

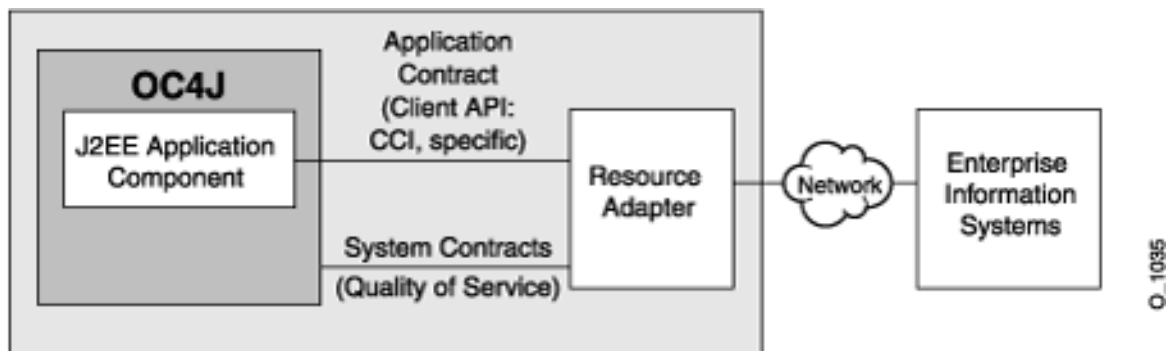
This chapter describes how to use the J2EE Connector Architecture (J2CA) 1.0 in an Oracle Application Server Containers for J2EE (OC4J) application. This chapter covers the following topics:

- [Introduction](#)
- [Deploying and Undeploying Resource Adapters](#)
- [Quality of Service Contracts](#)

Introduction

The J2EE Connector Architecture defines a standard architecture for connecting the J2EE platform to heterogeneous Enterprise Information Systems (EISs). Typical EISs include Enterprise Resource Planning (ERP), database systems, mainframe transaction processing, and legacy applications that are not written in the Java programming language. [Figure 8–1](#) shows the J2EE Connector Architecture.

Figure 8–1 J2EE Connector Architecture



Resource Adapters

A *resource adapter* is a driver that an application server or an application client uses to connect to a specific EIS. Examples of resource adapters are a JDBC driver to connect to a relational database, an ERP resource adapter to connect to an ERP system, and a transaction processing (TP) resource adapter to connect to a TP monitor. The J2EE 1.3 specification requires application servers to support both standalone and embedded resource adapters.

Standalone Resource Adapters

A resource adapter module that can be deployed directly into an application server independent of other applications is called a *standalone resource adapter*. These adapters, which are stored in standalone resource adapter archive (RAR) files, are available to all applications that are deployed in the application server instance. See "[Example of RAR File Structure](#)" on page 8-2 for an example of the contents and structure of an RAR archive.

Embedded Resource Adapters

A resource adapter module that is deployed as part of a J2EE application that also contains one or more J2EE modules, is called an *embedded resource adapter*. These adapters are available only to the J2EE applications with which they are bundled in an enterprise application archive (EAR) file.

Example of RAR File Structure

Here is an example of the contents and structure of an RAR archive:

```
/META-INF/ra.xml
/META-INF/oc4j-ra.xml
/howto.html
/images/icon.jpg
/ra.jar
/cci.jar
/win.dll
/solaris.so
```

Note: The JAR files that are referred to in the RAR file can be located in any directory within the archive.

Note: The file `/META-INF/oc4j-ra.xml` is not generally part of the RAR archive provided by the RAR vendor, and is typically generated by OC4J during deployment. But a deployer can choose to add the file `oc4j-ra.xml` to the RAR archive before deployment. Alternatively, the deployer can edit the generated file.

Depending on the resource adapter, applications or application modules might need to access adapter-specific classes that are bundled with the RAR. In the case of standalone resource adapters, these custom classes are available to all applications that are deployed within OC4J. In the case of embedded resource adapters, they are available only to modules that are part of the same application as the embedded adapter.

The ra.xml Descriptor

The `ra.xml` descriptor is the standard J2EE deployment descriptor for resource adapters. For details, see the J2EE Connector Architecture 1.0 specification.

Application Interface

The client API furnished by a resource adapter can be either a client API that is specific to the type of a resource adapter and its underlying EIS, or the standard Common Client Interface (CCI). For more information on CCI, see the J2EE Connector 1.0

Specification. An example of a client API is JDBC, the client API that is specific to relational database accesses.

You can determine what client interface a resource adapter supports. The client interface is specified in the `<connection-interface>` element in the `ra.xml` file bundled in the RAR archive.

Quality of Service Contracts

J2EE Connector Architecture also defines three Quality of Service (QoS) contracts between an application server and an EIS.

- *Connection Management* enables application components to connect to an EIS and leverage any connection pooling provided by the application server. Also see ["Configuring Connection Pooling"](#) on page 8-10.

Note: The J2EE Connector connection pooling interface differs from the JDBC interface. J2EE Connector connection pools are not shared with JDBC connection pools, nor do properties set for one connection pool affect the other.

- *Transaction Management* enables an application server to use a transaction manager to manage transactions across multiple resource managers.

Transaction management does not require any deployment-time configuration. For more information, see the J2EE Connector 1.0 Specification.

Support for optional features:

- OC4J does not support the optional connection sharing (section 6.9 in the J2EE Connector Architecture 1.0 specification) and local transaction optimization (section 6.12) features.
- OC4J does not support two-phase commit for J2EE Connector Architecture resource adapters. (For information on the limitations of two-phase commit, see [Chapter 7, "Java Transaction API"](#).)
- Security management provides authentication, authorization, and secure communication between the J2EE server and the EIS. Also see ["Managing EIS Sign-On"](#) on page 8-11.

All resource adapters must support their side of the QoS contracts to be pluggable into application servers.

Deploying and Undeploying Resource Adapters

This section discusses the details of deploying and undeploying resource adapters.

Deployment Descriptors

OC4J supports three deployment descriptors: `ra.xml`, `oc4j-ra.xml`, and `oc4j-connectors.xml`. The `ra.xml` descriptor is always supplied with the resource adapter. Whenever you deploy a resource adapter, OC4J generates `oc4j-ra.xml` if the file doesn't already exist in the archive. In addition, for an embedded resource adapter, OC4J generates `oc4j-connectors.xml` if it doesn't exist in the archive.

The oc4j-ra.xml Descriptor

The oc4j-ra.xml descriptor provides OC4J-specific deployment information for resource adapters. The file contains one or more <connector-factory> elements.

You can do the following using oc4j-ra.xml:

- Configure and bind instances of connection factories.

Connection factories are used by application components to obtain connections to the EIS. The name of the connection factory class is specified in the connectionfactory-impl-class element, defined in ra.xml. OC4J allows the deployer to configure instances of this class and to bind them to the Java Naming and Directory Interface (JNDI) name space.

The deployer can do this by creating <connector-factory> elements and assigning a JNDI location to each using the location attribute. The deployer can also configure each instance using <config-property> elements.

The list of configurable properties is specified in ra.xml, as <config-property> elements. The deployer can either specify or override values for these properties in oc4j-ra.xml, using <config-property> elements.

Example: Consider a resource adapter with a connection factory implementation of com.example.eis.ConnectionFactoryImpl. Assume that this adapter has been deployed standalone with one configured connection factory, whose JNDI location is myEIS/connFctry1. The <connector-factory> has been configured to connect to host myMc123 on port 1999. Also assume there is an EJB application that looks up and use this connection factory, using a logical name of eis/myEIS.

The following are the files that are relevant to this example.

ra.xml: Specification of connection factory implementation (as provided by the resource adapter vendor).

```
<resourceadapter>
  ...
  <config-property>
    <config-property-name>HostName</config-property-name>
    <config-property-type>java.lang.String</config-property-type>
  </config-property>
  <config-property>
    <config-property-name>Port</config-property-name>
    <config-property-type>java.lang.Integer</config-property-type>
    <config-property-value>2345</config-property-value>
  </config-property>
  <connectionfactory-impl-class>
    com.example.eis.ConnectionFactoryImpl
  </connectionfactory-impl-class>
  ...
</resourceadapter>
```

oc4j-ra.xml: Specification of connection factory implementation with properties myMc123 (host) and 1999 (port), to be bound to JNDI location myEIS/connFctry1 (likely generated by OC4J and edited by deployer).

```
<connector-factory location="myEIS/connFctry1">
  ...
  <config-property>
    <config-property-name>HostName</config-property-name>
    <config-property-value>myMc123</config-property-value>
  </config-property>
  <config-property>
```

```

        <config-property-name>Port</config-property-name>
        <config-property-value>1999</config-property-value>
    </config-property>
    ...
</connector-factory>

```

Note: The `<config-property-type>` element does not appear in the `oc4j-ra.xml` file because the type cannot be changed.

`ejb-jar.xml`: Specification of resource reference (that is, connection factory) accessed by EJB (as provided by the application vendor).

```

<resource-ref>
  <res-ref-name>eis/myEIS</res-ref-name>
  <res-type>javax.resource.cci.ConnectionFactory</res-type>
  <res-auth>Application</res-auth>
</resource-ref>

```

`orion-ejb-jar.xml`: Mapping of the logical reference name to the real JNDI name (likely generated by OC4J and edited by deployer).

```

<resource-ref-mapping name = "eis/myEIS" location = "myEIS/connFctry1" />

```

EJB class: Usage of the connection factory (created by developer).

```

try
{
    Context ic = new InitialContext();
    cf = (ConnectionFactory) ic.lookup("java:comp/env/eis/myEIS");
} catch (NamingException ex) {
    ex.printStackTrace();
}

```

- Customize connection pooling

The deployer can configure connection pooling for each instance of a connection factory using the `<connection-pooling>` element. This element is discussed under ["Configuring Connection Pooling"](#) on page 8-10.

- Manage authentication

The deployer can use the `<security-config>` element to configure an authentication scheme for each instance of a connection factory. This element is applicable only if application components use container-managed sign-on. Also see ["Managing EIS Sign-On"](#) on page 8-11.

- Set up logging

The deployer can set up logging, per connection factory instance, using the `<log>` element. Here is an example:

```

<connector-factory location="myEIS/connFctry1">
  <log>
    <file path="./logConnFctry1.log" />
  </log>
</connector-factory>

```

If the path name is not specified or if the directory does not exist, logging is not enabled and OC4J prints out a warning message. If the directory exists but the file does not, OC4J creates the file and enables logging. Because there is no default

location for the log file, if the <log> element is not specified, logging is not enabled.

Additionally, the deployer can also add a <description> element to each <connector-factory> element. The element contains a description of the connection factory and is not interpreted by OC4J.

The oc4j-connectors.xml Descriptor

The resource adapters that are deployed to OC4J can be configured through the oc4j-connectors.xml descriptor. There is one oc4j-connectors.xml file for all of the standalone adapters (as a group) as well as one per application.

Note: The file /META-INF/oc4j-connectors.xml is not generally part of the EAR archive provided by the EAR vendor, and is typically generated by OC4J during deployment. But a deployer can choose to add the file oc4j-connectors.xml to the EAR archive before deployment. Alternatively, the deployer can edit the generated file.

The root element is <oc4j-connectors>. Each individual connector is represented by a <connector> element that specifies the name and path name for the connector. Each <connector> element contains the following elements:

- <description>: text description of the connector. It is not interpreted by OC4J. This element is optional.
- <native-library path="pathname">: directory containing native libraries. If you do not specify this element, OC4J expects the libraries to be located in the directory containing the decompressed RAR directory. OC4J interprets the pathname attribute relative to the decompressed RAR directory. This element is optional.
- <security-permission enabled="booleanvalue">: permissions to be granted to each resource adapter. Each <security-permission> contains a <security-permission-spec> that conforms to the Java 2 Security policy file syntax.

OC4J automatically generates a <security-permission> element in oc4j-connectors.xml for each <security-permission> element in ra.xml. Each generated element has the enabled attribute set to false. Setting the enabled attribute to true grants the named permission. That is, the deployer has to explicitly grant the permissions requested by the resource adapter. The default behavior of OC4J is to not grant those permissions during deployment.

Example:

```
<oc4j-connectors>
  <connector name="myEIS" path="eis.rar">
    <native-library path="lib"></native-library>
    <security-permission>
      <security-permission-spec enabled="false">
        grant {permission java.lang.RuntimePermission "LoadLibrary.*"};
      </security-permission-spec>
    </security-permission>
  </connector>
</oc4j-connectors>
```

Note: The path attribute of the <native-library> element should point to the directory in which the .dll or .so files are located. For the preceding example, here is a possible RAR structure:

```
/META-INF/ra.xml
/ra.jar
/lib/win.dll
/lib/solaris.so
```

Standalone Resource Adapters

During deployment of standalone resource adapters, give each a unique name for future operations, such as undeployment of the resource adapter. OC4J does not permit deployment of two standalone resource adapters that have the same name.

The deployment descriptors and decompressed RAR files are located as shown in [Table 8-2](#).

Deployment

During deployment, OC4J decompresses the RAR file and creates OC4J-specific deployment descriptor files if they do not exist already. The deployment process automatically adds <connector> entries in the oc4j-connectors.xml file. Skeleton entries for <connector-factory> elements are created as well in oc4j-ra.xml. The deployer can edit these two files for further configuration. For more information, see the *Oracle Application Server Administrator's Guide*.

You deploy standalone resource adapters in one of the following ways:

- [Deploying and Undeploying Using dcmctl](#)
- [Deploying and Undeploying Using admin.jar](#)

Deploying and Undeploying Using dcmctl To deploy a standalone resource adapter to an Oracle Application Server instance, use the command-line tool dcmctl with the deployApplication option. Here is the syntax:

```
dcmctl deployApplication -f example.rar -a example
```

The deployApplication switch is supported by additional command-line switches:

- -f myRA.rar: path name of the resource adapter's RAR file. This switch is required.
- -a myRA: resource adapter's name. This switch is required.

To remove a deployed resource adapter, dcmctl with the undeployApplication option. Here is the syntax:

```
dcmctl undeployApplication -a example
```

The required -a argument specifies which adapter is being removed.

dcmctl supports RAR files, as well as WAR and EAR files. For more information, see the *Oracle Application Server Administrator's Guide*.

Deploying and Undeploying Using admin.jar To deploy a standalone resource adapter to an OC4J standalone instance, use the command-line tool admin.jar with the -deployconnector switch. Here is the syntax:

```
-deployconnector -file mypath.rar -name myname -nativeLibPath libpathname
-grantAllPermissions
```

The `-deployconnector` switch is supported by additional command-line switches:

- `-file myRA.rar` - path name of the resource adapter's RAR file. This switch is required.
- `-name myRA`: resource adapter's name. This switch is required.
- `-nativeLibPath libpathname` - path name for native libraries within the RAR file (see also the `<native-library>` element in "[The oc4j-connectors.xml Descriptor](#)" on page 8-6).
- `-grantAllPermissions` - grants all runtime permissions requested within the RAR file (see also the `<security-permission>` element in "[The oc4j-connectors.xml Descriptor](#)" on page 8-6).

Example:

```
java -jar admin.jar ormi://localhost admin welcome -deployconnector -file
./myRA.rar -name myRA
```

Note: For more information about `admin.jar`, see the *Oracle Application Server Containers for J2EE Stand Alone User's Guide*. You can download this document when you download the OC4J standalone product from OTN.

Deploying Manually

If you prefer to deploy your connector manually, follow these steps:

1. Create a `connectorname` directory under `OC4J_HOME/connectors_dir`.
2. Copy the connector's RAR file into `OC4J_HOME/connectors_dir/connectorname`.
3. Create an `oc4j-connectors.xml` file in `connectors_dir` for the new resource adapter, or add a `<connector>` element to the file if it already exists.
4. Restart OC4J. OC4J generates a new `oc4j-ra.xml` in `OC4J_HOME/application-deployments/default/connectorname` for the adapter. You must modify the generated file to contain a `<connector_factory>` element that is appropriate for your connector.

Note: See "[The oc4j-ra.xml Descriptor](#)" on page 8-4 and "[The oc4j-connectors.xml Descriptor](#)" on page 8-6 for details on `oc4j-connectors.xml` and `oc4j-ra.xml`.

To remove a deployed resource adapter, use the `-undeployconnector` switch of `admin.jar`. Here is the syntax:

```
-undeployconnector -name myname
```

The required `-name` argument specifies which adapter is being removed. This command removes all `<connector>` entries that use the specified resource adapter from `oc4j-connectors.xml` and deletes the directories and files that were created during deployment.

Embedded Resource Adapters

Embedded resource adapters cannot be deployed or undeployed independent of the application of which they are a part. The name of the adapter can be specified in the `oc4j-connectors.xml` file; if not specified in this file, the name used for the adapter is that of the RAR archive.

The deployment descriptors and decompressed RAR files are located as shown in [Table 8–2](#).

Deployment

As part of deploying the EAR file that contains the embedded resource adapter, OC4J decompresses the RAR file and creates OC4J-specific deployment descriptor files if they do not exist already. The deployment process automatically adds `<connector>` entries in the `oc4j-connectors.xml` file. Skeleton entries for `<connector-factory>` elements are created as well in `oc4j-ra.xml`. The deployer can edit these two files for further configuration.

Deploy applications that include embedded resource adapters in one of the following ways:

- [Deploying Using dcmctl](#)
- [Deploying Using admin.jar](#)

Deploying Using dcmctl For information on using `dcmctl`, see the *Oracle Application Server Administrator's Guide*.

Deploying Using admin.jar For more information about `admin.jar`, see the *Oracle Application Server Containers for J2EE Stand Alone User's Guide*.

Locations of Relevant Files

[Table 8–1](#) shows the paths to various deployment directories that OC4J creates during deployment that are referenced throughout the guide. The paths are relative to the root directory of your OC4J installation. The deployment directories can be customized within `server.xml` by setting the attributes shown in the table. These attributes belong to the `<application-server>` element.

Table 8–1 Directory Locations

	Attribute	Description of Attribute	Default Value
<code>connectors_dir</code>	<code>connector-directory</code>	The root directory for all standalone resource adapters.	<code>connectors</code>
<code>applications_dir</code>	<code>applications-directory</code>	The root directory for all applications.	<code>applications</code>
<code>application_deployments_dir</code>	<code>deployment-directory</code>	The root directory for all files generated at deployment.	<code>application-deployments</code>

[Table 8–2](#) shows the paths to various files produced during deployment that are referenced throughout the guide. The paths are relative to the root directory of your OC4J installation. In [Table 8–2](#), `appname` is the name under which the application is deployed.

Table 8–2 File Locations

	Standalone Resource Adapter	Embedded Resource Adapter
Location of decompressed RAR archive	connectors_dir/deployment_name	applications_dir/appname/rar_name
oc4j-connectors.xml	config or as defined in the <connectors> tag in application.xml.	application_deployments_dir/appname/META-INF or as defined in the <connectors> tag in orion-application.xml.
oc4j-ra.xml	application_deployments_dir/default/deployment_name	application_deployments_dir/appname/rar_name or application_deployments_dir/appname/connector_name if an oc4j-connectors.xml file is included in the EAR and you have specified a connector element with a name attribute.

Specifying Quality of Service Contracts

You can configure connection pooling and authentication mechanisms on a per-connection basis at deployment time. This section describes the different ways to accomplish this.

Configuring Connection Pooling

Connection pooling is a J2EE 1.3 feature that allows a set of connections to be reused within an application. Because the J2EE Connector 1.0 specification is intended to be general rather than database-specific, the J2EE Connector connection-pooling interface differs significantly from the JDBC interface.

To set a connection pooling property in oc4j-ra.xml, specify a <property> element within the optional <connection-pooling> element. If you don't specify this element, whenever the application requests a connection, a new connection is created. Here is the syntax:

```
<property name="propname" value="propvalue" />.
```

The value for `propname` must be one of:

- `maxConnections` - maximum number of connections permitted within a pool. If no value is specified, there is no limit on the number of connections.
- `minConnections` - minimum number of connections. If `minConnections` is greater than 0, the specified number of connections are opened when OC4J is initialized. OC4J may not be able to open the connections if necessary information is unavailable at initialization time. For instance, if the connection requires a JNDI lookup, it cannot be created, because JNDI information is not available until initialization is complete. The default value is 0.
- `scheme` - specifies how OC4J handles connection requests after the maximum permitted number of connections is reached. You must specify one of the following values:
 - `dynamic` - OC4J always creates a new connection and returns it to the application, even if this violates the maximum limit. When these

limit-violating connections are closed, they are destroyed instead of being returned to the connection pool.

Note: OC4J does not destroy pooled connections upon close unless the pool size is above the maximum specified in the `maxConnections` property.

- `fixed` - OC4J raises an exception when the application requests a connection and the maximum limit has been reached.
- `fixed_wait` - OC4J blocks the application's connection request until an in-use connection is returned to the pool. If `waitTimeout` is specified, OC4J throws an exception if no connection becomes available within the specified time limit.
- `waitTimeout` - Maximum number of seconds that OC4J waits for an available connection if `maxConnections` has been exceeded and the `fixed_wait` scheme is in effect. In all other cases, this property is ignored.

Note: If you make no `waitTimeout` specification, the default behavior is not to time out.

Here is an example of a `<connection-pooling>` element configuration:

```
<connection-pooling>
  <description>my pooling configuration </description>
  <property name="waitTimeout" value="60" />
  <property name="scheme" value="fixed_wait" />
  <property name="maxConnections" value="3" />
  <property name="minConnections" value="1" />
</connection-pooling>
```

The example defines a connection pool with a minimum of one connection (OC4J tries to create one connection during start up) and a maximum of three connections. When all three connections are in use and a request for connection is issued, the pool with a `fixed_wait` scheme tries to wait a maximum of 60 seconds for a connection to be returned to the pool. If there is still no connection available after 60 seconds, an exception is thrown to the caller of the API that requested a new connection.

Managing EIS Sign-On

As part of extending the end-to-end security of the J2EE mode to cover integration to EISs, J2EE Connector architecture allows application components to associate a security context with connections established to the EIS.

Application components can either sign on to the EIS by themselves, or have OC4J manage the sign-on. Component-managed sign-on must be implemented programmatically, while container-managed sign-on can be specified either declaratively or programmatically. Specify the type of sign-on using the `<res-auth>` deployment descriptor element for EJB or Web components.

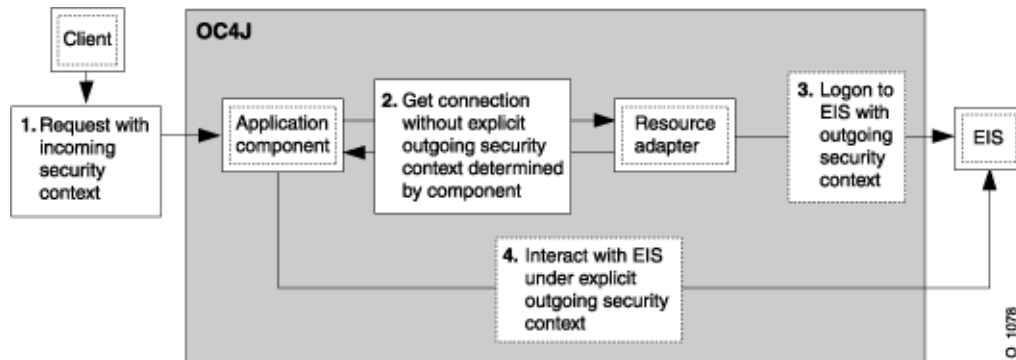
Note: The remainder of this section assumes that you are familiar with chapter 7 of the J2EE Connector Architecture 1.0 specification. The specification uses the terms initiating principal, caller principal, and resource principal. As used in this section, the incoming security context refers to either initiating principal or caller principal, and the outgoing security context refers to resource principal.

Component-Managed Sign-On

When deploying applications that manage EIS sign-on by themselves, set <res-auth> to Application. The application component is responsible for providing explicit security information for the sign-on.

Figure 8–2 shows the steps involved in component-managed sign-on. The steps are detailed following the diagram.

Figure 8–2 Component-Managed Sign-On



1. The client makes a request, which is associated with an incoming security context.
2. As part of servicing the request, the application component maps the incoming security context to an outgoing security context and then uses the outgoing security context to request a connection to the EIS.
3. As part of the connection acquisition, the resource adapter logs on to the EIS using the outgoing security context provided by the application component.
4. Once the connection is acquired, the application component can interact with the EIS under the established outgoing security context.

The following example is an excerpt from an application that performs component-managed sign-on.

Example:

```

Context initctx = new InitialContext();
// perform JNDI lookup to obtain a connection factory
javax.resource.cci.ConnectionFactory cxf =
    (javax.resource.cci.ConnectionFactory) initctx.lookup("java:com/env/eis/MyEIS");

// If component-managed sign-on is specified, the code
// should instead provide explicit security
// information in the getConnection call

// We need to get a new ConnectionSpec implementation
  
```

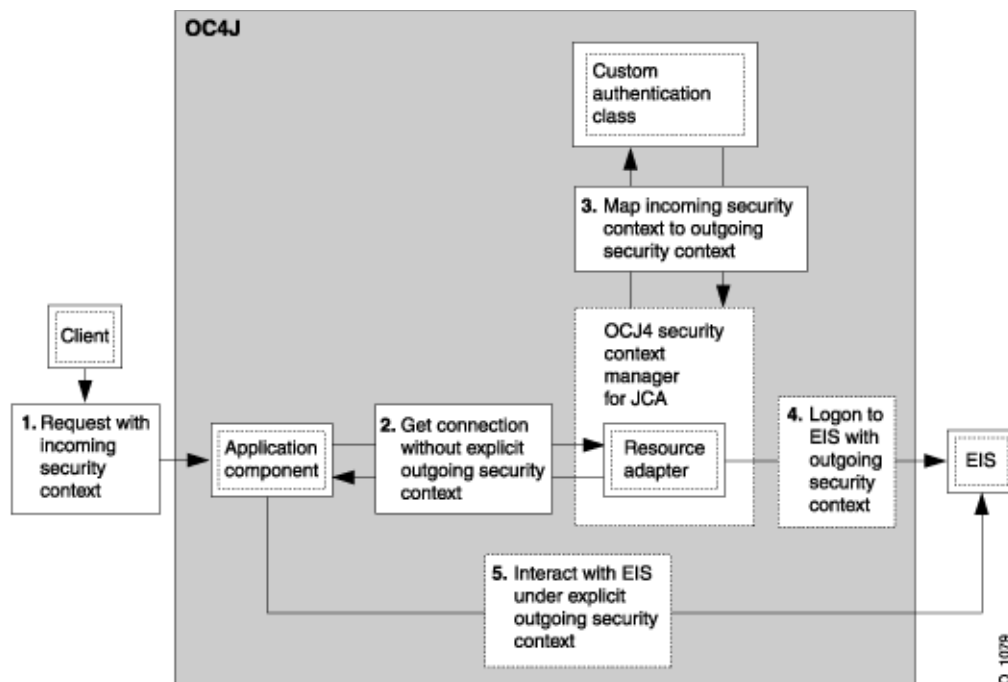
```
// instance for setting login attributes
com.myeis.ConnectionSpecImpl connSpec = ...
connSpec.setUserName("EISuser");
connSpec.setPassword("EISpassword");
javax.resource.cci.Connection cx = cxf.getConnection(connSpec);
```

Container-Managed Sign-On

When deploying applications that depend on the container for EIS sign-on, set <res-auth> to Container. The container is responsible for providing security information for the sign-on. Additionally, the container uses deployment descriptors or pluggable authentication classes to determine outgoing security context.

Figure 8–3 shows the steps involved in container-managed sign-on. The steps are detailed following the diagram.

Figure 8–3 Container-Managed Sign-On



1. The client makes a request, which is associated with an incoming security context.
2. As part of servicing the request, the application component requests a connection to the EIS.
3. As part of the connection acquisition, the container (the OC4J) security context manager shown in Figure 8–3) maps the incoming security context to outgoing security context, based on deployment descriptor elements (not shown in the figure) or authentication class provided.
4. The resource adapter logs on to the EIS using the outgoing security context provided by the container.
5. Once the connection is acquired, the application component can interact with the EIS under the established outgoing security context.

The following example is an excerpt from an application that depends on container-managed sign-on.

Example:

```
Context initctx = new InitialContext();

// perform JNDI lookup to obtain a connection factory
javax.resource.cci.ConnectionFactory cxf =
    (javax.resource.cci.ConnectionFactory) initctx.lookup("java:com/env/eis/MyEIS");

// For container-managed sign-on, no security information is passed in the
// getConnection call
javax.resource.cci.Connection cx = cxf.getConnection();
```

Declarative Container-Managed Sign-On

You can create principal mappings in the `oc4j-ra.xml` file. To employ the principal mappings mechanism, use the `<principal-mapping-entries>` subelement under the `<security-config>` element.

Each `<principal-mapping-entry>` element contains a mapping from initiating principal to resource principal and password.

Use the `<default-mapping>` element to specify the user name and password for the default resource principal. This principal is used to log on to the EIS if there is no `<principal-mapping-entry>` element whose initiating user corresponds to the current initiating principal. If the element `<principal-mapping-entries>` is not specified, OC4J may not be able to log in to the EIS.

For example, if the OC4J principal `scott` should be logged in to the EIS as user name `scott` and password `tiger`, while all other OC4J users should be logged in to the EIS using user name `guest` with password `guestpw`, the `<connector-factory>` element in `oc4j-ra.xml` should look like this:

```
<connector-factory name="..." location="...">
  ...
  <security-config>
    <principal-mapping-entries>
      <default-mapping>
        <res-user>guest</res-user>
        <res-password>guestpw</res-password>
      </default-mapping>
      <principal-mapping-entry>
        <initiating-user>scott</initiating-user>
        <res-user>scott</res-user>
        <res-password>tiger</res-password>
      </principal-mapping-entry>
    </principal-mapping-entries>
  </security-config>
  ...
</connector-factory>
```

Note: The `<res-password>` element supports password indirection. For more information, refer to the *Oracle Application Server Containers for J2EE Security Guide*.

Programmatic Container-Managed Sign-On

OC4J supports the use of programmatic authentication—either through the use of an OC4J-specific mechanism or a standard mechanism like the Java Authentication and Authorization Service (JAAS). See the Sun JAAS specification for more information.

OC4J-Specific Authentication Classes

OC4J provides the `oracle.j2ee.connector.PrincipalMapping` interface for principal mapping. Its methods appear in [Table 8-3](#).

To use OC4J-specific programmatic container-managed sign-on, an implementation of this interface must be provided.

Table 8-3 Method Description for `oracle.j2ee.connector.PrincipalMapping` Interface

Method Signature	Description
<pre>public void init(java.util.Properties prop)</pre>	Called by OC4J to initialize the settings for the <code>PrincipalMapping</code> implementation class. OC4J passes the properties specified in the <code><config-property></code> elements in <code>oc4j-ra.xml</code> to this method. The implementation class can use the properties for setting default user name and password, LDAP connection information, or default mapping.
<pre>public void setManagedConnectionFactory (ManagedConnectionFactory mcf)</pre>	Used by OC4J to provide the implementation class with the <code>ManagedConnectionFactory</code> instance that is needed to create a <code>PasswordCredential</code> .
<pre>public void setAuthenticationMechanisms (java.util.Map authMechanisms)</pre>	Called by OC4J to pass the authentication mechanisms supported by the resource adapter to the <code>PrincipalMapping</code> implementation class. The key of the map passed is a string containing the supported mechanism type, such as <code>BasicPassword</code> or <code>Kerby5</code> . The value is a string containing the corresponding credentials interface as declared in <code>ra.xml</code> , such as <code>javax.resource.spi.security.PasswordCredential</code> . The map can contain multiple elements if the resource adapter supports multiple authentication mechanisms.
<pre>public javax.security.auth.Subject mapping (javax.security.auth.Subject initiatingSubject)</pre>	Used by OC4J to allow the implementation class to perform the principal mapping. An application user subject is passed, and the implementation of this method should return a subject for use by the resource adapter to log in to the EIS resource per the J2CA 1.0 specifications. The implementation may return <code>null</code> if the proper resource principal cannot be determined.

When a connection to the EIS is created, OC4J invokes the mapping method with the initiating user as the `initiatingPrincipal`. The mapping method must return a `Subject` containing the resource principal and credential. The `Subject` that is returned must adhere to either option A or option B in section 8.2.6 of the Connector Architecture 1.0 specification.

OC4J also provides the abstract class `oracle.j2ee.connector.AbstractPrincipalMapping`. This class furnishes a default implementation of the `setManagedConnectionFactory()` and `setAuthenticationMechanism()` methods, as well as utility methods to determine whether the resource adapter supports the `BasicPassword` or Kerberos version 5 (Kerby5) authentication mechanisms, and a method for extracting the `Principal` from the application server user `Subject`. By extending the `oracle.j2ee.connector.AbstractPrincipalMapping` class, developers need only implement the `init` and `mapping` methods.

The methods exposed by the `oracle.j2ee.connector.AbstractPrincipalMapping` class appear in [Table 8-4](#).

Table 8-4 Method Description for `oracle.j2ee.connector.AbstractPrincipalMapping` Class

Method Signature	Description
<code>public abstract void init (java.util.Properties prop)</code>	This method must be implemented by the subclasses. See <code>PrincipalMapping</code> interface, described in Table 8-3 , for details.
<code>public void setManagedConnectionFactory (ManagedConnectionFactory mcf)</code>	Stores the <code>ManagedConnectionFactory</code> instance that is passed in. Subclasses need not implement this method, and can make use of the <code>getManagedConnectionFactory</code> object saved by this method.
<code>public void setAuthenticationMechanisms (java.util.Map authMechanisms)</code>	Stores the map of authentication mechanisms. Subclasses need not implement this mechanism. Instead, they can make use of the <code>isBasicPasswordSupported</code> or <code>isKerbv5Supported</code> methods to determine which authentication mechanism is supported by the resource adapter. The method <code>getAuthenticationMechanisms</code> can be used to retrieve the authentication mechanisms as well.
<code>public javax.security.auth.Subject mapping (javax.security.auth.Subject initiatingSubject)</code>	Used by OC4J to allow the implementation class to perform the principal mapping. An application user subject is passed, and the implementation of this method should return a subject for use by the resource adapter to log in to the EIS resource per the J2EE Connector Architecture specifications. The implementation may return null if the proper resource principal cannot be determined.
<code>public abstract javax.security.auth.Subject mapping (javax.security.auth.Subject initiatingSubject)</code>	This method must be implemented by the subclasses. See <code>PrincipalMapping</code> interface, described in Table 8-3 , for details.
<code>public ManagedConnectionFactory getManagedConnectionFactory()</code>	Utility method provided by this abstract class to return the <code>ManagedConnectionFactory</code> instance that might be required to create a <code>PasswordCredentials</code> object.
<code>public java.util.Map getAuthenticationMechanisms()</code>	Utility method to return the map of all authentication mechanisms supported by this resource adapter, as provided by OC4J. The key of the map returned is a string containing the supported mechanism type, such as <code>BasicPassword</code> or <code>Kerbv5</code> . The value is a string containing the corresponding credentials interface as declared in <code>ra.xml</code> , such as <code>javax.resource.spi.security.PasswordCredential</code> .
<code>public boolean isBasicPasswordSupported()</code>	Utility method to allow subclass to determine whether the <code>BasicPassword</code> authentication mechanism is supported by this resource adapter.
<code>public boolean isKerbv5Supported()</code>	Utility method to allow subclass to determine whether the <code>Kerbv5</code> authentication mechanism is supported by this resource adapter.
<code>public java.security.Principal getPrincipal (javax.security.auth.Subject subject)</code>	Utility method provided to extract the <code>Principal</code> object from the given application server user subject passed from OC4J

Extending `AbstractPrincipalMapping` This simple example demonstrates how to extend the `oracle.j2ee.connector.AbstractPrincipalMapping` abstract class to provide a principal mapping that always maps the user to the default user and password. Specify the

default user and password by using properties under the `<principal-mapping-interface>` element in `oc4j-ra.xml`.

The `PrincipalMapping` class is called `MyMapping`. It is defined as follows:

```
package com.acme.app;

import java.util.*;
import javax.resource.spi.*;
import javax.resource.spi.security.*;
import oracle.j2ee.connector.AbstractPrincipalMapping;
import javax.security.auth.*;
import java.security.*;

public class MyMapping extends AbstractPrincipalMapping
{
    String m_defaultUser;
    String m_defaultPassword;

    public void init(Properties prop)
    {
        if (prop != null)
        {
            // Retrieves the default user and password from the properties
            m_defaultUser = prop.getProperty("user");
            m_defaultPassword = prop.getProperty("password");
        }
    }

    public Subject mapping(Subject initiatingSubject)
    {
        // This implementation only supports BasicPassword authentication
        // mechanism. Return if the resource adapter does not support it.
        if (!isBasicPasswordSupported())
            return null;
        // Use the utility method to retrieve the Principal from the
        // OC4J user. This code is included here only as an example.
        // The principal obtained is not being used in this method.
        Principal principal = getPrincipal(initiatingSubject);
        char[] resPasswordArray = null;
        if (m_defaultPassword != null)
            resPasswordArray = m_defaultPassword.toCharArray();
        // Create a PasswordCredential using the default user name and
        // password, and add it to the Subject per option A in section
        // 8.2.6 in the Connector 1.0 spec.
        PasswordCredential cred =
            new PasswordCredential(m_defaultUser, resPasswordArray);
        cred.setManagedConnectionFactory(getManagedConnectionFactory());
        initiatingSubject.getPrivateCredentials().add(cred);
        return initiatingSubject;
    }
}
```

After you create your implementation class, copy a JAR file containing the class into the directory containing the decompressed RAR file. See [Table 8-2](#) for the location of the RAR file. After copying the file, edit `oc4j-ra.xml` to contain a `<principal-mapping-interface>` element for the new class.

For example:

```
<connector-factory name="..." location="...">
    ...
```

```

    <security-config>
      <principal-mapping-interface>
        <impl-class>com.acme.app.MyMapping</impl-class>
        <property name="user" value="scott" />
        <property name="password" value="tiger" />
      </principal-mapping-interface>
    </security-config>
    ...
  </connector-factory>

```

JAAS Pluggable Authentication Classes

You can also manage sign-on to the EIS programmatically with JAAS. OC4J furnishes a JAAS pluggable authentication framework that conforms to Appendix C in the Connector Architecture 1.0 specification. With this framework, an application server and its underlying authentication services remain independent from each other, and new authentication services can be plugged in without requiring modifications to the application server.

Some examples of authentication modules are:

- Principal Mapping JAAS module
- Credential Mapping JAAS module
- Kerberos JAAS module (for Caller Impersonation)

The JAAS login modules can be furnished by the customer, the EIS vendors, or the resource adapter vendors. Login modules must implement the `javax.security.auth.spi.LoginModule` interface, as documented in the Sun JAAS specification.

OC4J provides initiating user subjects to login modules by passing an instance of `javax.security.auth.Subject` containing any public certificates and an instance of an implementation of `java.security.Principal` representing the OC4J user. OC4J can pass a null `Subject` if there is no authenticated user (that is, an anonymous user). The initiating user subject is passed to the `initialize` method of the JAAS login module.

The JAAS login module's `login` method must, based on the initiating user, find the corresponding resource principal and create new `PasswordCredential` or `GenericCredential` instances for the resource principal. The resource principal and credential objects are then added to the initiating `Subject` in the `commit` method. The resource credential is passed to the `createManagedConnection` method in the `javax.resource.spi.ManagedConnectionFactory` implementation that is provided by the resource adapter.

If a null `Subject` is passed, the JAAS login module is responsible for creating a new `javax.security.auth.Subject` containing the resource principal and the appropriate credential.

JAAS and the `<connector-factory>` Element Each `<connector-factory>` element in `oc4j-ra.xml` can specify a different JAAS login module. Specify a name for the connector factory configuration in the `<jaas-module>` element. Here is an example of a `<connector-factory>` element in `oc4j-ra.xml` that uses JAAS login modules for container-managed sign-on:

```

<connector-factory connector-name="myBlackbox" location="eis/myEIS1">
  <description>Connection to my EIS</description>
  <config-property name="connectionURL"
    value="jdbc:oracle:thin:@localhost:5521:orcl" />

```

```
<security-config>
  <jaas-module>
    <jaas-application-name>JAASModuleDemo</jaas-application-name>
  </jaas-module>
</security-config>
</connector-factory>
```

In JAAS, you must specify which LoginModule to use for a particular application, and in what order to invoke the LoginModules. JAAS uses the value that are specified in the `<jaas-application-name>` element to look up LoginModules. See the *Oracle Application Server Containers for J2EE Security Guide* for more information.

Special Features Accessible Via Programmatic Interface

In addition to mapping from OC4J users to EIS users, login modules and OC4J-specific authentication classes can also map from OC4J groups to EIS users.

The `oracle.j2ee.connector` package contains the `InitiatingPrincipal` class that represents OC4J users and the `InitiatingGroup` class that represents OC4J groups. OC4J creates instances of `InitiatingPrincipal` and incorporates them into the Subject that is passed to the `initialize` method of the login modules as well as to the `mapping` method of the OC4J-specific authentication class.

The `oracle.j2ee.connector` package also contains the `InitiatingPrincipal` class that implements the `java.security.Principal` interface and adds the method `getGroups()`. The `getGroups` method returns a `java.util.Set` of `oracle.j2ee.connector.InitiatingGroup` objects, representing the OC4J groups or JAZN roles that this OC4J user belongs to. The group membership is defined in OC4J-specific descriptor files such as `principals.xml` or `jazn-data.xml`, depending on the user manager. The `oracle.j2ee.connector.InitiatingGroup` class implements but does not extend the functionality of the `java.security.Principal` interface.

Java Object Cache

This chapter describes the Oracle Application Server Containers for J2EE (OC4J) Java Object Cache, including its architecture and programming features. This chapter covers the following topics:

- [Java Object Cache Concepts](#)
- [Java Object Cache Object Types](#)
- [Java Object Cache Environment](#)
- [Developing Applications Using Java Object Cache](#)
- [Working with Disk Objects](#)
- [Working with StreamAccess Objects](#)
- [Working with Pool Objects](#)
- [Running in Local Mode](#)
- [Running in Distributed Mode](#)

Java Object Cache Concepts

Oracle Application Server 10g offers the Java Object Cache to help e-businesses manage Web site performance issues for dynamically generated content. The Java Object Cache improves the performance, scalability, and availability of Web sites running on Oracle Application Server 10g.

By storing frequently accessed or expensive-to-create objects in memory or on disk, the Java Object Cache eliminates the need to repeatedly create and load information within a Java program. The Java Object Cache retrieves content faster and greatly reduces the load on application servers.

The Oracle Application Server 10g cache architecture includes the following cache components:

- **Oracle Application Server Web Cache.** The Web Cache sits in front of the application servers (Web servers), caching their content and providing that content to Web browsers that request it. When browsers access the Web site, they send HTTP requests to the Web Cache. The Web Cache, in turn, acts as a virtual server to the application servers. If the requested content has changed, the Web Cache retrieves the new content from the application servers.

The Web Cache is an HTTP-level cache, maintained outside the application, providing fast cache operations. It is a pure, content-based cache, capable of caching static data (such as HTML, GIF, or JPEG files) or dynamic data (such as servlet or JSP results). Given that it exists as a flat content-based cache outside the

application, it cannot cache objects (such as Java objects or XML DOM—Document Object Model—objects) in a structured format. In addition, it offers relatively limited postprocessing abilities on cached data.

- **Java Object Cache.** The Java Object Cache provides caching for expensive or frequently used Java objects when the application servers use a Java program to supply their content. Cached Java objects can contain generated pages or can provide support objects within the program to assist in creating new content. The Java Object Cache automatically loads and updates objects as specified by the Java application.
- **Web Object Cache.** The Web Object Cache is a Web-application-level caching facility. It is an application-level cache, embedded and maintained within a Java Web application. The Web Object Cache is a hybrid cache, both Web-based and object-based. Using the Web Object Cache, applications can cache programmatically, using application programming interface (API) calls (for servlets) or custom tag libraries (for JSPs). The Web Object Cache is generally used as a complement to the Web cache. By default, the Web Object Cache uses the Java Object Cache as its repository.

A custom tag library or API enables you to define page fragment boundaries and to capture, store, reuse, process, and manage the intermediate and partial execution results of JSP pages and servlets as cached objects. Each block can produce its own resulting cache object. The cached objects can be HTML or XML text fragments, XML DOM objects, or Java serializable objects. These objects can be cached conveniently in association with HTTP semantics. Alternatively, they can be reused outside HTTP, such as in outputting cached XML objects through Simple Mail Transfer Protocol (SMTP), Java Message Service (JMS), Advanced Queueing (AQ), or Simple Object Access Protocol (SOAP).

Note: This chapter focuses on the Java Object Cache. For a full discussion of all three caches and their differences, see the *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Java Object Cache Basic Architecture

Figure 9–1 shows the basic architecture for the Java Object Cache. The cache delivers information to a user process. The process could be a servlet application that generates HTML pages, or any other Java application.

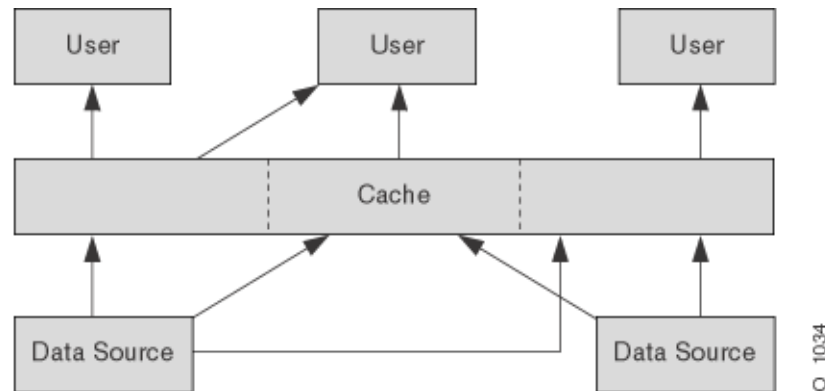
The Java Object Cache is an in-process, process-wide caching service for general application use. That is, objects are cached within the process memory space, and the Java Object Cache is a single service that is shared by all threads running in the process, in contrast to a service that runs in another process. The Java Object Cache can manage any Java object. To facilitate sharing of cached objects, all objects within the cache are accessed by name. The caching service does not impose a structure on objects being cached. The name, structure, type, and original source of the object are all defined by the application.

To maximize system resources, all objects within the cache are shared. However, access to cached objects is not serialized by access locks, allowing for a high level of concurrent access. When an object is invalidated or updated, the invalid version of the object remains in the cache as long as there are references to that particular version of the object. It is thus possible to have multiple versions of an object in the cache at the same time; however, there is never more than one valid version of the object. The old or invalid versions of an object are visible only to applications that had references to

the version before it was invalidated. If an object is updated, a new copy of the object is created in the cache, and the old version is marked as invalid.

Objects are loaded into the cache with a user-provided `CacheLoader` object. This loader object is called by the Java Object Cache when a user application requests an object from the cache and it is not already present. [Figure 9-1](#) is a graphical representation of the architecture. The application interacts with the cache to retrieve objects, and the cache interacts through the `CacheLoader` with the data source. This process gives a clean division between object creation and object use.

Figure 9-1 Java Object Cache Basic Architecture

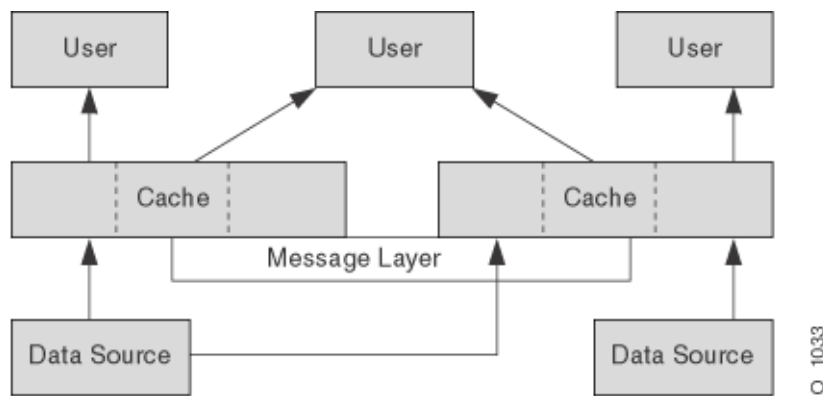


Distributed Object Management

The Java Object Cache can be used in an environment in which multiple Java processes are running the same application or working on behalf of the same application. In this environment, it is useful to have identical objects cached in different processes. For simplicity, availability and performance, the Java Object Cache is specific to each process. There is no centralized control of which objects are loaded into a process. However, the Java Object Cache coordinates object updating and invalidation between processes. If an object is updated or invalidated in one process, then it is also updated or invalidated in all other associated processes. This distributed management allows a system of processes to stay synchronized without the overhead of centralized control.

[Figure 9-2](#) is a graphical representation of the following:

- How the application interacts with the Java Object Cache to retrieve objects
- How the Java Object Cache interacts with the data source
- How the caches of the Java Object Cache coordinate cache events through the cache messaging system

Figure 9–2 Java Object Cache Distributed Architecture

How the Java Object Cache Works

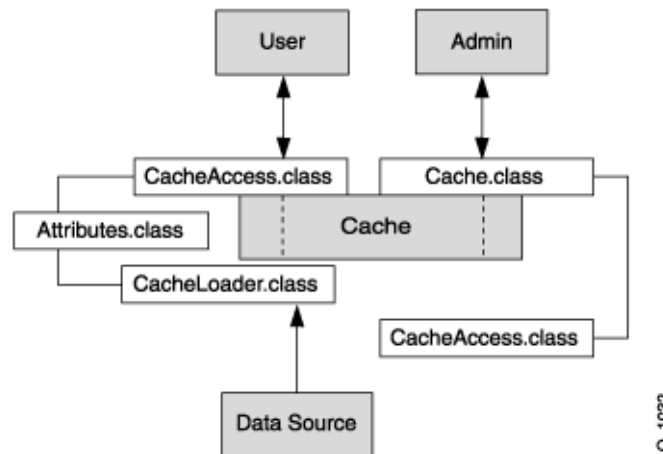
The Java Object Cache manages Java objects within a process, across processes, and on a local disk. The Java Object Cache provides a powerful, flexible, and easy-to-use service that significantly improves Java performance by managing local copies of Java objects. There are very few restrictions on the types of Java objects that can be cached or on the original source of the objects. Programmers use the Java Object Cache to manage objects that, without cache access, are expensive to retrieve or to create.

The Java Object Cache is easy to integrate into new and existing applications. Objects can be loaded into the object cache, using a user-defined object, the `CacheLoader`, and can be accessed through a `CacheAccess` object. The `CacheAccess` object supports local and distributed object management. Most of the functionality of the Java Object Cache does not require administration or configuration. Advanced features support configuration using administration APIs in the `Cache` class. Administration includes setting configuration options, such as naming local disk space or defining network ports. The administration features allow applications to fully integrate the Java Object Cache.

Each cached Java object has a set of associated attributes that control how the object is loaded into the cache, where the object is stored, and how the object is invalidated. Cached objects are invalidated based on time or an explicit request. (Notification can be provided when the object is invalidated.) Objects can be invalidated by group or individually.

[Figure 9–3](#) illustrates the basic Java Object Cache APIs. [Figure 9–3](#) does not show distributed cache management.

Figure 9–3 Java Object Cache Basic APIs



Cache Organization

The Java Object Cache is organized as follows:

- Cache Environment.** The cache environment includes cache regions, subregions, groups, and attributes. Cache regions, subregions, and groups associate objects and collections of objects. Attributes are associated with cache regions, subregions, groups, and individual objects. Attributes affect how the Java Object Cache manages objects.
- Cache Object Types.** The cache object types include memory objects, disk objects, pooled objects, and `StreamAccess` objects.

Table 9–1 contains a summary of the constructs in the cache environment and the cache object types.

Table 9–1 Cache Organizational Construct

Cache Construct	Description
Attributes	Functionality associated with cache regions, groups, and individual objects. Attributes affect how the Java Object Cache manages objects.
Cache region	An organizational name space for holding collections of cache objects within the Java Object Cache.
Cache subregion	An organizational name space for holding collections of cache objects within a parent region, subregion, or group.
Cache group	An organizational construct used to define an association between objects. The objects within a region can be invalidated as a group. Common attributes can be associated with objects within a group.
Memory object	An object that is stored and accessed from memory.
Disk object	An object that is stored and accessed from disk.
Pooled object	A set of identical objects that the Java Object Cache manages. The objects are checked out of the pool, used, and then returned.
<code>StreamAccess</code> object	An object that is loaded using a Java <code>OutputStream</code> and accessed using a Java <code>InputStream</code> . The object is accessed from memory or disk, depending on the size of the object and the cache capacity.

Java Object Cache Features

The Java Object Cache provides the following features:

- Objects can be updated or invalidated
- Objects can be invalidated either explicitly, or with an attribute specifying the expiration time or the idle time
- Objects can be coordinated between processes
- Object loading and creation can be automatic
- Object loading can be coordinated between processes
- Objects can be associated in cache regions or groups with similar characteristics
- Cache event notification provides for event handling and special processing
- Cache management attributes can be specified for each object or applied to cache regions or groups

Java Object Cache Object Types

This section describes the object types that the Java Object Cache manages:

- [Memory Objects](#)
- [Disk Objects](#)
- [StreamAccess Objects](#)
- [Pool Objects](#)

Note: Objects are identified by a name that can be any Java object. The Java object used for the identifying name must override the default Java object `equals` method and the default Java object `hashCode` method. If the object is distributed, and can be updated or saved to disk, the `Serializable` interface must be implemented.

Memory Objects

Memory objects are Java objects that the Java Object Cache manages. Memory objects are stored in the Java virtual machine (JVM) heap space as Java objects. Memory objects can hold HTML pages, the results of a database query, or any information that can be stored as a Java object.

Memory objects are usually loaded into the Java Object Cache with an application-supplied loader. The source of the memory object can be external (for example, using data in a table on the Oracle9i Database Server). The application supplied loader accesses the source and either creates or updates the memory object. Without the Java Object Cache, the application would be responsible for accessing the source directly, rather than using the loader.

You can update a memory object by obtaining a private copy of the memory object, applying the changes to the copy, and then placing the updated object back in the cache (using the `CacheAccess.replace()` method). This replaces the original memory object.

The `CacheAccess.defineObject()` method associates attributes with an object. If attributes are not defined, then the object inherits the default attributes from its associated region, subregion, or group.

An application can request that a memory object be spooled to a local disk (using the `SPOOL` attribute). Setting this attribute allows the Java Object Cache to handle memory objects that are large, or costly to re-create and seldom updated. When the disk cache is set up to be significantly larger than the memory cache, objects on disk stay in the disk cache longer than objects in memory.

Combining memory objects that are spooled to a local disk with the distributed feature from the `DISTRIBUTE` attribute provides object persistence (when the Java Object Cache is running in distributed mode). Object persistence allows objects to survive the restarting of the JVM.

Disk Objects

Disk objects are stored on a local disk and are accessed directly from the disk by the application using the Java Object Cache. Disk objects can be shared across Java Object Cache processes, or they can be local to a particular process, depending on disk location specified and the setting for the `DISTRIBUTE` attribute (and whether the Java Object Cache is running in distributed or local mode).

Disk objects can be invalidated explicitly or by setting the `TimeToLive` or `IdleTime` attributes. When the Java Object Cache requires additional space, disk objects that are not being referenced can be removed from the cache.

StreamAccess Objects

StreamAccess objects are special cache objects set up to be accessed using the Java `InputStream` and `OutputStream` classes. The Java Object Cache determines how to access the `StreamAccess` object, based on the size of the object and the capacity of the cache. Smaller objects are accessed from memory; larger objects are streamed directly from disk. All `streamAccess` objects are stored on disk.

The cache user's access to the `StreamAccess` object is through an `InputStream`. All the attributes that apply to memory objects and disk objects also apply to `StreamAccess` objects. A `StreamAccess` object does not supply a mechanism to manage a stream—for example, `StreamAccess` objects cannot manage socket endpoints. `InputStream` and `OutputStream` objects are available to access fixed-sized, potentially large objects.

Pool Objects

Pool objects are a special class of objects that the Java Object Cache manages. A pool object contains a set of identical object instances. The pool object itself is a shared object; the objects within the pool are private objects. Individual objects within the pool can be checked out to be used and then returned to the pool when they are no longer needed.

Attributes, including `TimeToLive` or `IdleTime` can be associated with a pool object. These attributes apply to the pool object as a whole.

The Java Object Cache instantiates objects within a pool using an application-defined factory object. The size of a pool decreases or increases based on demand and on the values of the `TimeToLive` or `IdleTime` attributes. A minimum size for the pool is specified when the pool is created. The minimum size value is interpreted as a request rather than a guaranteed minimum value. Objects within a pool object are subject to

removal from the cache because of a lack of space, so the pool can decrease below the requested minimum value. A maximum pool size value can be set by putting a hard limit on the number of objects available in the pool.

Java Object Cache Environment

The Java Object Cache environment includes the following:

- [Cache Regions](#)
- [Cache Subregions](#)
- [Cache Groups](#)
- [Region and Group Size Control](#)
- [Cache Object Attributes](#)

This section describes these Java Object Cache environment constructs.

Cache Regions

The Java Object Cache manages objects within a cache region. A *cache region* defines a name space within the cache. Each object within a cache region must be uniquely named, and the combination of the cache region name and the object name must uniquely identify an object. Thus, cache region names must be unique from other region names, and all objects within a region must be uniquely named relative to the region. (Multiple objects can have the same name if they are within different regions or subregions.)

You can define as many regions as you need to support your application. However, most applications require only one region. The Java Object Cache provides a default region; when a region is not specified, objects are placed in the default region.

Attributes can be defined for a region and are then inherited by the objects, subregions, and groups within the region.

Cache Subregions

The Java Object Cache manages objects within a cache region. Specifying a subregion within a cache region defines a child hierarchy. A *cache subregion* defines a name space within a cache region or within a higher cache subregion. Each object within a cache subregion must be uniquely named, and the combination of the cache region name, the cache subregion name, and the object name must uniquely identify an object.

You can define as many subregions as you need to support your application.

A subregion inherits its attributes from its parent region or subregion unless the attributes are defined when the subregion is defined. A subregion's attributes are inherited by the objects within the subregion. If a subregion's parent region is invalidated or destroyed, the subregion is also invalidated or destroyed.

Cache Groups

A *cache group* creates an association between objects within a region. Cache groups allow related objects to be manipulated together. Objects are typically associated in a cache group because they must be invalidated together, or they use common attributes. Any set of cache objects within the same region or subregion can be associated using a cache group, which can, in turn, include other cache groups.

A Java Object Cache object can belong to only one group at any given time. Before an object can be associated with a group, the group must be explicitly created. A group is defined with a name. A group can have its own attributes, or it can inherit its attributes from its parent region, subregion, or group.

Group names are not used to identify individual objects, but rather to define a set or collection of objects that have something in common. A group does not define a hierarchical name space. Object type does not distinguish objects for naming purposes; therefore, a region cannot include a group and a memory object with the same name. You must use subregions to define a hierarchical name space within a region.

Groups can contain groups, with the groups having a parent and child relationship. The child group inherits attributes from the parent group.

Region and Group Size Control

With the Java Object Cache, you can specify the maximum size of a region or group as either the number of objects in the region or group, or the maximum number of bytes allowed. If the number of bytes controls the region capacity, then set the size attribute for all objects in the region. This can be set either directly by the user when the object is created, or automatically by setting the `Attributes.MEASURE` attribute flag. You can set the size of a region or group at multiple levels in the naming hierarchy—that is, at the region and subregion level, or at the group level within a region or another group.

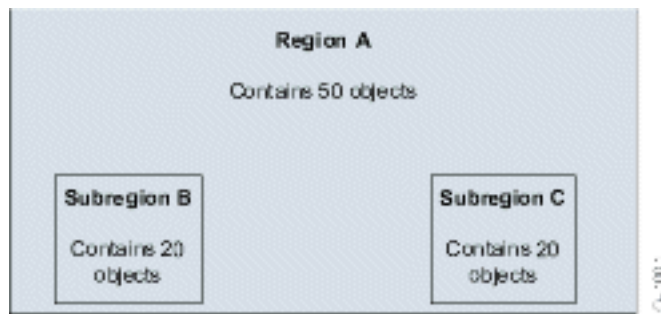
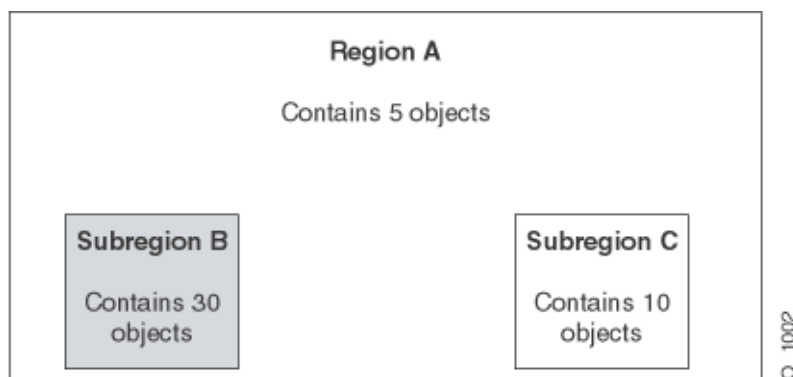
When the capacity of a region or group is reached, the `CapacityPolicy` object associated with that region or group, if defined, is called. If no capacity policy has been specified, then the default policy is used. The default policy follows: If a nonreferenced object of lesser or equal priority is found, then it is invalidated in favor of the new object. If the priority attribute has not been set for an object, then the priority is assumed to be `Integer.MAX_VALUE`. When searching for an object to remove, all objects in the immediate region or group and all subregions and subgroups are searched. The first object that can be removed, based on the capacity policy, is removed. So, for example, this may not be the object of lowest priority in the search area.

[Figure 9–4](#) and [Figure 9–5](#) give examples. In each illustration, the grayed portions indicate the search area.

The capacity of region A is set to 50 objects, with subregion B and subregion C set to 20 objects each. If the object count of region A reaches 50, with 10 directly in region A and 20 each in subregions B and C, then the capacity policy for region A is called. The object that is removed can come from region A or from one of its subregions.

[Figure 9–4](#) shows this situation.

If subregion B reaches 20 before the capacity of region A is reached, then the capacity policy for subregion B is called, and only objects within subregion B are considered for removal. [Figure 9–5](#) shows this situation.

Figure 9–4 Capacity Policy Example, Part 1**Figure 9–5 Capacity Policy Example, Part 2**

Cache Object Attributes

Cache object *attributes* affect how the Java Object Cache manages objects. Each object, region, subregion, and group has a set of associated attributes. An object's applicable attributes contain either the default attribute values; the attribute values inherited from the object's parent region, subregion, or group; or the attribute values that you select for the object.

Attributes fall into two categories:

- The first category is attributes that must be defined before an object is loaded into the cache. [Table 9–2](#) summarizes these attributes. None of the attributes shown in [Table 9–2](#) has a corresponding `set` or `get` method, except the `LOADER` attribute. Use the `Attributes.setFlags()` method to set these attributes.
- The second category is attributes that can be modified after an object is stored in the cache. [Table 9–3](#) summarizes these attributes.

Note: Some attributes do not apply to certain types of objects. See the "Object Types" sections in the descriptions in [Table 9–2](#) and [Table 9–3](#).

Using Attributes Defined Before Object Loading

The attributes shown in [Table 9–2](#) must be defined for an object before the object is loaded. These attributes determine an object's basic management characteristics.

The following list shows the methods that you can use to set the attributes shown in [Table 9–2](#) (by setting the values of an `Attributes` object argument).

- `CacheAccess.defineRegion()`
- `CacheAccess.defineSubRegion()`
- `CacheAccess.defineGroup()`
- `CacheAccess.defineObject()`
- `CacheAccess.put()`
- `CacheAccess.createPool()`
- `CacheLoader.createDiskObject()`
- `CacheLoader.createStream()`
- `CacheLoader.SetAttributes()`

Note: You cannot reset the attributes shown in [Table 9–2](#) by using the `CacheAccess.resetAttributes()` method.

Table 9–2 Java Object Cache Attributes—Set at Object Creation

Attribute Name	Description
DISTRIBUTE	<p>Specifies whether an object is local or distributed. When using the Java Object Cache distributed-caching feature, an object is set as a local object so that updates and invalidations are not propagated to other caches in the site.</p> <p>Object Types: When set on a region, subregion, or a group, this attribute sets the default value for the <code>DISTRIBUTE</code> attribute for the objects within the region, subregion, or group unless the objects explicitly set their own <code>DISTRIBUTE</code> attribute. Because pool objects are always local, this attribute does not apply to pool objects.</p> <p>Default Value: All objects are local.</p>
GROUP_TTL_DESTROY	<p>Indicates that the associated object, group, or region should be destroyed when the <code>TimeToLive</code> expires.</p> <p>Object Types: When set on a region or a group, all the objects within the region or group, and the region, subregion, or group itself are destroyed when the <code>TimeToLive</code> expires.</p> <p>Default Value: Only group member objects are invalidated when the <code>TimeToLive</code> expires.</p>
LOADER	<p>Specifies the <code>CacheLoader</code> associated with the object.</p> <p>Object Types: When set on a region or group, the specified <code>CacheLoader</code> becomes the default loader for the region, subregion, or group. The <code>LOADER</code> attribute is specified for each object within the region or the group.</p> <p>Default Value: Not set.</p>
ORIGINAL	<p>Indicates that the object was created in the cache, rather than loaded from an external source. <code>ORIGINAL</code> objects are not removed from the cache when the reference count goes to zero. <code>ORIGINAL</code> objects must be explicitly invalidated when they are no longer useful.</p> <p>Object Types: When set on a region or group, this attribute sets the default value for the <code>ORIGINAL</code> attribute for the objects within the region, subregion, or group, unless the objects set their own <code>ORIGINAL</code> attribute.</p> <p>Default Value: Not set.</p>

Table 9–2 (Cont.) Java Object Cache Attributes—Set at Object Creation

Attribute Name	Description
REPLY	<p>Specifies that a reply message will be sent from remote caches after a request for an object update or invalidation has completed. Set this attribute when a high level of consistency is required between caches. If the <code>DISTRIBUTE</code> attribute is not set, or the cache is started in non-distributed mode, <code>REPLY</code> is ignored.</p> <p>Object Types: When set on a region or group, this attribute sets the default value for the <code>REPLY</code> attribute for the objects within the region, subregion, or group, unless the objects explicitly set their own <code>REPLY</code> attribute. For memory, <code>StreamAccess</code>, and disk objects, this attribute applies only when the <code>DISTRIBUTE</code> attribute is set to the value <code>DISTRIBUTE</code>. Because pool objects are always local, this attribute does not apply for pool objects.</p> <p>Default Value: No reply is sent. When <code>DISTRIBUTE</code> is set to local, the <code>REPLY</code> attribute is ignored.</p>
SPOOL	<p>Specifies that a memory object should be stored on disk rather than being lost when the cache system removes it from memory to regain space. This attribute applies only to memory objects. If the object is also distributed, the object can survive the death of the process that spooled it. Local objects are accessible only by the process that spools them, so if the Java Object Cache is not running in distributed mode, the spooled object is lost when the process dies.</p> <p>Note: An object must be serializable to be spooled.</p> <p>Object Types: When set on a region, subregion, or group, this attribute sets the default value for the <code>SPOOL</code> attribute for the objects within the region, subregion, or group unless the objects set their own <code>SPOOL</code> attribute.</p> <p>Default Value: Memory objects are not stored to disk.</p>
SYNCHRONIZE	<p>This attribute indicates that updates to this object must be synchronized. If this flag is set, only the "owner" of an object can load or replace the object. Ownership is obtained using the <code>CacheAccess.getOwnership()</code> method. The "owner" of an object is the <code>CacheAccess</code> object. Setting the <code>SYNCHRONIZE</code> attribute does not prevent a user from reading or invalidating the object.</p> <p>Object Types: When set on a region, subregion, or group, the ownership restriction is applied to the region, subregion, or group as a whole. Pool objects do not use this attribute.</p> <p>Default Value: Updates are not synchronized.</p>
SYNCHRONIZE_DEFAULT	<p>Indicates that all objects in a region, subregion, or group should be synchronized. Each user object in the region, subregion, or group is marked with the <code>SYNCHRONIZE</code> attribute. Ownership of the object must be obtained before the object can be loaded or updated.</p> <p>Setting the <code>SYNCHRONIZE_DEFAULT</code> attribute does not prevent a user from reading or invalidating objects. Thus, ownership is not required for reads or invalidation of objects that have the <code>SYNCHRONIZE</code> attribute set.</p> <p>Object Types: When set on a region, subregion, or group, ownership is applied to individual objects within the region, subregion, or group. Pool objects do not use this attribute.</p> <p>Default Value: Updates are not synchronized.</p>
ALLOWNULL	<p>Specifies that the cache accepts null as a valid value for the affected objects. Null objects that are returned by a <code>cacheLoader</code> object are cached, rather than generating an <code>ObjectNotFoundException</code>.</p> <p>Object Types: When set on a region, subregion, group, or pool, this attribute applies individually to each object within the region, subregion, group, or pool unless explicitly set for the object.</p> <p>Default Value: OFF. (Nulls are not allowed.)</p>

Table 9–2 (Cont.) Java Object Cache Attributes—Set at Object Creation

Attribute Name	Description
MEASURE	<p>Specifies that the size attribute of the cached object is calculated, automatically, when the object is loaded or replaced in the cache. The capacity of the cache or region can then be accurately controlled based on object size, rather than object count.</p> <p>Object Types: When set on a region, subregion, group, or pool, this attribute applies individually to each object within the region, subregion, group, or pool unless explicitly set for the object.</p> <p>Default Value: OFF. (The size of an object is not automatically calculated.)</p>
CapacityPolicy	<p>Specifies the CapacityPolicy object to be used to control the size of the region or group. This attribute is ignored if set for an individual object.</p> <p>Object Types: When set on a region, subregion, or group, this attribute applies to the entire region or group. This attribute is not applicable to individual objects or pools.</p> <p>Default Value: OFF. (No capacity policy is defined for a region or group. If the region or group reaches capacity, the first nonreferenced object in the region or group is invalidated.)</p>

Using Attributes Defined Before or After Object Loading

A set of Java Object Cache attributes can be modified either before or after object loading. [Table 9–3](#) lists these attributes. These attributes can be set using the methods in the list under "[Using Attributes Defined Before Object Loading](#)" on page 9-10, and can be reset using the `CacheAccess.resetAttributes()` method.

Table 9–3 Java Object Cache Attributes

Attribute Name	Description
DefaultTimeToLive	<p>Establishes a default value for the TimeToLive attribute that is applied to all objects individually within the region, subregion, or group. This attribute applies only to regions, subregions, and groups. This value can be overridden by setting the TimeToLive on individual objects.</p> <p>Object Types: When set on a region, subregion, group, or pool, this attribute applies to all the objects within the region, subregion, group, or pool unless the objects explicitly set their own TimeToLive.</p> <p>Default Value: No automatic invalidation.</p>
IdleTime	<p>Specifies the amount of time an object can remain idle, with a reference count of 0, in the cache before being invalidated. If the TimeToLive or DefaultTimeToLive attribute is set, the IdleTime attribute is ignored.</p> <p>Object Types: When set on a region, subregion, group, or pool, this attribute applies individually to each object within the region, subregion, group, or pool unless the object explicitly sets IdleTime.</p> <p>Default Value: No automatic IdleTime invalidation.</p>
CacheEventListener	<p>Specifies the CacheEventListener associated with the object.</p> <p>Object Types: When set on a region, subregion, or a group, the specified CacheEventListener becomes the default CacheEventListener for the region, subregion, or group unless a CacheEventListener is specified individually on objects within the region, subregion, or group.</p> <p>Default Value: No CacheEventListener is set.</p>

Table 9–3 (Cont.) Java Object Cache Attributes

Attribute Name	Description
TimeToLive	<p>Establishes the maximum amount of time that an object remains in the cache before being invalidated. If associated with a region, subregion, or group, all objects in the region, subregion, or group are invalidated when the time expires. If the region, subregion, or group is not destroyed (that is, if <code>GROUP_TTL_DESTROY</code> is not set), the <code>TimeToLive</code> value is reset.</p> <p>Object Types: When set for a region, subregion, group, or pool, this attribute applies to the region, subregion, group, or pool, as a whole, unless the objects explicitly set their own <code>TimeToLive</code>.</p> <p>Default Value: No automatic invalidation.</p>
Version	<p>An application can set a <code>Version</code> for each instance of an object in the cache. The <code>Version</code> is available for application convenience and verification. The caching system does not use this attribute.</p> <p>Object Types: When set on a region, subregion, group, or pool, this attribute applies to all the objects within the region, subregion, group, or pool unless the objects explicitly set their own <code>Version</code>.</p> <p>Default Value: The default <code>Version</code> is 0.</p>
Priority	<p>Controls which objects are removed from the cache or region when its capacity has been reached. This attribute, an integer, is made available to the <code>CapacityPolicy</code> object used to control the size of the cache, region, or group. The larger the number, the higher the priority. For region and group capacity control, when an object is removed to make room, specifically for another object, an object of higher priority is never removed to allow an object of lower priority to be cached. For the cache capacity control, lower priority objects are chosen for eviction over higher priority.</p> <p>Object Types: When set on a region, subregion, group, or pool, this attribute applies individually to each object within the region, subregion, group, or pool unless explicitly set for the object.</p> <p>Default Value: <code>integer.MAX_VALUE</code>.</p>
MaxSize	<p>Specifies the maximum number of bytes available for a region or group. If this attribute is specified for an object, it is ignored.</p> <p>Object Types: When set on a region, subregion, or group, this attribute applies to the entire region or group. This attribute is not applicable to individual objects or pools.</p> <p>Default Value: No limit.</p>
MaxCount	<p>Specifies the maximum number of objects that can be stored in a region or group. If this attribute is specified for an object, it is ignored.</p> <p>Object Types: When set on a region, subregion, or group, this attribute applies to the entire region or group. This attribute is not applicable to individual objects or pools.</p> <p>Default Value: No limit.</p>
User-defined attributes	<p>Attributes can be defined by the user. These are name-value pairs that are associated with the object, group, or region. They are intended to be used in conjunction with a <code>CapacityPolicy</code> object, although they can be defined as needed by the cache user.</p> <p>Object Types: When set on a region, subregion, group, or pool, these attributes are available to each object within the region, subregion, group, or pool unless explicitly reset for the object.</p> <p>Default Value: No user-defined attributes are set by default.</p>

Developing Applications Using Java Object Cache

This section describes how to develop applications that use the Java Object Cache. This section covers the following topics:

- [Cache Object Attributes](#)
- [Importing Java Object Cache](#)
- [Defining a Cache Group](#)
- [Defining a Cache Subregion](#)
- [Defining and Using Cache Objects](#)
- [Implementing a CacheLoader Object](#)
- [Invalidating Cache Objects](#)
- [Destroying Cache Objects](#)
- [Multiple Object Loading and Invalidation](#)
- [Java Object Cache Configuration](#)
- [Declarative Cache](#)
- [Capacity Control](#)
- [Implementing a Cache Event Listener](#)
- [Restrictions and Programming Pointers](#)

Importing Java Object Cache

The Oracle installer installs the Java Object Cache JAR file `cache.jar` in the directory `$ORACLE_HOME/javacache/lib` on UNIX or in `%ORACLE_HOME%\javacache\lib` on Windows.

To use the Java Object Cache, import `oracle.ias.cache`, as follows:

```
import oracle.ias.cache.*;
```

Defining a Cache Region

All access to the Java Object Cache is through a `CacheAccess` object, which is associated with a cache region. You define a cache region, usually associated with the name of an application, using the `CacheAccess.defineRegion()` static method. If the cache has not been initialized, then `defineRegion()` initializes the Java Object Cache.

When you define the region, you can also set attributes. Attributes specify how the Java Object Cache manages objects. The `Attributes.setLoader()` method sets the name of a cache loader. [Example 9-1](#) shows this.

Example 9-1 *Setting the Name of a CacheLoader*

```
Attributes attr = new Attributes();
MyLoader mloader = new MyLoader();
attr.setLoader(mloader);
attr.setDefaultTimeToLive(10);

final static String APP_NAME_ = "Test Application";
CacheAccess.defineRegion(APP_NAME_, attr);
```

The first argument for `defineRegion` uses a `String` to set the region name. This static method creates a private region name within the Java Object Cache. The second argument defines the attributes for the new region using the default cache attributes.

Defining a Cache Group

Create a cache group when you want to create an association between two or more objects within the cache. Objects are typically associated in a cache group because they must be invalidated together or because they have a common set of attributes.

Any set of cache objects within the same region or subregion can be associated using a cache group, including other cache groups. Before an object can be associated with a cache group, the cache group must be defined. A cache group is defined with a name and can use its own attributes, or it can inherit attributes from its parent cache group, subregion, or region. The code in [Example 9-2](#) defines a cache group within the region named `Test Application`:

Example 9-2 *Defining a Cache Group*

```
final static String APP_NAME_ = "Test Application";
final static String GROUP_NAME_ = "Test Group";
// obtain an instance of CacheAccess object to a named region
CacheAccess caccess = CacheAccess.getAccess(APP_NAME_);
// Create a group
ccaccess.defineGroup(GROUP_NAME_);
// Close the CacheAccess object
ccaccess.close();
```

Defining a Cache Subregion

Define a subregion when you want to create a private name space within a region or within a previously defined subregion. The name space of a subregion is independent of the parent name space. A region can contain two objects with the same name, as long as the objects are within different subregions.

A subregion can contain anything that a region can contain, including cache objects, groups, or additional subregions. Before an object can be associated with a subregion, the subregion must be defined. A cache subregion is defined with a name and can use its own attributes, or it can inherit attributes from its parent cache region or subregion. Use the `getParent()` method to obtain the parent of a subregion.

The code in [Example 9-3](#) defines a cache subregion within the region named `Test Application`.

Example 9-3 *Defining a Cache Subregion*

```
final static String APP_NAME_ = "Test Application";
final static String SUBREGION_NAME_ = "Test SubRegion";
// obtain an instance of CacheAccess object to a named region
CacheAccess caccess = CacheAccess.getAccess(APP_NAME_);
// Create a SubRegion
ccaccess.defineSubRegion(SUBREGION_NAME_);
// Close the CacheAccess object
ccaccess.close();
```

Defining and Using Cache Objects

You may sometimes want to describe to the Java Object Cache how an individual object should be managed within the cache before the object is loaded. You can specify management options when the object is loaded, by setting attributes within the `CacheLoader.load()` method. However, you can also associate attributes with an object by using the `CacheAccess.defineObject()` method. If attributes are not defined for an object, then the Java Object Cache uses the default attributes set for the region, subregion, or group with which the object is associated.

[Example 9-4](#) shows how to set attributes for a cache object. The example assumes that the region `APP_NAME_` has already been defined.

Example 9-4 Setting Cache Attributes

```
import oracle.ias.cache.*;
final static String APP_NAME_ = "Test Application";
CacheAccess cacc = null;
try
{
    cacc = CacheAccess.getAccess(APP_NAME_);
    // set the default IdleTime for an object using attributes
    Attributes attr = new Attributes();
    // set IdleTime to 2 minutes
    attr.setIdleTime(120);

    // define an object and set its attributes
    cacc.defineObject("Test Object", attr);

    // object is loaded using the loader previously defined on the region
    // if not already in the cache.
    result = (String)cacc.get("Test Object");
} catch (CacheException ex){
    // handle exception
} finally {
    if (cacc!= null)
        cacc.close();
}
```

Implementing a CacheLoader Object

The Java Object Cache has two mechanisms for loading an object into the cache:

- The object can be put into the cache directly by the application using the `CacheAccess.put()` method.
- You can implement a `CacheLoader` object.

In most cases, implementing the `CacheLoader` is the preferred method. With a cache loader, the Java Object Cache automatically determines if an object needs to be loaded into the cache when the object is requested. And the Java Object Cache coordinates the load if multiple users request the object at the same time.

A `CacheLoader` object can be associated with a region, subregion, group, or object. Using a `CacheLoader` allows the Java Object Cache to schedule and manage object loading, and handle the logic for "if the object is not in cache then load."

If an object is not in the cache, then when an application calls the `CacheAccess.get()` or `CacheAccess.preLoad()` method, the cache executes the `CacheLoader.load` method. When the `load` method returns, the Java Object Cache inserts the returned object into the cache. Using `CacheAccess.get()`, if the cache is

full, the object is returned from the loader, and the object is immediately invalidated in the cache. (Therefore, using the `CacheAccess.get()` method with a full cache does not generate a `CacheFullException`.)

When a `CacheLoader` is defined for a region, subregion, or group, it is taken to be the default loader for all objects associated with the region, subregion, or group. A `CacheLoader` object that is defined for an individual object is used only to load the object.

Note: A `CacheLoader` object that is defined for a region, subregion, or group or for more than one cache object must be written with concurrent access in mind. The implementation should be thread-safe, because the `CacheLoader` object is shared.

Using CacheLoader Helper Methods

The `CacheLoader` cache provides several helper methods that you can use from within the `load()` method implementation. [Table 9-4](#) summarizes the available `CacheLoader` methods.

Table 9-4 *CacheLoader Methods Used in load()*

Method	Description
<code>setAttributes()</code>	Sets the attributes for the object being loaded.
<code>netSearch()</code>	Searches other available caches for the object to load. Objects are uniquely identified by the region name, subregion name, and the object name.
<code>getName()</code>	Returns the name of the object being loaded.
<code>getRegion()</code>	Returns the name of the region associated with the object being loaded.
<code>createStream()</code>	Creates a <code>StreamAccess</code> object.
<code>createDiskObject()</code>	Creates a disk object.
<code>exceptionHandler()</code>	Converts noncache exceptions into <code>CacheExceptions</code> , with the base set to the original exception.
<code>log()</code>	Records messages in the cache service log.

[Example 9-5](#) illustrates a `CacheLoader` object using the `cacheLoader.netSearch()` method to check whether the object being loaded is available in distributed Java Object Cache caches. If the object is not found using `netSearch()`, then the load method uses a more expensive call to retrieve the object. (An expensive call may involve an HTTP connection to a remote Web site or a connection to the Oracle9i Database Server.) For this example, the Java Object Cache stores the result as a `String`.

Example 9-5 *Implementing a CacheLoader*

```
import oracle.ias.cache.*;
class YourObjectLoader extends CacheLoader{
    public YourObjectLoader () {
    }
    public Object load(Object handle, Object args) throws CacheException
    {
        String contents;
        // check if this object is loaded in another cache
```



```

    try {
        contents = (String)netSearch(handle, 5000); // wait for up to 5 secs
        return new String(contents);
    } catch(ObjectNotFoundException ex){}

    try {
        contents = expensiveCall(args);
        return new String(contents);
    } catch (Exception ex) {throw exceptionHandler("Loadfailed", ex);}
}

private String expensiveCall(Object args) {
    String str = null;
    // your implementation to retrieve the information.
    // str = ...
    return str;
}
}

```

Invalidating Cache Objects

An object can be removed from the cache either by setting the `TimeToLive` attribute for the object, group, subregion, or region, or by explicitly invalidating or destroying the object.

Invalidating an object marks the object for removal from the cache. Invalidating a region, subregion, or group invalidates all the individual objects from the region, subregion, or group, leaving the environment—including all groups, loaders, and attributes—available in the cache. Invalidating an object does not undefine the object. The object loader remains associated with the name. To completely remove an object from the cache, use the `CacheAccess.destroy()` method.

An object can be invalidated automatically based on the `TimeToLive` or `IdleTime` attribute. When the `TimeToLive` or `IdleTime` expires, objects are, by default, invalidated and not destroyed.

If an object, group, subregion, or region is defined as distributed, the invalidate request is propagated to all caches in the distributed environment.

To invalidate an object, group, subregion, or region, use the `CacheAccess.invalidate()` method as follows:

```

CacheAccess cacc = CacheAccess.getAccess("Test Application");
cacc.invalidate("Test Object"); // invalidate an individual object
cacc.invalidate("Test Group"); // invalidate all objects associated with a group
cacc.invalidate(); // invalidate all objects associated with the region cacc
cacc.close(); // close the CacheAccess handle

```

Destroying Cache Objects

An object can be removed from the cache either by setting the `TimeToLive` attribute for the object, group, subregion, or region, or by explicitly invalidating or destroying the object.

Destroying an object marks the object and the associated environment, including any associated loaders, event handlers, and attributes for removal from the cache. Destroying a region, subregion, or a group marks all objects associated with the region, subregion, or group for removal, including the associated environment.

An object can be destroyed automatically based on the `TimeToLive` or `IdleTime` attributes. By default, objects are invalidated and are not destroyed. If the objects must

be destroyed, set the attribute `GROUP_TTL_DESTROY`. Destroying a region also closes the `CacheAccess` object used to access the region.

To destroy an object, group, subregion, or region, use the `CacheAccess.destroy()` method as follows:

```
CacheAccess cacc = CacheAccess.getAccess("Test Application");
cacc.destroy("Test Object"); // destroy an individual object
cacc.destroy("Test Group"); // destroy all objects associated with
                             // the group "Test Group"

cacc.destroy();           // destroy all objects associated with the region
                           // including groups and loaders
```

Multiple Object Loading and Invalidation

In most cases, objects are loaded into the cache individually; in some cases, however, multiple objects can be loaded into the cache as a set. The primary example of this is when multiple cached objects can be created from a single read from a database. In this case, it is much more efficient to create multiple objects from a single call to the `CacheLoader.load` method.

To support this scenario, the abstract class `CacheListLoader` and the method `CacheAccess.loadList` have been added. The `CacheListLoader` object extends the `CacheLoader` object defining the abstract method `loadList` and the helper methods `getNextObject`, `getList`, `getNamedObject`, and `saveObject`. The cache user implements the `CacheListLoader.loadList` method. Employing the helper methods, the user can iterate through the list of objects, creating each one and saving it to the cache. If the helper methods defined in `CacheLoader` are used from the `CacheListLoader` method, then `getNextObject` or `getNamedObject` should be called first to set the correct context.

The `CacheAccess.loadList` method takes as an argument an array of object names to be loaded. The cache processes this array of objects. Any objects that are not currently in the cache are added to a list that is passed to the `CacheListLoader` object that is defined for the cached objects. If a `CacheListLoader` object is not defined for the objects or the objects have different `CacheListLoader` objects defined, then each object is loaded individually, using the `CacheLoader.load` method defined.

It is always best to implement both the `CacheListLoader.loadList` method and the `CacheListLoader.load` method. Which method is called depends on the order of the user requests to the cache. For example, if the `CacheAccess.get` method is called before the `CacheAccess.loadList` method, then the `CacheListLoader.load` method is used rather than the `CacheAccess.loadList` method.

As a convenience, the `invalidate` and `destroy` methods have been overloaded to also handle an array of objects.

[Example 9-6](#) shows a sample `CacheListLoader`, and [Example 9-7](#) shows sample usage.

Example 9-6 Sample `CacheListLoader`

```
Public class ListLoader extends CacheListLoader
{
    public void loadList(Object handle, Object args) throws CacheException
    {
        while(getNextObject(handle) != null)
        {
```

```

        // create the cached object based on the name of the object
        Object cacheObject = myCreateObject(getName(handle));
        saveObject(handle, cacheObject);
    }
}

public Object load(Object handle, Object args) throws CacheException
{
    return myCreateObject(getName(handle));
}

private Object myCreateObject(Object name)
{
    // do whatever to create the object
}
}

```

Example 9-7 Sample Usage

```

// Assumes the cache has already been initialized

CacheAccess cacc;
Attributes attr;
ListLoader loader = new
ListLoader();
String objList[];
Object obj;

// set the CacheListLoader for the region
attr = new Attributes();
attr.setLoader(loader);

//define the region and get access to the cache
CacheAccess.defineRegion("region name", attr);
cacc = CacheAccess.getAccess("region name");

// create the array of object names
objList = new String[3];
for (int j = 0; j < 3; j++)
    objList[j] = "object " + j;

// load the objects in the cache via the CacheListLoader.loadList method
cacc.loadList(objList);

// retrieve the already loaded object from the cache
obj = cacc.get(objList[0]);

// do something useful with the object

// load an object using the CacheListLoader.load method
obj = cache.get("another object")

// do something useful with the object

```

Java Object Cache Configuration

The Java Object Cache is NOT initialized automatically upon OC4J startup. You can force OC4J to initialize jcache by using `-Doracle.ias.jcache=true` in `opmn.xml`, as shown in the following example:

```

<ias-component id="OC4J">
  <process-type id="home" module-id="OC4J" status="enabled">
    <module-data>
      <category id="start-parameters">
        <data id="java-options" value="-server
-Djava.security.policy=$ORACLE_HOME/j2ee/home/config/java2.policy
-Djava.awt.headless=true
-DApplicationServerDebug=true
-Ddatasource.verbose=true
-Djdbc.debug=true -Doracle.ias.jcache=true"/>
      </category>
      <category id="stop-parameters">
        <data id="java-options"
value="-Djava.security.policy=$ORACLE_HOME/j2ee/home/config/java2.
policy -Djava.awt.headless=true"/>
      </category>
    </module-data>
    <start timeout="600" retry="2"/>
    <stop timeout="120"/>
    <restart timeout="720" retry="2"/>
    <port id="ajp" range="3301-3400"/>
    <port id="rmi" range="3201-3300"/>
    <port id="jms" range="3701-3800"/>
    <process-set id="default_island" numprocs="1"/>
  </process-type>
</ias-component>

```

The OC4J runtime initializes the Java Object Cache using configuration settings defined in the file `javacache.xml`. The file path is specified in the `<javacache-config>` tag of the `OC4J server.xml` file. The default relative path values of `javacache.xml` in `server.xml` are the following:

```
<javacache-config path="../../../javacache/admin/javacache.xml"/>
```

The rules for writing `javacache.xml` and the default configuration values are specified in an XML schema. The XML schema file `ora-cache.xsd` and the default `javacache.xml` are in the directory `$ORACLE_HOME/javacache/admin` on UNIX and in `%ORACLE_HOME%\javacache\admin` on Windows.

In earlier versions of Java Object Cache (before 9.0.4), configuration was done through the file `javacache.properties`. Starting with version 10g (9.0.4), Java Object Cache configuration is done through `javacache.xml`.

Note: If you install both a release that uses `javacache.properties` (before 9.0.4) and a release that uses `javacache.xml` (9.0.4 or later) on the same host, then you must ensure that the `javacache.xml` `discovery-port` attribute and `javacache.properties` `coordinatorAddress` attribute are not configured to the same port. If they are, then you must manually change the value in one or the other to a different port number. The default range is 7000-7099.

A sample configuration follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<cache-configuration
xmlns=http://www.oracle.com/oracle/ias/cache/configuration
xmlns: xsi="http://www.w3.org/2001/XMLSchema-instance"

```

```

xsi:schemaLocation="http://www.oracle.com/oracle/ias/cache/configuration
ora-cache.xsd">
  <logging>
    <location>javacache.log</location>
    <level>ERROR</level>
  </logging>
  <communication>
    <isDistributed>true</isDistributed>
    <discoverer discovery-port="7000"/>
  </communication>
  <persistence>
    <location>diskcache</location>
    <disksize>32</disksize>
  </persistence>
  <max-objects>1000</max-objects>
  <max-size>48</max-size>
  <clean-interval>30</clean-interval>
</cache-configuration>
    
```

Table 9–5 contains the valid property names and the valid types for each property.

Table 9–5 Java Object Cache Configuration Properties

Configuration XML Element	Description	Type
clean-interval	Specifies the time, in seconds, between each cache cleaning. At the cache-cleaning interval, the Java Object Cache checks for objects that have been invalidated by the <code>TimeToLive</code> or <code>IdleTime</code> attributes that are associated with the object. (Table 9–3 describes these attributes.) Default value: 60	Positive integer
ping-interval	Specifies the time, in seconds, between each cache death detection for determining the availability of the remote cache systems. Default value: 60	Positive integer
max-size	Specifies the maximum size of the memory, in megabytes, available to the Java Object Cache. Default value: 10	Positive integer
max-objects	Specifies the maximum number of in-memory objects that are allowed in the cache. The count does not include group objects, or objects that have been spooled to disk and are not currently in memory. Default value: 5000	Positive integer
region-name-separator	Specifies the separator between a parent region and a child region name. See "Examples" on page 9-24. Default value: /	String
preload-file	Specifies the full path to the declarative cache configuration file. The format of the file must conform to the declarative cache schema (<code>cache.xsd</code>). The declarative cache configuration allows the system to predefine cache regions, groups, objects, attributes, and policies upon Java Object Cache service initialization. For more information about the declarative cache, see "Declarative Cache" on page 9-25. Also see "Examples" on page 9-24. Note: The file path of the declarative cache XML schema is <code>ORACLE_HOME/javacache/admin/cache.xsd</code> . Refer to the XML schema when writing a declarative cache file. Default value: To not use a declarative cache.	String

Table 9–5 (Cont.) Java Object Cache Configuration Properties

Configuration XML Element	Description	Type
communication	<p>Indicates whether the cache is distributed. Specifies the IP address and port that the Java Object Cache initially contacts to join the caching system, when using distributed caching.</p> <p>If the <code>distribute</code> property is set for an object, then updates and invalidation for that object are propagated to other caches known to the Java Object Cache.</p> <p>If the <code>isDistributed</code> subelement of the <code>communication</code> element is set to <code>false</code>, all objects are treated as local, even when the attributes set on objects are set to <code>distribute</code>. See "Examples" on page 9-24.</p> <p>Default value: Cache is not distributed (<code>isDistributed</code> subelement is set to <code>false</code>).</p>	complex (has subelements)
logging	<p>Specifies the logger attributes such as log file name and log level. The available options of the log level are <code>OFF</code>, <code>FATAL</code>, <code>ERROR</code>, <code>DEFAULT</code>, <code>WARNING</code>, <code>TRACE</code>, <code>INFO</code>, and <code>DEBUG</code>. See "Examples" on page 9-24.</p> <p>Default log file name:</p> <p>on UNIX:</p> <pre>\$ORACLE_HOME/javacache/admin/logs/javacache.log</pre> <p>on Windows:</p> <pre>%ORACLE_HOME%\javacache\admin\logs\javacache.log</pre> <p>Default log level: <code>DEFAULT</code></p>	Complex (has subelements)
persistence	<p>Specifies the disk cache configuration, such as absolute path to the disk cache root and maximum size for the disk cache. If a root path is specified, the default maximum size of the disk cache is 10 MB. The unit of the disk cache size is megabytes. See "Examples" on page 9-24.</p> <p>Default value: Disk caching is not available.</p>	Complex (has subelements)

Note: Configuration properties are distinct from the Java Object Cache attributes that you specify using the `Attributes` class.

Examples

The following example illustrates the use of the `<preload-file>` element:

- Specify a declarative cache configuration file:

```
<preload-file>/app/oracle/javacache/admin/decl.xml</preload-file>
```

The following examples illustrate the use of the `<communication>` element:

- Turn off distributed cache:

```
<communication>
  <isDistributed>false</isDistributed>
</communication>
```

- Distribute cache among multiple JVMs in local machine:

```
<communication>
  <isDistributed>>true</isDistributed>
</communication>
```

- Specify the initial discovery port that the Java Object Cache initially contacts to join the caching system in the local node:

```
<communication>
  <isDistributed>true</isDistributed>
  <discoverer discovery-port="7000">
</communication>
```

- Specify the IP address and initial discovery port that the Java Object Cache initially contacts to join the caching system.

```
<communication>
<isDistributed>true</isDistributed>
<discoverer ip="192.10.10.10" discovery-port="7000">
</communication>
```

- Specify multiple IP addresses and the initial discovery port that the Java Object Cache initially contacts to join the caching system. If the first specified address is not reachable, it contacts the next specified address:

```
<communication>
  <isDistributed>true</isDistributed>
  <discoverer ip="192.10.10.10" discovery-port="7000">
  <discoverer ip="192.11.11.11" discovery-port="7000">
  <discoverer ip="192.22.22.22" discovery-port="7000">
  <discoverer ip="192.22.22.22" discovery-port="8000">
</communication>
```

The following examples illustrate the use of the `<persistence>` element:

- Specify a root path for the disk cache using the default disk size:

```
<persistence>
  <location>/app/9iAS/javacache/admin/diskcache</location>
</persistence>
```

- Specify a root path for the disk cache with a disk size of 20 MB:

```
<persistence>
  <location>/app/9iAS/javacache/admin/diskcache</location>
  <disksize>20</disksize>
</persistence>
```

The following examples illustrate the use of the `<logging>` element:

- Specify a log file name:

```
<logging>
<location>/app/9iAS/javacache/admin/logs/my_javacache.log</location>
</logging>
```

- Specify log level as INFO:

```
<logging>
<location>/app/9iAS/javacache/admin/logs/my_javacache.log</location>
<level>INFO</level>
</logging>
```

Declarative Cache

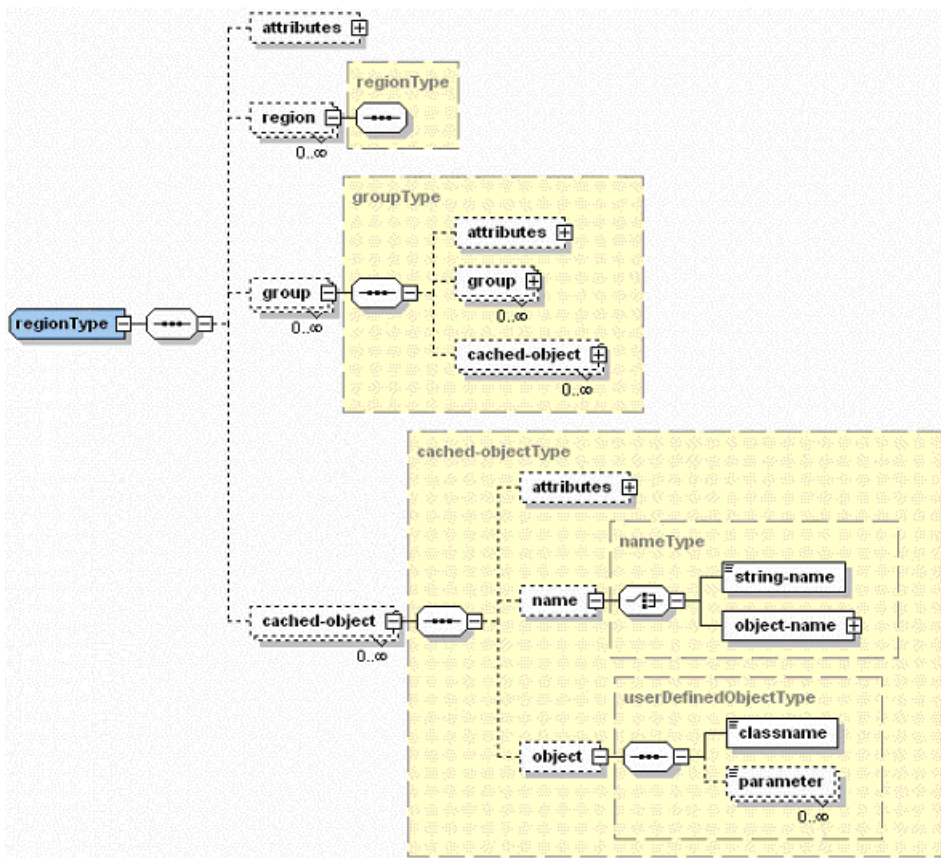
With the 10g Release 2 (10.1.2) release of the Java Object Cache, object, group, and region, as well as cache attributes, can be defined declaratively. You do not need to

write any Java code to define cache objects and attributes in your applications when using declarative cache.

A declarative cache file can be read automatically during Java Object Cache initialization. Specify the location of the declarative cache file in the `<preload-file>` element of the cache configuration file. (See ["Sharing Cached Objects in an OC4J Servlet"](#) on page 9-50 for cache configuration file syntax.) In addition, the declarative cache file can be loaded programmatically or explicitly with the public methods in `oracle.ias.cache.Configurator.class`. Multiple declarative cache files are also permitted.

Figure 9-6 shows the declarative cache.

Figure 9-6 Declarative Cache Architecture



You can set up the Java Object Cache to automatically load a declarative cache file during system initialization. [Example 9-8](#) shows this. [Example 9-9](#) shows how to programmatically read the declarative cache file.

Example 9-8 Automatically Load Declarative Cache

```
<!-- Specify declarative cache file:my_decl.xml in javacache.xml -->
<cache-configuration>
...
<preload-file>/app/9iAS/javacache/admin/my_decl.xml</preload-file>
...
</cache-configuration>
```


Example 9–9 Programmatically Read Declarative Cache File

```
try {
    String filename = "/app/9iAS/javacache/admin/my_decl.xml";
    Configurator config = new Configurator(filename);
    Config.defineDeclarable();
} catch (Exception ex) {
}
```

Declarative Cache File Sample

```
<?xml version="1.0" encoding="UTF-8"?>
<cache xmlns="http://www.javasoft.com/javax/cache"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.oracle.com/javax/cache">
  <region name="fruit">
    <attributes>
      <time-to-live>3000</time-to-live>
      <max-count>200</max-count>
      <capacity-policy>
        <classname>com.acme.MyPolicy</classname>
      </capacity-policy>
    </attributes>
    <group name="apple">
      <attributes>
        <flag>spool</flag>
        <flag>distribute</flag>
        <cache-loader>
          <classname>com.acme.MyLoader</classname>
          <parameter name="color">red</parameter>
        </cache-loader>
      </attributes>
    </group>
    <cached-object>
      <name>
        <string-name>theme</string-name>
      </name>
      <object>
        <classname>com.acme.DialogHandler</classname>
        <parameter name="prompt">Welcome</parameter>
      </object>
    </cached-object>
  </region>
</cache>
```

Declarative Cache File Format

The declarative cache file is in XML format. The file contents should conform to the declarative cache XML schema that is shipped with Oracle Application Server 10g. The file path of the XML schema is *ORACLE_HOME*/javacache/admin/cache.xsd.

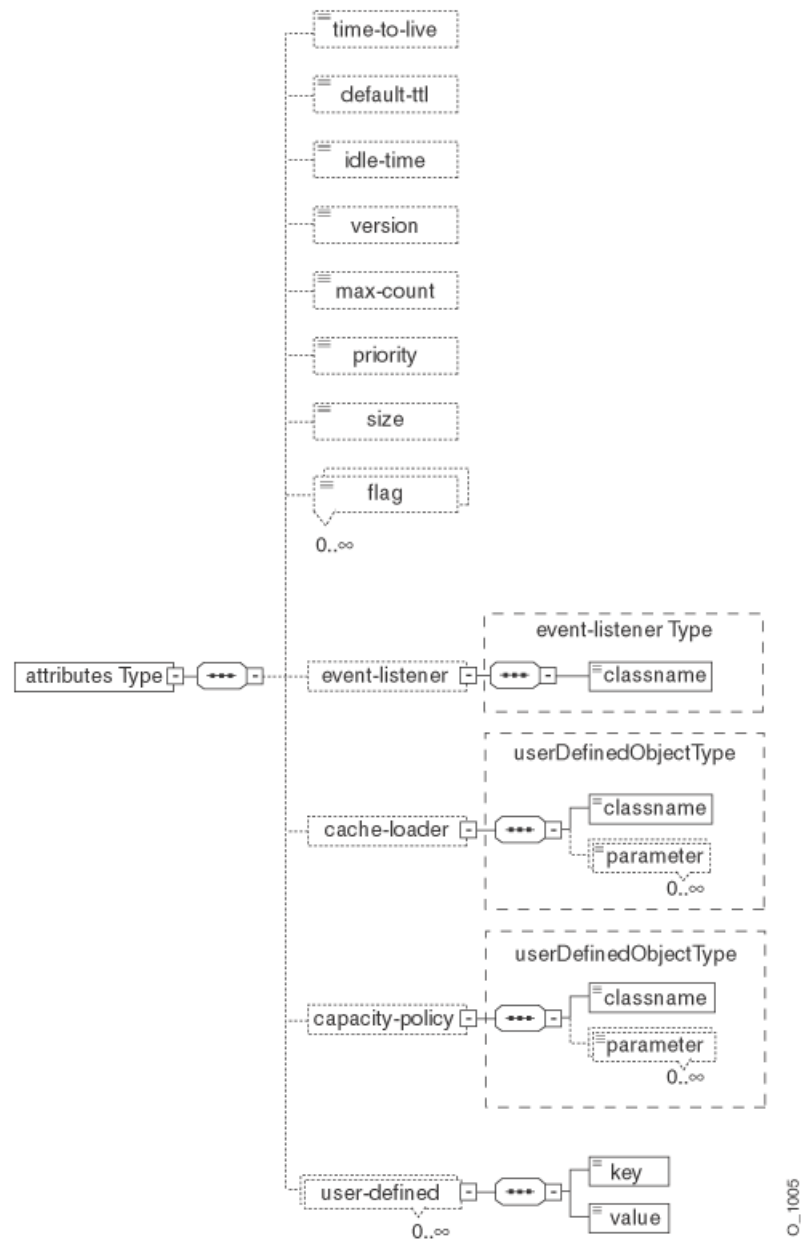
Table 9–6 lists the elements of the declarative cache schema, their children, and the valid types for each element. See **"Examples"** on page 9-29 for code that shows usage for most elements.

Table 9–6 Description of Declarative Cache Schema (*cache.xsd*)

Element	Description	Children	Type
region	Declare a cache region or subregions.	<attributes> <region> <group> <cached-object>	regionType
group	Declare a cache group or subgroup.	<attributes> <group> <cached-object>	groupType
cached-object	Declare a cache object.	<attributes> <name> <object>	objectType
name	Declare the name for a cached object. The name can use a simple string type or it can be a type of a specified Java object.	<string-name> <object-name>	nameType
object	Declare a user-defined Java object. The class of the specified object must implement the declarable interface of the <code>oracle.ias.cache</code> package.	<classname> <parameter>	userDefinedObjectType
attributes	Declare an attributes object for a cache region, group, or cache object. Each child element corresponds to each field in the attributes class of the <code>oracle.ias.cache</code> package. See the Javadoc of <code>Attributes.class</code> for more details.	<time-to-live> <default-ttl> <idle-time> <version> <max-count> <priority> <size> <flag> <event-listener> <cache-loader> <capacity-policy> <user-defined>	attributesType
event-listener	Declare a <code>CacheEventListener</code> object.	<classname>	event-listenerType
cache-loader	Declare a <code>CacheLoader</code> object.	<classname> <parameter>	userDefinedObjectType
capacity-policy	Declare a <code>CapacityPolicy</code> object.	<classname> <parameter>	userDefinedObjectType
user-defined	Declare user-defined string type attributes.	<key> <value>	element

Figure 9–7 shows the attributes of the declarative cache schema.

Figure 9-7 Declarative Cache Schema Attributes



Examples

The following examples show use of elements in Table 9-6:

- Declare cache region and subregions with the <region> element:

```

<region name="themes">
  <region name="cartoon">
    <!-- sub region definition -->
  </region>
  <group name="colors">
    <!-- group definition -->
  </group>
</region>

```

- Declare cache group and subgroups with the `<group>` element:

```
<group name="colors">
  <group name="dark">
    <!-- sub group definition -->
  </group>
</group>
```

- Declare a cached object with the `<cached-object>` element:

```
<cached-object>
  <name>
    <string-name>DialogHandler</string-name>
  </name>
  <object>
    <classname>com.acme.ConfigManager</classname>
    <parameter name="color">blue</parameter>
  </object>
</cached-object>
```

- Declare the name for a cached object with the `<name>` element using a string:

```
<name>
  <string-name>DialogHandler</string-name>
</name>
```

Declare the name for a cached object with the `<name>` element using an object:

```
<name>
  <object-name>
    <classname>DialogHandler</classname>
    <parameter name="color">green</parameter>
  </object-name>
</name>
```

- Declare a user-defined Java object with the `<object>` element:

```
<object>
  <classname>com.acme.CustomConfigManager</classname>
  <parameter name="color">blue</parameter>
</object>
```

```
// Implementation of CustomConfigManager.java
package com.acme;
import oracle.ias.cache.Declarable;
public class CustomConfigManager implements Declarable {
}
```

- Declare an attributes object for a cache region, group, or cache object with the `<attributes>` element:

```
<attributes>
  <time-to-live>4500</time-to-live>
  <default-ttl>6000</default-ttl>
  <version>99</version>
  <max-count>8000</max-count>
  <priority>50</priority>
  <flag>spool</flag>
  <flag>allownull</flag>
  <flag>distribute</flag>
  <flag>reply</flag>
  <cache-loader>
```

```

        <classname>MyLoader</classname>
        <parameter name="debug">false</parameter>
    </cache-loader>
</attributes>

```

- Declare user-defined string type attributes with the `<user-defined>` element:

```

<attributes>
  <user-defined>
    <key>color</key>
    <value>red</value>
  </user-defined>
</attributes>

```

Declarable User-Defined Objects

The topology of the cache objects, object attributes, and user-defined objects can all be described in the declarative cache file. For the system to load and instantiate a user-defined Java object (including `CacheLoader`, `CacheEventListener`, and `CapacityPolicy`) declared in the declarative cache file, the object must be an instance of the `oracle.ias.cache.Declarable` interface. That is, you must implement the `oracle.ias.cache.Declarable` interface for any Java objects declared in the declarative cache file. You must be aware that all user-defined Java objects are loaded by the JVM's default class loader instead of the application's class loaders. After the declarable object is instantiated, the system implicitly invokes its `init(Properties props)` method. The method uses the user-supplied parameters (name-value pair) defined in the declarative cache file to perform any necessary initialization task. [Example 9-10](#) shows how to define an object by declaratively passing in a parameter (color = yellow).

Example 9-10 Define An Object by Declaratively Passing in a Parameter

In the declarative XML file:

```

<cached-object>
  <name>
    <string-name>Foo</string-name>
  </name>
  <object>
    <classname>com.acme.MyCacheObject</classname>
    <parameter name="color">yellow</parameter>
  </object>
</cached-object>

```

Declarable object implementation:

```

package com.acme;

import oracle.ias.cache.*;
import java.util.Properties;

public class MyCacheObject implements Declarable {

    private String color_;

    /**
     * Object initialization
     */
    public void init(Properties prop) {
        color_ = prop.getProperty("color");
    }
}

```

```
}
```

Declarable CacheLoader, CacheEventListener, and CapacityPolicy

When you specify a `CacheLoader`, `CacheEventListener`, or `CapacityPolicy` object in the declarative cache file, the object itself must also be an instance of `oracle.ias.cache.Declarable`. This requirement is similar to that of the user-defined object. You must implement a declarable interface for each specified object in addition to extending the required abstract class. [Example 9-11](#) shows a declarable `CacheLoader` implementation.

Example 9-11 Declarable CacheLoader Implementation

```
import oracle.ias.cache.*;
import java.util.Properties;

public class MyCacheLoader extends CacheLoader implements Declarable {

    public Object load(Object handle, Object argument) {
        // should return meaningful object based on argument
        return null;
    }

    public void init(Properties prop) {
    }
}
```

Initializing the Java Object Cache in a Non-OC4J Container

To use the Java Object Cache in any Java application but run it in a non-OC4J runtime, insert the following reference to where the application (Java class) is initialized:

```
Cache.open(/path-to-ocnfig-file/javacache.xml);
```

If you invoke `Cache.open()` without any parameter in your code, then the Java Object Cache uses its internal default configuration parameter. You can also initialize the Java Object Cache by invoking `Cache.init(CacheAttributes)`. This allows you to derive the configuration parameters from your own configuration file or generate them programmatically.

If the Java Object Cache is not used in the OC4J runtime, you must include `cache.jar` in the classpath where the JVM is launched. You must also initialize the Java Object Cache explicitly by invoking `Cache.open(String config_filename)`, where `config_filename` is the full path to a valid `javacache.xml` file, or by invoking `Cache.init(CacheAttributes)`.

Use any of the following method invocations to initialize the Java Object Cache explicitly in a non-OC4J container:

- `Cache.open();`
Use the default Java Object Cache configuration stored in the `cache.jar` file.
- `Cache.open(/path-to-oracle-home/javacache/admin/javacache.xml);`

Use the configuration defined in the `javacache.xml` file.
- `Cache.open(/path-to-user's-own-javacache.xml);`
Use the configuration defined in the specific `javacache.xml` file.

- `Cache.init(CacheAttributes);`

Use the configuration that is set in a `CacheAttributes` object.

For J2EE applications running in an OC4J container, the path to the `javacache.xml` file can be configured in the OC4J `server.xml` configuration file. The cache can be initialized automatically when the OC4J process is started. See OC4J configuration for details.

In a non-OC4J container, if you do not use any of the preceding method invocations, the Java Object Cache is initialized implicitly (using default configuration settings stored in `cache.jar`) when you invoke `Cache.getAccess()` or `Cache.defineRegion()`.

Capacity Control

The new capacity control feature allows the cache user to specify the policy to employ when determining which objects should be removed from the cache when the capacity of the cache, region, or group has been reached. To specify the policy, extend the abstract class `CapacityPolicy`, and set the instantiated object as an attribute of the cache, region, or group.

For regions and groups, the `CapacityPolicy` object is called when the region or group has reached its capacity and a new object is being loaded. An object in the region or group must be found to invalidate, or the new object is not saved in the cache. (It is returned to the user but is immediately invalidated.)

The `CapacityPolicy` object that is associated with the cache as a whole is called when capacity of the cache reaches some "high water mark," some percentage of the configured maximum. When the high water mark is reached, the cache attempts to remove objects to reduce the load in the cache to 3% below the high water mark. The high water mark is specified by the `capacityBuffer` cache attribute. If the `capacityBuffer` is set to 5, then the cache begins removing objects from the cache when it is 95% full (100% - 5%) and continues until the cache is 92% full (95% - 3%). The default value for `capacityBuffer` is 15.

The capacity policy used for the cache can be different from those used for specific regions or groups.

By default, the capacity policy for groups and regions is to remove a nonreferenced object of equal or lesser priority when a new object is added and capacity has been reached. For the cache, the default policy is to remove objects that have not been referenced in the last two clean intervals, with preference to objects of priority—that is, low priority objects that have not been referenced recently are removed first.

To help create a capacity policy, many statistics are kept for objects in the cache and aggregated across the cache, regions, and groups. The statistics are available to the `CapacityPolicy` object. For cache objects, the following statistics are maintained:

- Priority
- Access count—the number of times the object has been referenced
- Size—the size of the object in bytes (if available)
- Last access time—the time in milliseconds that the object was last accessed
- Create time—the time in milliseconds when the object was created
- Load time—the number of milliseconds required to load the object (if the object was added to the cache with `CacheAccess.put`, this value is 0)

Along with these statistics, all attributes associated with the object are available to the `CapacityPolicy` object.

The following aggregated statistics are maintained for the cache, regions, and groups. For each of these statistics, the low, high, and average value is maintained. These statistics are recalculated at each clean interval or when `Cache.updateStats()` is called.

- Priority
- Access count—the number of times that the object has been referenced
- Size—the size of the object in bytes (if available)
- Last access time—the time in milliseconds that the object was last accessed
- Load time—the number of milliseconds required to load the object (if the object was added to the cache with `CacheAccess.put`, this value is 0)

[Example 9–12](#) is a sample `CapacityPolicy` object for a region, based on object size.

Example 9–12 Sample CapacityPolicy Based on Object Size

```
class SizePolicy extends CapacityPolicy
{
    public boolean policy (Object victimHandle, AggregateStatus aggStatus,
        long currentTime , Object newObjectHandle) throws CacheException
    {
        int          newSize;
        int          oldSize;

        oldSize = getAttributes(victimHandle).getSize();
        newSize = getAttributes(newObjectHandle).getSize();
        if (newSize >= oldSize)
            return true;
        return false;
    }
}
```

[Example 9–13](#) is a sample `CapacityPolicy` for the cache, based on access time and reference count. If an object has below-average references and has not been accessed in the last 30 seconds, then it is removed from the cache.

Example 9–13 Sample CapacityPolicy Based on Access Time and Reference Count

```
class SizePolicy extends CapacityPolicy
{
    public boolean policy (Object victimHandle, AggregateStatus aggStatus, long
        currentTime , Object newObjectHandle) throws CacheException
    {
        long          lastAccess;
        int           accessCount;
        int           avgAccCnt;

        lastAccess    = getStatus(victimHandle).getLastAccess();
        accessCount    = getStatus(victimHandle).getAccessCount();
        avgAccCnt      = aggStatus.getAccessCount(AggregateStatus.AVG);

        if (lastAccess + 30000 < currentTime && accessCount < avgAccCnt)
            return true;
    }
}
```


Implementing a Cache Event Listener

Many events can occur in the life cycle of a cached object, including object creation and object invalidation. This section shows how an application can be notified when cache events occur.

To receive notification of the creation of an object, implement event notification as part of the `cacheLoader`. For notification of invalidation or updates, implement a `CacheEventListener`, and associate the `CacheEventListener` with an object, group, region, or subregion using `Attributes.setCacheEventListener()`.

`CacheEventListener` is an interface that extends `java.util.EventListener`. The cache event listener provides a mechanism to establish a callback method that is registered and then executes when the event occurs. In the Java Object Cache, the event listener executes when a cached object is invalidated or updated.

An event listener is associated with a cached object, group, region, or subregion. If an event listener is associated with a group, region, or subregion, then by default, the listener runs only when the group, region, or subregion itself is invalidated.

Invalidating a member does not trigger the event. The

`Attributes.setCacheEventListener()` call takes a boolean argument that, if `true`, applies the event listener to each member of the region, subregion, or group, rather than to the region, subregion, or group itself. In this case, the invalidation of an object within the region, subregion, or group triggers the event.

The `CacheEventListener` interface has one method, `handleEvent()`. This method takes a single argument, a `CacheEvent` object that extends `java.util.EventObject`. This object has two methods, `getID()`, which returns the type of event (`OBJECT_INVALIDATION` or `OBJECT_UPDATED`), and `getSource()`, which returns the object being invalidated. For groups and regions, the `getSource()` method returns the name of the group or region.

The `handleEvent()` method is executed in the context of a background thread that the Java Object Cache manages. Avoid using Java Native Interface (JNI) code in this method, because the expected thread context may not be available.

[Example 9–14](#) illustrates how a `CacheEventListener` is implemented and associated with an object or a group.

Example 9–14 *Implementing a CacheEventListener*

```
import oracle.ias.cache.*;
// A CacheEventListener for a cache object
class MyEventListener implements
CacheEventListener {

    public void handleEvent(CacheEvent ev)
    {
        MyObject obj = (MyObject)ev.getSource();
        obj.cleanup();
    }

    // A CacheEventListener for a group object
    class MyGroupEventListener implements CacheEventListener {
    public void handleEvent(CacheEvent ev)
    {
        String groupName = (String)ev.getSource();
        app.notify("group " + groupName + " has been invalidated");
    }
    }
}
```

Use the `Attributes.listener` attribute to specify the `CacheEventListener` for a region, subregion, group, or object.

[Example 9-15](#) illustrates how to set a cache event listener on an object. [Example 9-16](#) illustrates how to set a cache event listener on a group.

Example 9-15 Setting a Cache Event Listener on an Object

```
import oracle.ias.cache.*;

class YourObjectLoader extends CacheLoader
{
    public YourObjectLoader () {
    }

    public Object load(Object handle, Object args) {
        Object obj = null;
        Attributes attr = new Attributes();
        MyEventListener el = new MyEventListener();
        attr.setCacheEventListener(CacheEvent.OBJECT_INVALIDATED, el);

        // your implementation to retrieve or create your object

        setAttributes(handle, attr);
        return obj;
    }
}
```

Example 9-16 Setting a Cache Event Listener on a Group

```
import oracle.ias.cache.*;
try
{
    CacheAccess cacc = CacheAccess.getAccess(myRegion);
    Attributes attr = new Attributes ();

    MyGroupEventListener listener = new MyGroupEventListener();
    attr.setCacheEventListener(CacheEvent.OBJECT_INVALIDATED, listener);

    cacc.defineGroup("myGroup", attr);
    //....
    cacc.close();

}catch(CacheException ex)
{
    // handle exception
}
```

Restrictions and Programming Pointers

This section covers restrictions and programming pointers when using the Java Object Cache.

- Do not share the `CacheAccess` object between threads. This object represents a user to the caching system. The `CacheAccess` object contains the current state of the user's access to the cache: what object is currently being accessed, what objects are currently owned, and so on. Trying to share the `CacheAccess` object is unnecessary and may result in unpredictable behavior.

- A `CacheAccess` object holds a reference to only one cached object at a time. If multiple cached objects are being accessed concurrently, then use multiple `CacheAccess` objects. For objects that are stored in memory, the consequences of not doing this are minor, because Java prevents the cached object from being garbage collected, even if the cache believes it is not being referenced. For disk objects, if the cache reference is not maintained, the underlying file could be removed by another user or by time-based invalidation, causing unexpected exceptions. To optimize resource management, keep the cache reference open as long as the cached object is being used.
- Always close a `CacheAccess` object when it is no longer being used. The `CacheAccess` objects are pooled. They acquire cache resources on behalf of the user. If the access object is not closed when it is not being used, then these resources are not returned to the pool and are not cleaned up until they are garbage collected by the JVM. If `CacheAccess` objects are continually allocated and not closed, then degradation in performance may occur.
- When local objects (objects that do not set the `Attributes.DISTRIBUTE` attribute) are saved to disk using the `CacheAccess.save()` method, they do not survive the termination of the process. By definition, local objects are visible only to the cache instance where they were loaded. If that cache instance goes away for any reason, then the objects that it manages, including on disk, are lost. If an object must survive process termination, then both the object and the cache must be defined `DISTRIBUTE`.
- The cache configuration, also called the cache environment, is local to a cache; this includes the region, subregion, group, and object definitions. The cache configuration is not saved to disk or propagated to other caches. Define the cache configuration during the initialization of the application.
- If a `CacheAccess.waitForResponse()` or `CacheAccess.releaseOwnership()` method call times out, then you must call it again until it returns successfully. If `CacheAccess.waitForResponse()` does not succeed, then you must call `CacheAccess.cancelResponse` to free resources. If `CacheAccess.releaseOwnership()` doesn't succeed, then you must call `CacheAccess.releaseOwnership` with a timeout value of -1 to free resources.
- When a group or region is destroyed or invalidated, distributed definitions take precedence over local definitions. That is, if the group is distributed, then all objects in the group or region are invalidated or destroyed across the entire cache system, even if the individual objects or associated groups are defined as local. If the group or region is defined as local, then local objects within the group are invalidated locally; distributed objects are invalidated throughout the entire cache system.
- When an object or group is defined with the `SYNCHRONIZE` attribute set, ownership is required to load or replace the object. However, ownership is not required for general access to the object or to invalidate the object.
- In general, objects that are stored in the cache should be loaded by the system class loader that is defined in the `classpath` when the JVM is initialized, rather than by a user-defined class loader. Specifically, any objects that are shared between applications or can be saved or spooled to disk must be defined in the system `classpath`. Failure to do so can result in a `ClassNotFoundException` or a `ClassCastException`.
- On some systems, the open file descriptors can be limited by default. On these systems, you may need to change system parameters to improve performance. On

UNIX systems, for example, a value of 1024 or greater can be an appropriate value for the number of open file descriptors.

- When configured in either local or distributed mode, at startup, one active Java Object Cache cache is created in a JVM process (that is, in the program running in the JVM that uses the Java Object Cache API).

Working with Disk Objects

The Java Object Cache can manage objects on disk as well as in memory.

This section covers the following topics:

- [Local and Distributed Disk Cache Objects](#)
- [Adding Objects to the Disk Cache](#)

Local and Distributed Disk Cache Objects

This section covers the following topics:

- [Local Objects](#)
- [Distributed Objects](#)

Local Objects

When operating in local mode, the cache attribute `isDistributed` is not set and all objects are treated as local objects (even when the `DISTRIBUTE` attribute is set for an object). In local mode, all objects in the disk cache are visible only to the Java Object Cache cache that loaded them, and they do not survive after process termination. In local mode, objects stored in the disk cache are lost when the process using the cache terminates.

Distributed Objects

If the cache attribute `isDistributed` is set to true, the cache will operate in distributed mode. Disk cache objects can be shared by all caches that have access to the file system hosting the disk cache. This is determined by the disk cache location configured. This configuration allows for better utilization of disk resources and allows disk objects to persist beyond the life of the Java Object Cache process.

Objects that are stored in the disk cache are identified using the concatenation of the path that is specified in the `diskPath` configuration property and an internally generated `String` representing the remaining path to the file. Thus, caches that share a disk cache can have a different directory structure, as long as the `diskPath` represents the same directory on the physical disk and is accessible to the Java Object Cache processes.

If a memory object that is saved to disk is also distributed, the memory object can survive the death of the process that spooled it.

Adding Objects to the Disk Cache

There are several ways to use the disk cache with the Java Object Cache, including:

- [Automatically Adding Objects](#)
- [Explicitly Adding Objects](#)
- [Using Objects that Reside Only in Disk Cache](#)

Automatically Adding Objects

The Java Object Cache automatically adds certain objects to the disk cache. Such objects can reside either in the memory cache or in the disk cache. If an object in the disk cache is needed, it is copied back to the memory cache. The action of spooling to disk occurs when the Java Object Cache determines that it requires free space in the memory cache. Spooling of an object occurs only if the `SPOOL` attribute is set for the object.

Explicitly Adding Objects

In some situations, you may want to force one or more objects to be written to the Java Object Cache disk cache. Using the `CacheAccess.save()` method, a region, subregion, group, or object is written to the disk cache. (If the object or objects are already in the disk cache, they are not written again.)

Note: Using `CacheAccess.save()` saves an object to disk even when the `SPOOL` attribute is not set for the object.

Calling `CacheAccess.save()` on a region, subregion, or group saves all the objects within the region, subregion, or group to the disk cache. During a `CacheAccess.save()` method call, if an object is encountered that cannot be written to disk, either because it is not serializable or for other reasons, then the event is recorded in the Java Object Cache log, and the save operation continues with the next object. When individual objects are written to disk, the write is synchronous. If a group or region is saved, then the write is performed as an asynchronous background task.

Using Objects that Reside Only in Disk Cache

Objects that you access only directly from disk cache are loaded into the disk cache by calling `CacheLoader.createDiskObject()` from the `CacheLoader.load()` method. The `createDiskObject()` method returns a `File` object that the application can use to load the disk object. If the attributes of the disk object are not defined for the disk object, then set them using the `createDiskObject()` method. The system manages local and distributed disk objects differently; the system determines if the object is local or distributed when it creates the object, based on the specified attributes.

Note: If you want to share a disk cache object between distributed caches in the same cache system, then you must define the `DISTRIBUTE` attribute when the disk cache object is created. This attribute cannot be changed after the object is created.

When `CacheAccess.get()` is called on a disk object, the full path name to the file is returned, and the application can open the file, as needed.

Disk objects are stored on a local disk and accessed directly from the disk by the application using the Java Object Cache. Disk objects can be shared by all Java Object Cache processes, or they can be local to a particular process, depending on the setting for the `DISTRIBUTE` attribute (and the mode that the Java Object Cache is running in, either distributed or local).

[Example 9-17](#) shows a loader object that loads a disk object into the cache.

Example 9–17 Creating a Disk Object in a CacheLoader

```
import oracle.ias.cache.*;

class YourObjectLoader extends CacheLoader
{
    public Object load(Object handle, Object args) {
        File file;
        FileOutputStream = out;
        Attributes attr = new Attributes();

        attr.setFlags(Attributes.DISTRIBUTE);
        try
        // The distribute attribute must be set on the createDiskObject method
        {
            file = createDiskObject(handle, attr);
            out = new FileOutputStream(file);

            out.write((byte[])getInfofromsomewhere());
            out.close();
        }
        catch (Exception ex) {
            // translate exception to CacheException, and log exception
            throw exceptionHandler("exception in file handling", ex)
        }
        return file;
    }
}
```

[Example 9–18](#) illustrates application code that uses a Java Object Cache disk object. This example assumes that the region named `Stock-Market` is already defined with the `YourObjectLoader` loader that was set up in [Example 9–17](#) as the default loader for the region.

Example 9–18 Application Code that Uses a Disk Object

```
import oracle.ias.cache.*;

try
{
    FileInputStream in;
    File file;
    String filePath;
    CacheAccess cacc = CacheAccess.getAccess("Stock-Market");

    filePath = (String)cacc.get("file object");
    file = new File(filePath);
    in = new FileInputStream(filePath);
    in.read(buf);

    // do something interesting with the data
    in.close();
    cacc.close();
}
catch (Exception ex)
{
    // handle exception
}
```

Working with StreamAccess Objects

A `StreamAccess` object is accessed as a stream and automatically loaded to the disk cache. The object is loaded as an `OutputStream` and read as an `InputStream`. Smaller `StreamAccess` objects can be accessed from memory or from the disk cache; larger `StreamAccess` objects are streamed directly from the disk. The Java Object Cache automatically determines where to access the `StreamAccess` object, based on the size of the object and the capacity of the cache.

The user is always presented with a stream object, an `InputStream` for reading and an `OutputStream` for writing, regardless of whether the object is in a file or in memory. The `StreamAccess` object allows the Java Object Cache user to always access the object in a uniform manner, without regard to object size or resource availability.

Creating a StreamAccess Object

To create a `StreamAccess` object, call the `CacheLoader.createStream()` method from the `CacheLoader.load()` method when the object is loaded into the cache. The `createStream()` method returns an `OutputStream` object. Use the `OutputStream` object to load the object into the cache.

If the attributes have not already been defined for the object, then set them using the `createStream()` method. The system manages local and distributed disk objects differently; the determination of local or distributed is made when the system creates the object, based on the attributes.

Note: If you want to share a `StreamAccess` object between distributed caches in the same cache system, then you must define the `DISTRIBUTE` attribute when the `StreamAccess` object is created. You cannot change this attribute after the object is created.

[Example 9–19](#) shows a loader object that loads a `StreamAccess` object into the cache.

Example 9–19 *Creating a StreamAccess Object in a Cache Loader*

```
import oracle.ias.cache.*;

class YourObjectLoader extends CacheLoader
{
    public Object load(Object handle, Object args) {
        OutputStream out;
        Attributes attr = new Attributes();
        attr.setFlags(Attributes.DISTRIBUTE);

        try
        {
            out = createStream(handle, attr);
            out.write((byte[])getInfofromsomewhere());
        }
        catch (Exception ex) {
            // translate exception to CacheException, and log exception
            throw exceptionHandler("exception in write", ex)
        }
        return out;
    }
}
```

Working with Pool Objects

A pool object is a special cache object that the Java Object Cache manages. A pool object contains a set of identical object instances. The pool object itself is a shared object; the objects within the pool are private objects that the Java Object Cache manages. Users access individual objects within the pool with a check out, using a pool access object, and then return the objects to the pool when they are no longer needed.

This section covers the following topics:

- [Creating Pool Objects](#)
- [Using Objects from a Pool](#)
- [Implementing a Pool Object Instance Factory](#)
- [Pool Object Affinity](#)

Creating Pool Objects

To create a pool object, use `CacheAccess.createPool()`. The `CreatePool()` method takes as arguments:

- A `PoolInstanceFactory`
- An `Attributes` object
- Two integer arguments

The integer arguments specify the maximum pool size and the minimum pool size. By supplying a group name as an argument to `CreatePool()`, a pool object is associated with a group.

Attributes, including `TimeToLive` or `IdleTime`, can be associated with a pool object. These attributes can be applied to the pool object itself, when specified in the attributes set with `CacheAccess.createPool()`, or they can be applied to the objects within the pool individually.

Using `CacheAccess.createPool()`, specify minimum and maximum sizes with the integer arguments. Specify the minimum first. It sets the minimum number of objects to create within the pool. The minimum size is interpreted as a request rather than a guaranteed minimum. Objects within a pool object are subject to removal from the cache due to lack of resources, so the pool can decrease the number of objects below the requested minimum value. The maximum pool size puts a hard limit on the number of objects available in the pool.

Note: Pool objects and the objects within a pool object are always treated as local objects.

[Example 9–20](#) shows how to create a pool object.

Example 9–20 *Creating a Pool Object*

```
import oracle.ias.cache.*;

try
{
    CacheAccess cacc = CacheAccess.getAccess("Stock-Market");
    Attributes attr = new Attributes();
    QuoteFactory poolFac = new QuoteFactory();
```



```

        // set IdleTime for an object in the pool to three minutes
        attr.setIdleTime(180);
        // create a pool in the "Stock-Market" region with a minimum of
        // 5 and a maximum of 10 object instances in the pool
        cacc.createPool("get Quote", poolFac, attr, 5, 10);
        cacc.close();
    }
    catch(CacheException ex)
    {
        // handle exception
    }
}

```

Using Objects from a Pool

To access objects in a pool, use a `PoolAccess` object. The `PoolAccess.getPool()` static method returns a handle to a specified pool. The `PoolAccess.get()` method returns an instance of an object from within the pool (this checks out an object from the pool). When an object is no longer needed, return it to the pool, using the `PoolAccess.returnToPool()` method, which checks the object back into the pool. Finally, call the `PoolAccess.close()` method when the pool handle is no longer needed.

[Example 9–21](#) describes the calls that are required to create a `PoolAccess` object, check an object out of the pool, and then check the object back in and close the `PoolAccess` object.

Example 9–21 Using a PoolAccess Object

```

PoolAccess pacc = PoolAccess.getPool("Stock-Market", "get Quote");
//get an object from the pool
GetQuote gq = (GetQuote)pacc.get();
// do something useful with the gq object
// return the object to the pool
pacc.returnToPool(gq);
pacc.close();

```

Implementing a Pool Object Instance Factory

The Java Object Cache instantiates and removes objects within a pool, using an application-defined factory object—a `PoolInstanceFactory`. The `PoolInstanceFactory` is an abstract class with two methods that you must implement: `createInstance()` and `destroyInstance()`.

The Java Object Cache calls `createInstance()` to create instances of objects that are being accumulated within the pool. The Java Object Cache calls `destroyInstance()` when an instance of an object is being removed from the pool. (Object instances from within the pool are passed into `destroyInstance()`.)

The size of a pool object (that is, the number of objects within the pool) is managed using these `PoolInstanceFactory()` methods. The system decreases or increases the size and number of objects in the pool, based on demand, and based on the values of the `TimeToLive` or `IdleTime` attributes.

[Example 9–22](#) shows the calls required when implementing a `PoolInstanceFactory`.

Example 9–22 Implementing Pool Instance Factory Methods

```
import oracle.ias.cache.*;
public class MyPoolFactory implements PoolInstanceFactory
{
    public Object createInstance()
    {
        MyObject obj = new MyObject();
        obj.init();
        return obj;
    }
    public void destroyInstance(Object obj)
    {
        ((MyObject)obj).cleanup();
    }
}
```

Pool Object Affinity

Object pools are a collection of serially reusable objects. A user "checks out" an object from the pool to perform a function, then "checks in" the object back to the pool when done. During the time the object is checked out, the user has exclusive use of that object instance. After the object is checked in, the user gives up all access to the object. While the object is checked out, the user can apply temporary modifications to the pool object (add state) to allow it to execute the current task. Since some cost is incurred to add these modifications, it would be beneficial to allow the user to, whenever possible, get the same object from the pool with the modifications already in place. Since the 9.0.2 version of the Java Object Cache, the only way to do this was never to check in the object, which would then defeat the purpose of the pool. To support the pool requirement described in this paragraph, the functionality described in the following two paragraphs has been added to the pool management of the Java Object Cache.

Objects checked into the pool using the `returnToPool` method of the `PoolAccess` object maintain an association with the last `PoolAccess` object that referenced the object. When the `PoolAccess` handle requests an object instance, the same object it had previously is returned. This association will be terminated if the `PoolAccess` handle is closed, or the `PoolAccess.release` method is called, or the object is given to another user. Before the object is given to another user, a callback is made to determine if the user is willing to give up the association with the object. If the user is not willing to dissolve the association, then the new user is not given access to the object. The interface `PoolAffinityFactory` extends the interface `PoolInstanceFactory`, adding the callback method `affinityRelease`. This method returns `true` if the association can be broken, and `false` otherwise.

If the entire pool is invalidated, the `affinityRelease` method is not called. Object instance cleanup is then performed with the `PoolInstanceFactory.instanceDestroy` method.

Running in Local Mode

When running in local mode, the Java Object Cache does not share objects or communicate with any other caches running locally on the same system or remotely across the network. Object persistence across system shutdowns or program failures is not supported when running in local mode.

By default, the Java Object Cache runs in local mode, and all objects in the cache are treated as local objects. When the Java Object Cache is configured in local mode, the cache ignores the `DISTRIBUTE` attribute for all objects.

Running in Distributed Mode

In distributed mode, the Java Object Cache can share objects and communicate with other caches running either locally on the same system or remotely across the network. Object updates and invalidations are propagated between communicating caches. Distributed mode supports object persistence across system shutdowns and program failures.

This section covers the following topics:

- [Configuring Properties for Distributed Mode](#)
- [Using Distributed Objects, Regions, Subregions, and Groups](#)
- [Cached Object Consistency Levels](#)
- [Sharing Cached Objects in an OC4J Servlet](#)

Configuring Properties for Distributed Mode

To configure the Java Object Cache to run in distributed mode, set the value of the `distribute` and `discoveryAddress` configuration properties in the `javacache.xml` file.

Setting the `distribute` Configuration Property

To start the Java Object Cache in distributed mode, set the `isDistributed` attribute to `true` in the configuration file. "[Java Object Cache Configuration](#)" on page 9-21 describes how to do this.

Setting the `discoveryAddress` Configuration Property

In distributed mode, invalidations, destroys, and replaces are propagated through the messaging system of the cache. The messaging system requires a known host name and port address to allow a cache to join the cache system when it is first initialized. Use the `discoverer` attribute in the communication section in the `javacache.xml` file to specify a list of host name and port addresses.

By default, the Java Object Cache sets the `discoverer` to the value `:12345` (this is equivalent to `localhost:12345`). To eliminate conflicts with other software on the site, have your system administrator set the `discoveryAddress`.

If the Java Object Cache spans systems, then configure multiple `discoverer` entries, with one `hostname:port` pair specified for each node. Doing this avoids any dependency on a particular system being available or on the order the processes are started. Also see "[Java Object Cache Configuration](#)" on page 9-21.

Note: All caches cooperating in the same cache system must specify the same set of host name and port addresses. The address list, set with the `discoverer` attributes, defines the caches that make up a particular cache system. If the address lists vary, then the cache system could be partitioned into distinct groups, resulting in inconsistencies between caches.

Using Distributed Objects, Regions, Subregions, and Groups

When the Java Object Cache runs in distributed mode, individual regions, subregions, groups, and objects can be either local or distributed. By default, objects, regions, subregions, and groups are defined as local. To change the default local value, set the `DISTRIBUTE` attribute when the object, region, or group is defined.

A distributed cache can contain both local and distributed objects.

Several attributes and methods in the Java Object Cache allow you to work with distributed objects and control the level of consistency of object data across the caches. Also see "[Cached Object Consistency Levels](#)" on page 9-49.

Using the REPLY Attribute with Distributed Objects

When updating, invalidating, or destroying objects across multiple caches, it may be useful to know when the action has completed at all the participating sites. Setting the `REPLY` attribute causes all participating caches to send a reply to the originator when a requested action has completed for the object. The `CacheAccess.waitForResponse()` method allows the user to block until all remote operations have completed.

To wait for a distributed action to complete across multiple caches, use `CacheAccess.waitForResponse()`. To ignore responses, use the `CacheAccess.cancelResponse()` method, which frees the cache resources used to collect the responses.

Both `CacheAccess.waitForResponse()` and `CacheAccess.cancelResponse()` apply to all objects that are accessed by the `CacheAccess` object. This feature allows the application to update several objects, then wait for all the replies.

[Example 9-23](#) illustrates how to set an object as distributed and handle replies when the `REPLY` attribute is set. In this example, you can also set the attributes for the entire region. Additionally, you can set attributes for a group or individual object, as appropriate for your application.

Example 9-23 *Distributed Caching Using Reply*

```
import oracle.ias.cache.*;

CacheAccess cacc;
String      obj;
Attributes attr = new Attributes ();
MyLoader   loader = new MyLoader();

// mark the object for distribution and have a reply generated
// by the remote caches when the change is completed

attr.setFlags(Attributes.DISTRIBUTE|Attributes.REPLY);
attr.setLoader(loader);

CacheAccess.defineRegion("testRegion",attr);
cacc = CacheAccess.getAccess("testRegion"); // create region with
//distributed attributes

obj = (String)cacc.get("testObject");
cacc.replace("testObject", obj + "new version"); // change will be
// propagated to other caches

cacc.invalidate("invalidObject"); // invalidation is propagated to other caches
```

```

try
{
// wait for up to a second,1000 milliseconds, for both the update
// and the invalidate to complete
    cacc.waitForResponse(1000);

catch (TimeoutException ex)
{
    // tired of waiting so cancel the response
    cacc.cancelResponse();
}
cacc.close();
}

```

Using SYNCHRONIZE and SYNCHRONIZE_DEFAULT

When updating objects across multiple caches, or when multiple threads access a single object, you can coordinate the update action. Setting the `SYNCHRONIZE` attribute enables synchronized updates, and requires an application to obtain ownership of an object before the object is loaded or updated.

The `SYNCHRONIZE` attribute also applies to regions, subregions, and groups. When the `SYNCHRONIZE` attribute is applied to a region, subregion, or group, ownership of the region, subregion, or group must be obtained before an object can be loaded or replaced in the region, subregion, or group.

Setting the `SYNCHRONIZE_DEFAULT` attribute on a region, subregion, or group applies the `SYNCHRONIZE` attribute to all the objects within the region, subregion, or group. Ownership must be obtained for the individual objects within the region, subregion, or group before they can be loaded or replaced.

To obtain ownership of an object, use `CacheAccess.getOwnership()`. After ownership is obtained, no other `CacheAccess` instance is allowed to load or replace the object. Reads and invalidation of objects are not affected by synchronization.

After ownership has been obtained and the modification to the object is completed, call `CacheAccess.releaseOwnership()` to release the object. `CacheAccess.releaseOwnership()` waits up to the specified time for the updates to complete at the remote caches. If the updates complete within the specified time, ownership is released; otherwise, a `TimeoutException` is thrown. If the method times out, call `CacheAccess.releaseOwnership()` again. `CacheAccess.releaseOwnership()` must return successfully for ownership to be released. If the timeout value is `-1`, then ownership is released immediately, without waiting for the responses from the other caches.

[Example 9-24](#) illustrates distributed caching using `SYNCHRONIZE` and `SYNCHRONIZE_DEFAULT`.

Example 9-24 Distributed Caching Using `SYNCHRONIZE` and `SYNCHRONIZE_DEFAULT`

```

import oracle.ias.cache.*;

CacheAccess cacc;
String      obj;
Attributes attr = new Attributes ();
MyLoader   loader = new MyLoader ();

// mark the object for distribution and set synchronize attribute
attr.setFlags(Attributes.DISTRIBUTE|Attributes.SYNCHRONIZE);
attr.setLoader(loader);

```

```
//create region
CacheAccess.defineRegion("testRegion");
cacc = CacheAccess.getAccess("testRegion");
cacc.defineGroup("syncGroup", attr); //define a distributed synchronized group
cacc.defineObject("syncObject", attr); // define a distributed synchronized object
attr.setFlagsToDefaults() // reset attribute flags

// define a group where SYNCHRONIZE is the default for all objects in the group
attr.setFlags(Attributes.DISTRIBUTE|Attributes.SYNCHRONIZE_DEFAULT);
cacc.defineGroup("syncGroup2", attr);
try
{
// try to get the ownership for the group don't wait more than 5 seconds
cacc.getOwnership("syncGroup", 5000);
obj = (String)cacc.get("testObject", "syncGroup"); // get latest object
// replace the object with a new version
cacc.replace("testObject", "syncGroup", obj + "new version");
obj = (String)cacc.get("testObject2", "syncGroup"); // get a second object
// replace the object with a new version
cacc.replace("testObject2", "syncGroup", obj + "new version");
}

catch (TimeoutException ex)
{
System.out.println("unable to acquire ownership for group");
cacc.close();
return;
}
try
{
cacc.releaseOwnership("syncGroup",5000);
}
catch (TimeoutException ex)
{
// tired of waiting so just release ownership
cacc.releaseOwnership("syncGroup", -1);
}
try
{
cacc.getOwnership("syncObject", 5000); // try to get the ownership for the object
// don't wait more than 5 seconds
obj = (String)cacc.get("syncObject"); // get latest object
cacc.replace("syncObject", obj + "new version"); // replace the object with a new version
}
catch (TimeoutException ex)
{
System.out.println("unable to acquire ownership for object");
cacc.close();
return;
}
try
{
cacc.releaseOwnership("syncObject", 5000);
}
catch (TimeoutException ex)
{
cacc.releaseOwnership("syncObject", -1); // tired of waiting so just release ownership
}
try
```

```

{
    cacc.getOwnership("Object2", "syncGroup2", 5000); // try to get the ownership for the object
    // where the ownership is defined as the default for the group don't wait more than 5 seconds
    obj = (String)cacc.get("Object2", "syncGroup2"); // get latest object
    // replace the object with new version
    cacc.replace("Object2", "syncGroup2", obj + "new version");
}

catch (TimeoutException ex)
{
    System.out.println("unable to acquire ownership for object");
    cacc.close();
    return;
}
try
{
    cacc.releaseOwnership("Object2", 5000);
}
catch (TimeoutException ex)
{
    cacc.releaseOwnership("Object2", -1); // tired of waiting so just release ownership
}
cacc.close();
}

```

Cached Object Consistency Levels

Within the Java Object Cache, each cache manages its own objects locally, within its JVM process. In distributed mode, when using multiple processes or when the system is running on multiple sites, a copy of an object can exist in more than one cache.

The Java Object Cache allows you to specify the consistency level that is required between copies of objects that are available in multiple caches. The consistency level that you specify depends on the application and the objects being cached. The supported levels of consistency vary, from none to all copies of objects being consistent across all communicating caches.

Setting object attributes specifies the level of consistency. The consistency between objects in different caches is categorized into the following four levels:

- [Using Local Objects](#) (No consistency requirements)
- [Propagating Changes Without Waiting for a Reply](#)
- [Propagating Changes and Waiting for a Reply](#)
- [Serializing Changes Across Multiple Caches](#)

Using Local Objects

If there are no consistency requirements between objects in distributed caches, then define an object as a local object. (When `Attributes.DISTRIBUTE` is unset, this specifies a local object.) Local is the default setting for objects. For local objects, all updates and invalidation are visible to only the local cache.

Propagating Changes Without Waiting for a Reply

To distribute object updates across distributed caches, define an object as distributed by setting the `DISTRIBUTE` attribute. All modifications to distributed objects are broadcast to other caches in the system. Using this level of consistency does not control

or specify when an object is loaded into the cache or updated, and does not provide notification as to when the modification has completed in all caches.

Propagating Changes and Waiting for a Reply

To distribute object updates across distributed caches and wait for the change to complete before continuing, set the object's `DISTRIBUTE` and `REPLY` attributes. When you set these attributes, notification occurs when a modification has completed in all caches. When you set `Attributes.REPLY` for an object, replies are sent back to the modifying cache when the modification has been completed at the remote site. These replies are returned asynchronously—that is, the `CacheAccess.replace()` and `CacheAccess.invalidate()` methods do not block. Use the `CacheAccess.waitForResponse()` method to wait for replies and block.

Serializing Changes Across Multiple Caches

To use the highest level of consistency of the Java Object Cache, set the appropriate attributes on the region, subregion, group, or object to make objects act as synchronized objects.

When you set `Attributes.SYNCHRONIZE_DEFAULT` on a region, subregion, or group, it sets the `SYNCHRONIZE` attribute for all the objects within the region, subregion, or group.

When you set `Attributes.SYNCHRONIZE` on an object, it forces applications to obtain ownership of the object before the object can be loaded or modified. Setting this attribute effectively serializes write access to objects. To obtain ownership of an object, use the `CacheAccess.getOwnership()` method. When you set the `Attributes.SYNCHRONIZE` attribute, notification is sent to the owner when the update is completed. Use `CacheAccess.releaseOwnership()` to block until any outstanding updates have completed and the replies are received. This releases ownership of the object so that other caches can update or load the object.

Note: Setting `Attributes.SYNCHRONIZE` for an object is not the same as setting `synchronized` on a Java method. With `Attributes.SYNCHRONIZE` set, the Java Object Cache forces the cache to serialize creates and updates of the object, but does not prevent the Java programmer from obtaining a reference to the object and then modifying the object.

When using this level of consistency, with `Attributes.SYNCHRONIZE`, the `CacheLoader.load()` method calls `CacheLoader.netSearch()` before loading the object from an external source. Calling `CacheLoader.netSearch()` in the load method tells the Java Object Cache to search all other caches for a copy of the object. This process prevents different versions of the object from being loaded into the cache from an external source. Proper use of the `SYNCHRONIZE` attribute, along with the `REPLY` attribute and the `invalidate` method, supports consistency of objects across the cache system

Sharing Cached Objects in an OC4J Servlet

To take advantage of the distributed functionality of the Java Object Cache or to share a cached object among servlets, some minor modification to an application's deployment may be necessary. Any user-defined objects that will be shared among servlets or distributed among JVMs must be loaded by the system class loader. By default, objects that are loaded by a servlet are loaded by the context class loader.

These objects are visible only to the servlets within the context that loaded them. The object definition is not available to other servlets or to the cache in another JVM. If the object is loaded by the system class loader, the object definition is available to other servlets and to the cache on other JVMs.

Note: Deprecated: With the Apache JServ servlet environment (JServ), the preceding functionality was accomplished by including the cached object in the `classpath` definition available when the JServ process was started.

With OC4J, the system `classpath` is derived from the manifest of the `oc4j.jar` file and any associated JAR files, including `cache.jar`. The `classpath` in the environment is ignored. To include a cached object in the `classpath` for OC4J, copy the class file to `ORACLE_HOME/javacache/sharedobjects/classes`, or add it to the JAR file `ORACLE_HOME/javacache/cachedobjects/share.jar`. Both the `classes` directory and the `share.jar` file have been included in the manifest for `cache.jar`.

Using User-Defined Class Loaders

You can place objects in the cache that require user-defined class loaders. The Cache Service supports user-defined class loaders by providing two methods:

`Attributes.setClassLoader()` and `Attributes.getClassLoader()`. Once you have set a region or a group to use a user-defined class loader, all objects under that region or group are loaded using this class loader (by inheriting attributes). This attribute does not apply to objects that are not region or group, and will be ignored if set.

The following example demonstrates setting user-defined class loader. Object A is loaded using `MyClassLoader`.

```
import oracle.ias.cache.*;
import java.lang.ClassLoader;
ClassLoader loader = this.getClass().getClassLoader();
CacheAttributes cAttr = new CacheAttributes();
Attributes attr = new Attributes();
CacheAccess cacc;
Cache.init(cAttr);
attr.setClassLoader(loader);
CacheAccess.defineRegion("region A", attr);
cacc = CacheAccess.getAccess("region A");
cacc.get("object A")
```

Note that user-defined class loaders can only be used for object contents, not object names. In other words, in the line

```
cacc.put(name, object);
```

`object` can require a user-defined class loader, but `name` cannot.

HTTP and Security for Distributed Cache

This section discusses HTTP and security for distributed cache.

HTTP

By default, the Cache Service uses a proprietary protocol built on top of TCP to communicate between caches. In addition to the proprietary protocol, the Cache Service also supports using HTTP to communicate between caches. While the proprietary protocol is kept for compatibility reasons, some of the newer functionalities are implemented exclusively for HTTP. In particular, in a distributed cache system, when getting disk or stream objects from remote caches, HTTP mode is required. `CacheAccess.getAllCached()`, `CacheLoader.getFromRemote()`, and `CacheLoader.netSearch()` are three example operations where HTTP mode is required for dealing with disk and stream objects. To enable HTTP mode, you must set `CacheAttributes.transport`.

To use HTTP, you must enable it on all of the caches within a distributed system. The functionality that previously used the proprietary protocol will also work with HTTP enabled.

The following examples show two ways to enable HTTP mode.

Example 1 - Enabling HTTP in `cache_attributes.xml` :

```
-----
<?xml version="1.0" encoding="UTF-8"?>
<cache-configuration xmlns="http://www.oracle.com/oracle/ias/cache/configuration"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <communication>
    <isDistributed>true</isDistributed>
    <transport>HTTP</transport>
  </communication>
</cache-configuration>
```

```
import oracle.ias.cache.*;
Cache.open("cache_attributes.xml");
```

Example 2 - Enabling HTTP in code:

```
import oracle.ias.cache.*;
CacheAttributes cAttr = new CacheAttributes();
cAttr.distribute = true;
cAttr.transport = CacheAttributes.HTTP;
Cache.init(cAttr);
```

SSL

For secure communication between caches, the Cache Service supports the SSL protocol. The JDK keytool program can be used to generate certificates and set up the keystore, as documented on Sun's J2SE 1.4.2 Key and Certificate Management Tool web page. The same key pair and certificate used for OC4J can be used for the Cache Service.

To use SSL, you must enable it on all of the caches within a distributed system.

After setting up the keystore, you need to tell the cache where the keystore is with this command:

```
java -jar $ORACLE_HOME/javacache/lib/cache.jar sslconfig <cache_attributes.xml>
<keystore_file> <password>
```

where

- `$ORACLE_HOME` is the home directory of the Oracle IAS instance.
- `cache_attributes.xml` is your cache configuration file.
- `keystore_file` is the full path to your keystore file as generated by keytool.
- `password` is the password you used in keytool to generate the key pair.

This generates an SSL configuration file to be used by the cache, where the name of the file is as specified in `cache_attributes.xml`. In addition, you need to set `CacheAttributes.isSSLEnabled` to `true`.

The following examples show two ways to enable SSL:

Example 1 - Enabling SSL in `cache_attributes.xml` :

```
<?xml version="1.0" encoding="UTF-8"?>
<cache-configuration
  xmlns="http://www.oracle.com/oracle/ias/cache/configuration"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <communication>
    <isDistributed>true</isDistributed>
    <useSSL>true</useSSL>
    <keyStore>.keyStore</keyStore>
    <sslConfigFile>.sslConfig</sslConfigFile>
  </communication>
</cache-configuration>
```

```
import oracle.ias.cache.*;
Cache.open("cache_attributes.xml");
```

Example 2 - Enabling SSL in code:

```
import oracle.ias.cache.*;
CacheAttributes cAttr = new CacheAttributes();
cAttr.distribute = true;
cAttr.isSSLEnabled = true;
cAttr.keyStoreLocation = ".keyStore";
cAttr.sslConfigFilePath = ".sslConfig";
Cache.init(cAttr);
```

Two caches must be using the same set of keys to communicate with each other. If the caches in a system reside on multiple machines, then you need to copy the keystore file to all machines and run the `java -jar ...` command for every cache configuration file in the system.

Firewall

To make a distributed cache system work across a firewall, the current workaround is to enable a set of outbound TCP ports at the firewall and to define them in `cache_attributes.xml`.

For example, `cache_attributes.xml` might look something like this if the ports are within the range of 7100 to 7199:

```
cache_attributes.xml
-----
<?xml version = '1.0' encoding = 'UTF-8'?>
```

```

<cache-configuration xmlns="http://www.oracle.com/oracle/ias/cache/configuration"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <communication>
    <isDistributed>true</isDistributed>
    <useSSL>false</useSSL>
    <sslConfigFile>.sslConfig</sslConfigFile>
    <port lower="7100" upper="7199"/>
    <discoverer discovery-port="7100" original="true" xmlns=""/>
  </communication>
</cache-configuration>

```

Make sure that the `discovery-port` is within the range specified.

Restricting Incoming Connections

For systems that are configured with more than one address to support multiple network subnets (private and public, for example), you can specify a configuration element, `localAddress`, in `cache_attributes.xml` to restrict incoming connections to a specified local address. By default, the distributed cache system will bind the listener socket to the primary host address returned by the operating system. If `localAddress` is specified, however, the cache will bind the listener socket to the specified address. The value specified for `localAddress` must be a fully qualified hostname or IP address. For example:

```

cache_attributes.xml
-----
<?xml version = '1.0' encoding = 'UTF-8'?>
<cache-configuration xmlns="http://www.oracle.com/oracle/ias/cache/configuration"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <communication>
    <isDistributed>true</isDistributed>
    <localAddress>123.456.78.90</localAddress>
    <discoverer discovery-port="7100" original="true" xmlns=""/>
  </communication>
</cache-configuration>

```

or

```

cache_attributes.xml
-----
<?xml version = '1.0' encoding = 'UTF-8'?>
<cache-configuration xmlns="http://www.oracle.com/oracle/ias/cache/configuration"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <communication>
    <isDistributed>true</isDistributed>
    <localAddress>computer.oracle.com</localAddress>
    <discoverer discovery-port="7100" original="true" xmlns=""/>
  </communication>
</cache-configuration>

```

In the latter case, even if the IP underneath a virtual hostname changes, JOC will remain unaffected.

Monitoring and Debugging

Besides `Cache.listCacheObjects()` and `Cache.dump()`, the Cache Service provides additional methods to reflect the current status of the cache and of the regions, groups, and individual objects within the cache. These methods can be found in the classes `CacheAccess`, `Cache`, and `AggregateStatus`.

The methods under `Cache` reflect the cache's own status. `getActiveHostInfo` returns an array of `CacheHostInfo` objects for all active caches in a cache system. `getCacheSize` estimates total space (bytes) taken up by memory objects in the cache. `getDistributedDiskCacheSize` and `getLocalDiskCacheSize` estimate total space (bytes) taken up by objects in the distributed and local disk caches, respectively. `getObjectCount` returns the current total number of objects in the cache.

The methods under `CacheAccess` reflect region, group, and individual object status. `listNames` names all objects under the region. `listObjects` names all objects under the region and also provides access to them. `listRegions` names all sub-regions under the region. These three methods are not recursive. For example, `listRegions` does not list the sub-regions under the sub-regions of the region.

`CacheAccess.getStatus()` reflects more detailed status information for a named individual object or group under the region in the form of an `ObjectStatus` object. This includes the cached object's access count, time of creation, size on disk (if stored on disk), last time of access, loading time (ms), priority (as set by the object's creator), and size in cache (bytes). If no object or group name is specified, `getStatus` returns the status of the region.

`CacheAccess.getAggregateStatus()`, on the other hand, returns overall statistics for a named group or region (sub-region) in the form of an `AggregateStatus` object. `AggregateStatus` reflects the low, average, and high values of attributes of the objects under the region or group. These attributes include access count, time of creation, last time of access, loading time, priority, and size in cache. In addition, the `AggregateStatus` object also includes the total object count for the region or group. Reflection methods in the `AggregateStatus` class allows you to access all of these numbers individually.

The Cache Service automatically compiles the information reflected by `getAggregateStatus` during every clean interval. To obtain the latest information, you need to call `Cache.updateStats()` before calling `getAggregateStatus`.

Here is an example of using `getAggregateStatus`

```
import oracle.ias.cache.*;
import java.util.Date;
import java.io.*;

CacheAccess cacc;

// create objects, load objects, etc.
...

AggregateStatus aggStats;
long          avgCreateTime;
Date          avg;

Cache.updateStats();
aggStats = cacc.getAggregateStatus();
avgCreateTime = aggStats.getCreateTime(AggregateStatus.AVG);
avg = new Date(avgCreateTime);

System.out.println("average creation time: " + avg);
```

CacheWatchUtil

By default, the Cache Service provides the `CacheWatchUtil` cache monitoring utility that can display current caches in the system, display a list of cached objects, display caches' attributes, reset cache logger severity, dump cache contents to the log, and so on.

To invoke `CacheWatchUtil`, while caches are running, type one of the following commands:

```
java oracle.ias.cache.CacheWatchUtil [-config=cache_config.xml] [-help]
```

or

```
java -jar $ORACLE_HOME/javacache/lib/cache.jar watch [-config=cache_config.xml] [-help]
```

where “-config=” and “-help” are optional parameters, and `cache_config.xml` is a cache configuration file.

- “help” gives you a list of commands you can invoke in the cache watcher. Among these commands,
- “set severity=<level> [CacheId]” sets logger severity level for a particular cache. The levels are:
 - -1 off
 - 0 fatal
 - 3 error
 - 4 default
 - 6 warning
 - 7 trace
 - 10 info
 - 15 debug
- “set timeout=<value>” sets group communication timeout for the cache system to value.
- “dump [CacheId]” dumps the contents of a particular cache to the log file.
- “invalidate” invalidates all objects in the cache system.
- “destroy” destroys all objects in the cache system (include memory, stream, and disk).

Typing “get config [CacheId]” returns the cache configuration information for a particular cache. You can retrieve remote cache configurations for verification, as shown in the following example.

```
cache> get config 3
ache 3 at localhost:53977
distribute = true
version = 9.0.4
max objects = 200
max cache size = 48
diskSize = 32
```

```

diskPath = <disk_path>
clean interval = 3
LogFileName = <log_file_name>
Logger = MyCacheLogger
Log severity = 3
cache address list = [127.0.0.1:22222, pos=-1, uid=0, orig, name=, pri=0
]

```

Typing “list caches” or “lc” lists all of the active caches in the system. The cache watcher also occupies a spot on the list, as shown in the following example. The UID column displays every cache’s ID. The cache watcher does not detect caches that have been configured but are not active.

```

cache> lc
Current coordinator: [127.0.0.1:53957, pos=0, uid=0, tag=27979955, pri=0]
#      UID      CacheAddress
-      -
1      0      localhost:53957
2      1      localhost:53965
3      2      localhost:53974
4      3      localhost:53977
5      4      localhost:53980
6      5      localhost:53997 <-- this cache watcher

```

Typing

```
"list objects [CacheId] [region=<region>] [sort=<0...7>]"
```

or

```
"lo [CacheId] [region=<region>] [sort=<0...7>]"
```

lists all objects in a specified cache, under a specified region, and in the order specified by the sort option. The sort options are:

- 0 by region name
- 1 by object name
- 2 by group name
- 3 by object type
- 4 by valid status
- 5 by reference count
- 6 by access count
- 7 by expiration

Without any options, lo lists all objects in all caches without sorting. The following example shows lo for cache 3, A-Region, sorted by object name. Columns have been adjusted to improve example readability.

```

cache> lo 3 region=A-Region sort=1
Cache 3 at localhost:53977
REGION  OBJNAME  GROUP  TYPE  REFCNT  ACCCNT  EXPIRE  VALID  LOCK
-----  -
[A-Region] [A-Group] [A-Region] Group  0      1      None      true  null
[A-Region] [A-Region] [null]    Region 0      4      295 Seconds true  null

```

[A-Region] [B-Group]	[A-Region] Group	0	3	None	true	null
[A-Region] [bar]	[B-Group] Loader	0	1	None	true	null

Finally, typing “groupdump” dumps all group communication information for all caches to the log file. It is unlikely that you will need to use this command or that you will find its output useful, but in the event of group communication errors, technical support might ask you to supply the information for problem diagnosis.

XML Schema for Cache Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.oracle.com/oracle/ias/cache/configuration"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.oracle.com/oracle/ias/cache/configuration"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="cache-configuration" type="CacheConfigurationType">
    <xs:annotation>
      <xs:documentation>Oracle JavaCache implementation</xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:complexType name="CacheConfigurationType">
    <xs:sequence>
      <xs:element name="logging" type="loggingType" minOccurs="0"/>
      <xs:element name="communication" type="communicationType" minOccurs="0"/>
      <xs:element name="persistence" type="persistenceType" minOccurs="0"/>
      <xs:element name="preload-file" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="max-objects" type="xs:positiveInteger" default="1000" minOccurs="0"/>
      <xs:element name="max-size" type="xs:positiveInteger" default="1000" minOccurs="0"/>
      <xs:element name="clean-interval" type="xs:positiveInteger" default="60" minOccurs="0"/>
      <xs:element name="ping-interval" type="xs:positiveInteger" default="60" minOccurs="0"/>
      <xs:element name="cacheName" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="loggingType">
    <xs:sequence>
      <xs:element name="location" type="xs:string" minOccurs="0"/>
      <xs:element name="level" type="loglevelType" minOccurs="0"/>
      <xs:element name="logger" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="communicationType">
    <xs:sequence>
      <xs:element name="isDistributed" type="xs:boolean" default="false" minOccurs="0"/>
      <xs:element name="transport" type="transportType" minOccurs="0"/>
      <xs:element name="useSSL" type="xs:boolean" minOccurs="0"/>
      <xs:element name="sslConfigFile" type="xs:string" minOccurs="0"/>
      <xs:element name="keyStore" type="xs:string" minOccurs="0"/>
      <xs:element name="port" minOccurs="0">
        <xs:complexType>
          <xs:attribute name="lower" type="xs:nonNegativeInteger" use="optional" default="0"/>
          <xs:attribute name="upper" type="xs:nonNegativeInteger" use="optional" default="0"/>
        </xs:complexType>
      </xs:element>
      <xs:element name="localAddress" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

```



```

<xs:element name="discoverer" minOccurs="0" maxOccurs="unbounded">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="discovererType">
        <xs:attribute name="order" type="xs:nonNegativeInteger"/>
        <xs:attribute name="original" type="xs:boolean"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:element name="discovererElection" type="electionType" minOccurs="0"/>
</xs:sequence>
</xs:complexType>
<xs:complexType name="discovererType">
  <xs:attribute name="ip" type="xs:string"/>
  <xs:attribute name="discovery-port" type="xs:positiveInteger" use="required"/>
</xs:complexType>
<xs:complexType name="persistenceType">
  <xs:sequence>
    <xs:element name="location" type="xs:string"/>
    <xs:element name="disksize" type="xs:positiveInteger" default="30" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
<xs:simpleType name="loglevelType">
  <xs:restriction base="xs:token">
    <xs:enumeration value="OFF"/>
    <xs:enumeration value="FATAL"/>
    <xs:enumeration value="ERROR"/>
    <xs:enumeration value="DEFAULT"/>
    <xs:enumeration value="WARNING"/>
    <xs:enumeration value="TRACE"/>
    <xs:enumeration value="INFO"/>
    <xs:enumeration value="DEBUG"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="transportType">
  <xs:restriction base="xs:token">
    <xs:enumeration value="TCP"/>
    <xs:enumeration value="HTTP"/>
  </xs:restriction>
</xs:simpleType>
<xs:complexType name="electionType">
  <xs:sequence>
    <xs:element name="useMulticast" type="xs:boolean" minOccurs="0"/>
    <xs:element name="updateInterval" type="xs:positiveInteger" minOccurs="0"/>
    <xs:element name="resolutionInterval" type="xs:positiveInteger" minOccurs="0"/>
    <xs:element name="multicastAddress" minOccurs="0">
      <xs:complexType>
        <xs:attribute name="ip" type="xs:string" use="optional"/>
        <xs:attribute name="port" type="xs:string" use="optional"/>
        <xs:attribute name="TTL" type="xs:nonNegativeInteger" use="optional"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="usePriorityOrder" type="xs:boolean" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
</xs:schema>

```

XML schema for attribute declaration

```

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.oracle.com/oracle/ias/cache/configuration/declarative"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.oracle.com/oracle/ias/cache/configuration/declarative"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:complexType name="regionType">
    <xs:sequence>
      <xs:element name="attributes" type="attributesType" minOccurs="0"/>
      <xs:element name="region" type="regionType" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="group" type="groupType" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="cached-object" type="cached-objectType" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:complexType name="attributesType">
    <xs:sequence>
      <xs:element name="time-to-live" type="xs:positiveInteger" minOccurs="0"/>
      <xs:element name="default-ttl" type="xs:positiveInteger" minOccurs="0"/>
      <xs:element name="idle-time" type="xs:positiveInteger" minOccurs="0"/>
      <xs:element name="version" type="xs:string" minOccurs="0"/>
      <xs:element name="max-count" type="xs:positiveInteger" minOccurs="0"/>
      <xs:element name="priority" type="xs:positiveInteger" minOccurs="0"/>
      <xs:element name="size" type="xs:positiveInteger" minOccurs="0"/>
      <xs:element name="flag" minOccurs="0" maxOccurs="unbounded">
        <xs:simpleType>
          <xs:restriction base="flagType">
            <xs:enumeration value="distribute"/>
            <xs:enumeration value="reply"/>
            <xs:enumeration value="synchronize"/>
            <xs:enumeration value="spool"/>
            <xs:enumeration value="group_ttl_destroy"/>
            <xs:enumeration value="original"/>
            <xs:enumeration value="synchronize-default"/>
            <xs:enumeration value="allownull"/>
            <xs:enumeration value="measure"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="event-listener" type="event-listenerType" minOccurs="0"/>
      <xs:element name="cache-loader" type="userDefinedObjectType" minOccurs="0"/>
      <xs:element name="capacity-policy" type="userDefinedObjectType" minOccurs="0"/>
      <xs:element name="user-defined" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="key" type="xs:string"/>
            <xs:element name="value" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="flagType">
    <xs:list itemType="xs:token"/>
  </xs:simpleType>
  <xs:complexType name="userDefinedObjectType">
    <xs:sequence>
      <xs:element name="classname" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>

```

```

        <xs:element name="parameter" type="propertyType" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="propertyType">
    <xs:simpleContent>
        <xs:extension base="xs:string">
            <xs:attribute name="name" type="xs:string" use="required"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>
<xs:complexType name="event-listenerType">
    <xs:sequence>
        <xs:element name="classname" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="handle-event" type="handle-eventType" use="required"/>
    <xs:attribute name="default" type="xs:boolean"/>
</xs:complexType>
<xs:simpleType name="handle-eventType">
    <xs:restriction>
        <xs:simpleType>
            <xs:list itemType="xs:token"/>
        </xs:simpleType>
        <xs:enumeration value="object-invalidated"/>
        <xs:enumeration value="object-updated"/>
    </xs:restriction>
</xs:simpleType>
<xs:complexType name="groupType">
    <xs:sequence>
        <xs:element name="attributes" type="attributesType" minOccurs="0"/>
        <xs:element name="group" type="groupType" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="cached-object" type="cached-objectType" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="cached-objectType">
    <xs:sequence>
        <xs:element name="attributes" type="attributesType" minOccurs="0"/>
        <xs:element name="name" type="nameType" minOccurs="0"/>
        <xs:element name="object" type="userDefinedObjectType" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="nameType">
    <xs:choice>
        <xs:element name="string-name" type="xs:string"/>
        <xs:element name="object-name" type="userDefinedObjectType"/>
    </xs:choice>
</xs:complexType>
<xs:element name="cache">
    <xs:complexType>
        <xs:sequence maxOccurs="unbounded">
            <xs:element name="region" type="regionType"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:schema>

```


Numerics

- 1pc
 - See single-phase commit
- 2pc
 - See two-phase commit

A

- abnormal termination
 - OC4J JMS, 3-14
- AbstractPrincipalMapping
 - extending, 8-16
- administration
 - OracleAS JMS, 3-21
- administration properties
 - OracleAS JMS, 3-22
- admin.jar
 - resource adaptors, deploying, 8-7
 - resource adaptors, undeploying, 8-7
- admin.jar tool, 6-3, 6-4
- ALLOWNULL Java Object Cache attribute, 9-12
- ApplicationClientInitialContextFactory, 2-4
- application-client.jar
 - JNDI, 2-3, 2-5
- application-client.xml, 6-13
 - JNDI, 2-5
- <application-server> element, 8-9
- application.xml, 7-8
 - <data-sources> tag, 4-7
 - designating data-sources.xml, 4-7
 - location, 4-7
- AQ, 3-24
- Attributes.setCacheEventListener() method, 9-35
- authentication classes
 - OC4J-specific, 8-15

B

- bean-managed transactions
 - MDBs, and JMS clients, 7-13
- BMT
 - recovery, 7-11
- browse
 - JMS utility, 3-9

C

- cache
 - concepts, 9-1
- cache region, 9-8
- CacheAccess
 - createPool() method, 9-42
- CacheAccess.getOwnership() method, 9-47
- CacheAccess.releaseOwnership() method, 9-47
- CacheAccess.save() method, 9-39
- CacheEventListener
 - Java Object Cache attribute, 9-13
- CacheEventListener interface, 9-35
- CacheLoader.createStream() method, 9-41
- caching scheme, 4-21
- CapacityPolicy
 - Java Object Cache attribute, 9-13
- check
 - JMS utility, 3-9
- class
 - <data-source> attribute, 4-8
- clean-available-connections-threshold
 - <data-source> attribute, 4-9
- clean-interval configuration XML element, 9-23
- client-side installation requirements
 - RMI/IIOP, 6-2
 - RMI/ORMI, 5-2
- client-side transaction demarcation, 7-6
- clustering
 - issues, JMS and OracleAS JMS, 3-50
 - JNDI
 - enabling, 2-11
 - limitations, 2-11
 - overview, 2-11
- CMP
 - connection recovery, 7-11
 - retry count, 7-11
- CMT
 - recovery, 7-11
 - retry count, 7-11
- com.evermind.server package
 - ApplicationClientInitialContextFactory, 5-9, 6-13
 - ApplicationInitialContextFactory, 5-9
 - JNDI, 2-4
 - RMIInitialContextFactory, 5-9
- <commit-class> element, 7-9

- Common Secure Interoperability Version 2
 - See* CSiv2
- com.oracle.iioop.server package
 - IIOPInitialContextFactory, 6-13
- component-managed sign-on, 8-12
- configuration
 - two-phase commit transaction, 7-7
- configuration elements
 - OracleAS JMS hierarchical tree, 3-3
- configuration files
 - data sources, 4-7
- configuring
 - connection pooling, 8-10
 - high availability, 3-47
 - high availability, OJMS, 3-47
 - high availability, OracleAS JMS, 3-41
 - JMS provider, 3-25
 - OC4J for interoperability, 6-12
 - OC4J mount point in RMI/IIOP, 5-12
 - OC4J to support RMI tunneling, 5-11
 - Oracle JMS, 3-27
 - Oracle JMS provider in OC4J XML files, 3-27
 - OracleAS JMS, 3-2
 - OracleAS JMS ports, 3-2
 - resource provider with data sources
 - property, 3-28
 - single-phase commit, 7-2
 - timeouts in server.xml, 7-10
 - timeouts JTA, 7-10
- configuring custom resource provider
 - JMS, 3-23
- configuring resource provider
 - OJMS, 3-28, 3-29
- connection factories
 - default, in JMS, 3-4
- connection factory
 - configuration examples, 3-20
- connection pooling
 - configuring, 8-10
- connection-driver
 - <data-source> attribute, 4-8
- ConnectionFactory
 - JMS, 3-4
- connection-factory element, 3-18
- connection-retry-interval
 - <data-source> attribute, 4-9
- connection-retry-interval
 - <data-source><default>, 4-9
- constructing
 - JNDI contexts, 2-2
 - JNDI InitialContext, 2-3
- container-managed sign-on, 8-13
- container-managed transactions
 - MDBs, 7-12
- <container-transaction> element, 7-5
- context factory
 - usage, 5-9, 6-13
- Context.bind API call, 2-2
- contextFactory
 - ApplicationClientInitialContextFactory, 6-13

- IIOPInitialContextFactory, 6-13
- contextFactory property, 6-13
- context.SECURITY_CREDENTIAL
 - JNDI-related environment properties, 2-6
- context.SECURITY_PRINCIPAL
 - JNDI-related environment properties, 2-6
- copy
 - JMS utility, 3-9
- CORBA Object Service Naming
 - See* CosNaming
- CORBA Transaction Service
 - see* OTS
- corbaname URL, 6-10
- CosNaming, 6-1, 6-10
- createDiskObject() method, 9-18, 9-39
- createInstance() method, 9-43
- CreatePool() method, 9-42
- createStream() method, 9-18
- CSiv2, 6-1

D

- Data Guard, 4-26
- data source
 - configuration, 4-6
 - configuration file, 4-7
 - connection sharing, 4-18
 - default, 4-11
 - definition, 4-1
 - emulated, 4-11
 - error conditions, 4-18
 - JDBC driver, 4-19
 - user name, 4-19
 - JDBC connections, 4-1
 - JNDI, 4-1
 - location of XML file, 4-7
 - nonemulated, 4-4
 - behavior, 4-18
 - JTA transaction, 4-18
 - Oracle JDBC extensions, 4-21
 - portable, lookup, 4-15
 - retrieving connection, 4-15
 - using DataDirect driver, 4-23
 - using OCI driver, 4-22
- <data-source> attribute
 - min-connections, 4-9
- data source entry
 - SQLServer, with DataDirect, 4-24
- data sources
 - configuration files, 4-7
 - defining, 4-6
 - defining in Enterprise Manager, 4-10
 - emulated, 4-2
 - introduction, 4-1
 - mixing, 4-5
 - native, 4-4
 - nonemulated, 4-4
 - summary, 1-2
 - two-phase commit, 4-19
 - types, 4-1

- using, 4-15
- data sources property
 - configuring resource provider, 3-28
- database
 - caching scheme, 4-21
- database configuration, 7-7
- database-schema, 4-9, 4-14
- DataDirect driver, 4-23
- DataDirect JDBC drivers
 - installing, 4-24
- <data-source>
 - attributes, 4-8
- <data-source> attribute
 - class, 4-8
 - clean-available-connections-threshold, 4-9
 - connection-driver, 4-8
 - connection-retry-interval, 4-9
 - ejb-location, 4-9
 - inactivity-timeout, 4-9
 - location, 4-8
 - max-connect-attempts, 4-9
 - max-connections, 4-9
 - name, 4-8
 - password, 4-9
 - rac-enabled, 4-9
 - schema, 4-9, 4-14
 - stmt-cache-size, 4-9
 - URL, 4-9
 - username, 4-8
 - wait-timeout, 4-9
 - xa-location, 4-9
- <data-source> attribute
 - connection-retry-interval, 4-9
- DataSource object
 - look-up, 4-17, 7-3
 - retrieving, 7-3
 - types, 4-1
- <data-source> tag, 4-8
- data-sources.xml, 4-6, 7-9
 - about, 4-7
 - designating location, 4-7
 - EAR file, 4-8
 - location, 4-7
 - preinstalled definitions, 4-11
 - use in JTA, 7-2
- DBMS_AQADM package, 3-26
- DBMS_AQADM.CREATE_QUEUE, 3-26
- DbUtil
 - oracleFatalError method, 7-11
- dcmctl
 - resource adaptors, deploying, 8-7
 - resource adaptors, undeploying, 8-7
- declarative container-managed sign-on, 8-14
- dedicated.rmicontext
 - JNDI-related environment properties, 2-5
- DefaultTimeToLive
 - Java Object Cache attribute, 9-13
- default-web-site.xml, 5-13
- defineGroup() method, 9-16
- defineObject() method, 9-17

- defineRegion() method, 9-15
- deployment
 - and interoperability, 6-12
- deployment descriptor
 - J2EE Connector, 8-3
 - JTA, 7-5
 - JTA attribute
 - Mandatory transaction attribute type, 7-5
 - Never transaction attribute type, 7-5
 - NotSupported transaction attribute type, 7-5
 - Required transaction attribute type, 7-5
 - RequiresNew transaction attribute type, 7-5
 - Supports transaction attribute type, 7-5
- dequeue-retry-count, 7-12
- dequeue-retry-interval, 7-12
- destinations
 - JMS utility, 3-9
- destroy() method, 9-19
- destroyInstance() method, 9-43
- disallowed-field, 4-15
- discoveryAddress property, 9-45
- DISTRIBUTE
 - Java Object Cache attribute, 9-11, 9-44, 9-45
- distributed transaction coordinator, 4-19
- drain
 - JMS utility, 3-9
- DTC, 4-19
- durables
 - JMS utility, 3-9

E

- EJB
 - interoperability, 6-1
 - making interoperable, 6-3, 6-6
- EJB interoperability
 - introduction, 6-1
- ejb-jar.xml
 - <message-driven-deployment> element, 7-12
- ejb-jar.xml, 8-5
- ejb-location
 - <data-source> attribute, 4-9
- emulated data sources, 4-2
- Enterprise Information Systems (EISs), 8-1
- Enterprise Manager
 - defining data sources, 4-10
- <entity-deployment> element, 6-12
- environment properties
 - JNDI-related, 2-5
- example
 - JNDI, servlet retrieves data source, 2-8
- examples
 - connection factory configuration, 3-20
- exception queue, predefined
 - OracleAS JMS, 3-14
- exceptionHandler() method, 9-18

F

- file-based persistence

- OracleAS JMS, 3-10
- files
- interoperability deployment, 6-12
- flags
- OC4J, starting interoperably, 6-12

G

- generated stub JAR file, 6-3, 6-4
- getConnection method, 4-17, 7-3, 7-4
- getID() method, 9-35
- getName() method, 9-18
- getOwnership() method, 9-47
- getOwnership() method, 9-50
- getParent() method, 9-16
- getRegion() method, 9-18
- getSource() method, 9-35
- global-web-application.xml, 5-11
- GROUP_TTL_DESTROY
 - Java Object Cache attribute, 9-11
- GROUP_TTL_DESTROY attribute, 9-19

H

- handleEvent() method, 9-35
- help
 - JMS utility, 3-9
- HiAvailability
 - and clustering, JMS, 3-40
- hierarchical tree
 - OracleAS JMS configuration elements, 3-3
- High Availability
 - Data Guard, 4-26
 - network failover, 4-26
 - Oracle Maximum Availability Architecture, 4-26
 - Real Application Clusters, 4-26
 - SQL exceptions, 4-32
 - TAF, 4-27
- high availability, 3-41, 3-47
 - configuring, 3-47
- High Availability Summary, 3-40
- http.tunnel.path
 - JNDI-related environment properties, 2-5

I

- identifying objects, 9-6
- IdleTime
 - Java Object Cache attribute, 9-13
- IIOP, 1-2, 6-1
- iiopClientJar switch, 6-3, 6-4
- IIOPInitialContextFactory, 2-10
- import
 - oracle.ias.cache, 9-15
- inactivity-timeout
 - <data-source> attribute, 4-9
- initial context
 - creating in OC4J, 2-4
 - JNDI, 2-1
- initial context factories
 - accessing objects in same application, 2-8

- accessing objects not in same application, 2-9
- JNDI, 2-4
- INITIAL_CONTEXT_FACTORY
 - InitialContext property, 2-3
- InitialContext
 - constructing in JNDI, 2-3
 - constructors, 2-3
- InitialContext object, 2-2
- InitialContext properties
 - INITIAL_CONTEXT_FACTORY, 2-3
 - PROVIDER_URL, 2-3
 - SECURITY_CREDENTIAL, 2-4
 - SECURITY_PRINCIPAL, 2-4
- installing
 - client-side, RMI/IIOP, 6-2
 - client-side, RMI/ORMI, 5-2
 - JMS provider, 3-25
 - OC4J client JAR files, 5-2, 6-2
- Internet Inter-ORB Protocol
 - See IIOP
- interoperability
 - adding to EJB, 6-3, 6-6
 - advanced, configuring manually, 6-8
 - advanced, configuring with Oracle Enterprise Manager, 6-8
 - advanced, in OracleAS environment, 6-8
 - configuring OC4J for, 6-12
 - files configuring, 6-12
 - naming, 6-1
 - OC4J flags, 6-12
 - overview, 1-2, 6-1
 - security, 6-1
 - simple, configuring manually, 6-6
 - simple, configuring with Oracle Enterprise Manager, 6-5
 - simple, in OracleAS environment, 6-5
 - transaction, 6-1
 - transport, 6-1
- interoperability, advanced
 - in standalone environment, 6-4
- interoperability, simple
 - in standalone environment, 6-3
- interoperable transport, 6-3
- introduction to data sources, 4-1
- introduction to OC4J services, 1-1
- invalidate() method, 9-19
- <ior-security-config> element, 6-12

J

- J2EE application clients
 - JNDI initial contexts, 2-4
- J2EE application components
 - JNDI initial contexts, 2-8
- J2EE Connector, 8-1
 - deployment descriptor, 8-3
 - resource adapters, 8-1
- J2EE Connector Architecture
 - deployment directory locations, 8-9
 - file locations, 8-10

- summary, 1-2
- JAAS
 - pluggable authentication classes, 8-18
- Java Message Service, *See* JMS
- Java Naming and Directory Interface
 - See* JNDI
- Java Object Cache, 9-1, 9-2
 - attributes, 9-10
 - basic architecture, 9-2
 - basic interfaces, 9-4
 - cache consistency levels, 9-49
 - cache environment, 9-8
 - classes, 9-4
 - configuration
 - clean-interval XML element, 9-23
 - maxObjects property, 9-23
 - maxSize property, 9-23
 - ping-interval XML element, 9-23
 - consistency levels
 - distributed with reply, 9-50
 - distributed without reply, 9-49
 - local, 9-49
 - synchronized, 9-50
 - default region, 9-8
 - defining a group, 9-16
 - defining a region, 9-15
 - defining an object, 9-17
 - destroy object, 9-19
 - disk cache
 - adding objects to, 9-38
 - disk objects, 9-38
 - definition of, 9-7
 - distributed, 9-39
 - local, 9-39
 - using, 9-39
 - distribute property, 9-45
 - distributed disk objects, 9-38
 - distributed groups, 9-46
 - distributed mode, 9-45
 - distributed objects, 9-46
 - distributed regions, 9-46
 - features, 9-6
 - group, 9-8
 - identifying objects, 9-6
 - invalidating objects, 9-19
 - local disk objects, 9-38
 - local mode, 9-44
 - memory objects
 - definition of, 9-6
 - local memory object, 9-6
 - spooled memory object, 9-6
 - updating, 9-6
 - naming objects, 9-6
 - object types, 9-5, 9-6
 - pool objects
 - accessing, 9-43
 - creating, 9-42
 - definition of, 9-7
 - using, 9-42
 - programming restrictions, 9-36
 - region, 9-8
 - StreamAccess object, 9-7
 - subregion, 9-8
 - summary, 1-2
- Java Object Cache attributes
 - ALLOWNULL, 9-12
 - CacheEventListener, 9-13
 - CapacityPolicy, 9-13
 - DefaultTimeToLive, 9-13
 - DISTRIBUTE, 9-11, 9-44, 9-45
 - GROUP_TTL_DESTROY, 9-11
 - IdleTime, 9-13
 - LOADER, 9-11
 - maxCount, 9-14
 - MaxSize, 9-14
 - MEASURE, 9-13
 - ORIGINAL, 9-11
 - Priority, 9-14
 - REPLY, 9-12
 - SPOOL, 9-12
 - SYNCHRONIZE, 9-12
 - SYNCHRONIZE_DEFAULT, 9-12
 - TimeToLive, 9-14
 - User-defined, 9-14
 - Version, 9-14
- Java Transaction API
 - See* JTA
- Java-CORBA exception mapping, 6-11
- java.naming.factory.initial property, 2-4, 5-7
- java.naming.provider.url
 - JNDI-related environment properties, 2-5
 - property, 5-7, 6-13
- java.util.Hashtable
 - JNDI, 2-3
- javax.naming package, 2-2
- javax.naming.Context interface
 - JNDI, 2-3
- javax.sql.DataSource, 4-1
- JDBC
 - Oracle extensions, 4-21
 - transactions, 7-6
- JMD
 - default connection factories, 3-4
- JMS, 3-1
 - configuring custom resource provider, 3-23
 - configuring provider, 3-25
 - ConnectionFactory, 3-4
 - Destination, 3-26
 - example, where to download, 3-1
 - HiAvailability and clustering, 3-40
 - installing provider, 3-25
 - OracleAS, 3-2
 - overview
 - programming models, 3-1
 - queue connection factory, 3-3
 - QueueConnectionFactory, 3-4
 - receiving a message, JMS steps, 3-6
 - resource providers, 3-23
 - sending a message, JMS steps, 3-5
 - summary, 1-1

- system properties, 3-21
- topic connection factory, 3-3
- TopicConnectionFactory, 3-4
- XAConnectionFactory, 3-4
- XAQueueConnectionFactory, 3-5
- XATopicConnectionFactory, 3-5
- JMS provider
 - configuring, 3-25
 - installing, 3-25
- JMS utility
 - browse, 3-9
 - check, 3-9
 - copy, 3-9
 - destinations, 3-9
 - drain, 3-9
 - durables, 3-9
 - help, 3-9
 - knobs, 3-9
 - move, 3-9
 - stats, 3-9
 - subscribe, 3-9
 - unsubscribe, 3-9
- <jms-config> element, 3-2
- jms/ConnectionFactory, 3-4
- jms/QueueConnectionFactory, 3-4
- jms-server element, 3-16
- jms/TopicConnectionFactory, 3-4
- jms/XAConnectionFactoryXAConnectionFactory
 - JMS, 3-4
- jms/XAQueueConnectionFactory, 3-5
- jms/XATopicConnectionFactory, 3-5
- jms.xml, 3-2
 - modifying with Oracle Enterprise Manager, 3-2
 - persistent-file attribute, 3-11
- JNDI, 2-1
 - application-client.jar, 2-3, 2-5
 - application-client.xml, 2-5
 - clustering
 - enabling, 2-11
 - limitations, 2-11
 - overview, 2-11
 - com.evermind.server package, 2-4
 - constructing contexts, 2-2
 - environment, 2-3
 - example, servlet retrieves data source, 2-8
 - initial context, 2-1
 - initial context factories, 2-4
 - InitialContext constructors, 2-3
 - java.util.Hashtable, 2-3
 - javax.naming.Context interface, 2-3
 - jndi.properties file, 2-3
 - orion-application-client.xml, 2-5
 - overview, 2-1
 - summary, 1-1
- JNDI initial components
 - from J2EE application clients, 2-8
- JNDI initial contexts
 - from J2EE application clients, 2-4
- JNDI lookup
 - properties in orion-ejb-jar.xml, 7-3

- jndi.jar file, 2-1
- jndi.properties file, 5-7, 6-13
 - JNDI, 2-3
- JNDI-related environment properties, 2-5
 - context.SECURITY_CREDENTIAL, 2-6
 - context.SECURITY_PRINCIPAL, 2-6
 - dedicated.rmicontext, 2-5
 - http.tunnel.path, 2-5
 - java.naming.provider.url, 2-5
- JTA
 - bean-managed transaction, 7-1, 7-6
 - client-side transaction demarcation, 7-6
 - code download site, 7-1
 - configuring timeouts, 7-10
 - container-managed transaction, 7-1, 7-4
 - demarcation, 7-1, 7-4
 - deployment descriptor, 7-5
 - MDBs, 7-11, 7-12
 - programmatic transaction demarcation, 7-6
 - resource enlistment, 7-2
 - retrieving data source, 7-3
 - retry count, 7-11
 - single-phase commit
 - definition, 7-2
 - single-phase commit, configuring, 7-2
 - specification web site, 7-1
 - summary, 1-2
 - transaction attribute types, 7-5
 - transactions, 7-6
 - two-phase commit, 7-6
 - two-phase commit, configuration, 7-7
 - two-phase commit, definition, 7-2

K

- knobs
 - JMS utility, 3-9

L

- LOADER
 - Java Object Cache attribute, 9-11
- location
 - <data-source> attribute, 4-8
- locations
 - deployment directories, 8-9
 - J2EE Connector Architecture, 8-9
 - J2EE Connector Architecture files, 8-10
- log element, 3-19
- log() method, 9-18

M

- MAA, 4-26
- Mandatory transaction attribute type, 7-5
- max-connect-attempts
 - <data-source> attribute, 4-9
- max-connections
 - <data-source> attribute, 4-9
- maxCount
 - Java Object Cache attribute, 9-14

- maxObjects property, 9-23
- MaxSize
 - Java Object Cache attribute, 9-14
- maxSize property, 9-23
- max-tx-retries attribute, 7-11
- MDBs
 - and OJMS, 3-39
 - JTA, 7-11, 7-12
 - transaction timeout, 7-12
 - transactions, 7-11
 - transactions with OC4J JMS, 7-12
 - transactions with Oracle JMS, 7-12
 - with bean-managed transactions and JMS clients, 7-13
 - with container-managed transactions, 7-12
- MEASURE
 - Java Object Cache attribute, 9-13
- message
 - receiving in JMS, steps, 3-6
 - sending in JMS, steps, 3-5
- message expiration
 - OracleAS JMS, 3-15
- message paging
 - OracleAS JMS, 3-15
- message-driven beans, *See* MDBs
- <message-driven-deployment> element, 7-12
- min-connections
 - <data-source> attribute, 4-9
- move
 - JMS utility, 3-9

N

- name
 - <data-source> attribute, 4-8
- naming interoperability, 6-1
- naming objects, 9-6
- native data sources, 4-4
- netSearch() method, 9-18, 9-50
- network failover, 4-26
- Never transaction attribute type, 7-5
- nonemulated data sources, 4-4
 - object, behavior, 4-18
- NotSupported transaction attribute type, 7-5

O

- OBJECT_INVALIDATION event, 9-35
- OBJECT_UPDATED event, 9-35
- OC4J
 - configuring to support RMI tunneling, 5-11
- OC4J client JAR files, 5-2, 6-2
- OC4J JMS
 - abnormal termination, 3-14
 - persistence file management, 3-12
- OC4J mount point
 - configuring, 5-12
- OC4J sample code page, 3-1
- oc4j-connectors.xml, 8-6
- OC4J-hosted beans

- invoking from non-OC4J container, 6-11
- oc4j.jms.debug OracleAS JMS control knob, 3-22
- oc4j.jms.forceRecovery OracleAS JMS control knob, 3-23
- oc4j.jms.listenerAttempts OracleAS JMS control knob, 3-22
- oc4j.jms.maxOpenFiles OracleAS JMS control knob, 3-22
- oc4j.jms.messagePoll OracleAS JMS control knob, 3-22
- oc4j.jms.noDms OracleAS JMS control knob, 3-22
- oc4j.jms.pagingThreshold, 3-23
- oc4j.jms.saveAllExpired OracleAS JMS control knob, 3-22
- oc4j.jms.saveAllExpired property, 3-15
- oc4j.jms.serverPoll OracleAS JMS control knob, 3-22
- oc4j.jms.socketBufsize OracleAS JMS control knob, 3-22
- oc4j-ra.xml, 8-4
- OCI driver, 4-22
- OJMS
 - as resource provider, 3-24
 - configure resource provider with Enterprise Manager, 3-27
 - configuring resource provider, 3-28, 3-29
 - define resource provider, 3-27
 - resource provider, 3-24
 - using as a resource provider, 3-24
 - using as resource provider, 3-24
- OJMS configuring, 3-47
- OPMN URL, 6-10
- opmn.xml file
 - editing, 5-6
- Oracle Application Server Containers for J2EE (OC4J)
 - interoperability, 6-1
 - interoperability flags, 6-12
- Oracle Enterprise Manager
 - configuring JMS ports, 3-2
 - modifying jms.xml, 3-2
- Oracle JMS
 - configuring, 3-27
- Oracle JMS provider
 - configuring in OC4J XML files, 3-27
- Oracle JMS, *See* OJMS
- Oracle Maximum Availability Architecture, 4-26
- OracleAS JMS, 3-2
 - administration, 3-21
 - administration properties, table, 3-22
 - configuration elements hierarchical tree, 3-3
 - configuring, 3-2
 - control knob oc4j.jms.debug, 3-22
 - control knob oc4j.jms.forceRecovery, 3-23
 - control knob oc4j.jms.listenerAttempts, 3-22
 - control knob oc4j.jms.maxOpenFiles, 3-22
 - control knob oc4j.jms.messagePoll, 3-22
 - control knob oc4j.jms.noDms, 3-22
 - control knob oc4j.jms.saveAllExpired, 3-22
 - control knob oc4j.jms.serverPoll, 3-22
 - control knob oc4j.jms.socketBufsize, 3-22
 - exception queue, predefined, 3-14

- file-based persistence, 3-10
- message expiration, 3-15
- message paging, 3-15
- port,
 - configuring, 3-2
- predefined exception queue, 3-14
- utilities, 3-8
- utilities, table, 3-9
- OracleAS JMS configuring, 3-41
- OracleAS JMS ports
 - configuring, 3-2
- OracleAS Web Cache, 9-1
- oracleFatalError method, 7-11
- oracle.ias.cache package, 9-15
- oracle.j2ee.connector package
 - AbstractPrincipalMapping, 8-16
- OracleTwoPhaseCommitDriver, 7-9
- ORIGINAL
 - Java Object Cache attribute, 9-11
- orion-application-client.xml
 - JNDI, 2-5
- orion-application.xml file
 - and JNDI resource provider, 3-23
 - EAR file, 4-8
 - <resource-provider>, 3-37, 3-38, 3-39
- OrionCMTDataSource, 7-9
- orion-ejb-jar.xml file
 - <entity-deployment> element, 6-12
- orion-ejb-jar.xml file
 - <session-deployment> element, 6-12
 - , 7-3
 - <ior-security-config> element, 6-12
- orion-ejb-jar.xml file, 8-5
- ORMI, 5-1
- ORMI tunneling, 5-11
- OTS, 6-1
- overview of JMS
- overview of OC4J services, 1-1

P

- password
 - <data-source> attribute, 4-9
 - indirection, 4-12
 - obfuscation, 4-12
- persistence file management
 - OC4J JMS, 3-12
- persistent-file attribute, 3-11
- ping-interval configuration XML element, 9-23
- pluggable authentication classes, 8-18
- PoolAccess
 - close() method, 9-43
 - get() method, 9-43
 - getPool() method, 9-43
 - object, 9-43
 - returnToPool() method, 9-43
- PoolInstanceFactory
 - implementing, 9-43
- predefined exception queue
 - OracleAS JMS, 3-14

- Priority
 - Java Object Cache attribute, 9-14
- programmatic container-managed sign-on, 8-15
- programmatic transaction demarcation, 7-6
- programming models
 - JMS, 3-1
- PROVIDER_URL
 - InitialContext property, 2-3

Q

- QoS
 - see* Quality of Service
- Quality of Service
 - contracts, specifying, 8-10
 - J2CA types, 8-3
- queue connection factory
 - JMS, 3-3
- queue element, 3-17
- QueueConnectionFactory
 - JMS, 3-4
- queue-connection-factory element, 3-18

R

- RAC, 4-26
- rac-enabled
 - <data-source> attribute, 4-9
- RAR file, 8-2
- ra.xml file, 8-4
- receiving a message
 - JMS steps, 3-6
- release_Ownership() method, 9-50
- releaseOwnership() method, 9-47
- Remote Method Invocation
 - See* RMI
- REPLY
 - Java Object Cache attributes, 9-12
- REPLY attribute, 9-46
- Required transaction attribute type, 7-5
- RequiresNew transaction attribute type, 7-5
- resource adapter archive
 - see* RAR file
- resource adapters
 - deploying, 8-3
 - embedded, 8-2, 8-9
 - introduction, 8-1
 - standalone, 8-7
 - undeploying, 8-3
 - with admin.jar, 8-7
- resource provider
 - configuring with data sources property, 3-28
 - OJMS, 3-24
 - OJMS, configure with Enterprise Manager, 3-27
 - OJMS, define, 3-27
- resource providers
 - JMS, 3-23
- <resource-env-ref> element, 3-33
- <resource-provider> element, 3-27, 3-37, 3-38, 3-39
- ResourceProvider interface

- JMS, 3-23
- OJMS, 3-24
- <resource-ref> element, 3-33, 4-16
- <res-ref-name> element, 4-16
- returnToPool() method, 9-43
- RMI
 - IIOP, 6-1
 - introduction, 5-1
 - ORMI, 5-1
 - overview, 1-2, 5-1
- RMI tunneling
 - configuring OC4J to support, 5-11
- <rmi-config> element, 5-5
- RMI/IIOP
 - advanced interoperability in OracleAS environment, 6-8
 - advanced interoperability in standalone environment, 6-4
 - configuring for advanced interoperability manually, 6-8
 - configuring for advanced interoperability with Oracle Enterprise Manager, 6-8
 - configuring for simple interoperability manually, 6-6
 - configuring for simple interoperability with Oracle Enterprise Manager, 6-5
 - configuring OC4J mount point, 5-12
 - contextFactory property, 6-13
 - Java-CORBA exception mapping, 6-11
 - java.naming.factory.initial property, 5-7
 - java.naming.provider.url property, 5-7, 6-13
 - jndi.properties file, 5-7, 6-13
 - simple interoperability in OracleAS environment, 6-5
 - simple interoperability in standalone environment, 6-3
- RMIInitialContextFactory, 2-10
- <rmi-server> element, 5-5
- rmi.xml
 - editing, 5-5

S

- sample code page, OC4J, 3-1
- save() method, 9-39
- schema
 - <data-source> attribute, 4-9, 4-14
- security interoperability, 6-1
- SECURITY_CREDENTIAL
 - InitialContext property, 2-4
- SECURITY_PRINCIPAL
 - InitialContext property, 2-4
- sending a message
 - JMS steps, 3-5
- <sep-config> element, 5-7, 6-12
- server.xml
 - <sep-config> element, 5-7
- server.xml
 - and RMI, 5-5
 - configuring timeouts, 7-10
 - <sep-config> element, 6-12
 - server.xml
 - <application-server> element, 8-9
 - service provider interfaces, 2-1
 - <session-deployment> element, 6-12
 - setAttributes() method, 9-18
 - setCacheEventListener() method, 9-35
 - single-phase commit
 - configuring, 7-2
 - SPIs, 2-1
 - SPOOL
 - Java Object Cache attribute, 9-12, 9-38
 - SQLServer
 - data source entry with DataDirect, 4-24
 - standalone resource adapters, 8-2
 - stats
 - JMS utility, 3-9
 - stmt-cache-size
 - <data-source> attribute, 4-9
 - StreamAccess object, 9-7
 - InputStream, 9-41
 - OutputStream, 9-41
 - using, 9-41
 - Streams Advanced Queuing (AQ), 3-24
 - subscribe
 - JMS utility, 3-9
 - Supports transaction attribute type, 7-5
 - SYNCHRONIZE
 - Java Object Cache attribute, 9-12, 9-47
 - SYNCHRONIZE_DEFAULT
 - Java Object Cache attribute, 9-12, 9-47

T

- TAF
 - configuration options, 4-29
 - configuring, 4-28
 - descriptor, 4-29
 - exceptions, 4-31
- timeouts
 - configuring, JTA, 7-10
- TimeToLive
 - Java Object Cache attribute, 9-14
- topic connection factory
 - JMS, 3-3
- topic element, 3-18
- TopicConnectionFactory
 - JMS, 3-4
- topic-connection-factory element, 3-18
- transaction
 - bean managed, 7-1
 - container-managed, 7-1
 - demarcation, 7-1, 7-4
 - resource enlistment, 7-1, 7-2
 - two-phase commit, 7-7
- transaction attribute types, 7-5
- transaction demarcation
 - client-side, JTA, 7-6
 - programmatic, JTA, 7-6
- transaction interoperability, 6-1

- transactions
 - JDBC, 7-6
 - JTA, 7-6
 - MDBs, 7-11
 - MDBs with OC4J JMS, 7-12
 - MDBs with Oracle JMS, 7-12
- transaction-timeout attribute, 7-12
- <transaction-type> element, 7-4, 7-6
- trans-attribute attribute, 7-11
- <trans-attribute> element, 7-4, 7-5
- Transparent Application Failover
 - See TAF
- transport interoperability, 6-1
- tunneling
 - ORMI, 5-11
- two-phase commit
 - data sources, 4-19
 - definition, 7-2
 - engine limitations, 7-10
 - OracleTwoPhaseCommitDriver, 7-9
 - overview, 7-6
- tx-retry-wait attribute, 7-11
- type-mapping, 4-14

U

- unsubscribe
 - JMS utility, 3-9
- URL
 - <data-source> attribute, 4-9
 - corbaname, 6-10
 - OPMN, 6-10
- User-defined
 - Java Object Cache attribute, 9-14
- username
 - <data-source> attribute, 4-8
- using resource provider
 - OJMS, 3-24
- utilities
 - OracleAS JMS, 3-8
 - OracleAS JMS, table, 3-9

V

- Version
 - Java Object Cache attribute, 9-14

W

- wait-timeout
 - <data-source> attribute, 4-9
- Web Cache, 9-1
- Web Object Cache, 9-1, 9-2

X

- xa-connection-factory element, 3-18
- xa-location
 - <data-source> attribute, 4-9
- XAQueueConnectionFactory
 - JMS, 3-5