Oracle® Application Server Containers for J2EE

Services Guide

10*g* (9.0.4)

Part No. B10326-01

September 2003

ORACLE®

Oracle Application Server Containers for J2EE Services Guide, 10*g* (9.0.4)

Part No. B10326-01

Primary Authors: Peter Purich, Elizabeth Hanes Perry

Contributing Authors: Janis Greenberg, Mark Kennedy

Contributors: Anirruddha Thakur, Anthony Lai, Ashok Banerjee, Brian Wright, Cheuk Chau, Debabrata Panda, Ellen Barnes, 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 Kavantzas, Rachel Chan, Rajkumar Irudayaraj, Raymond Ng, Sastry Malladi, Sheryl Maring, Stella Li, Sunil Kunisetty, Thomas Van Raalte.

Contents

3 Java Message Service

4 Data Sources

5 Oracle Remote Method Invocation

6 J2EE Interoperability

7 Java Transaction API

8 J2EE Connector Architecture

9 Java Object Cache

Index

List of Examples

List of Figures

List of Tables

Send Us Your Comments

Oracle Application Server Containers for J2EE Services Guide, 10*g* (9.0.4)

Part No. B10326-01

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

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

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

- Electronic mail: 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 (optionally) electronic mail address.

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

Preface

Oracle Application Server 10*g* (9.0.4) 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:

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

Audience

This book was written for developers familiar with the J2EE architecture who want to understand Oracle's implementation of 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. Standards will continue to evolve over time, and Oracle Corporation is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at:

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

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

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

Organization

This document contains the following chapters:

Chapter 1, "Introduction to OC4J Services"
Gives an overview of the service technologies included in OC4J.

Chapter 2, "Java Naming and Directory Interface"
Covers using the JNDI to look up objects.

Chapter 3, "Java Message Service"
Discusses plugging Resource Providers into the Java Message Service (JMS) and the two JMS providers of its own that Oracle furnishes.

Chapter 4, "Data Sources"
Discusses data sources, vendor-independent encapsulations of a connection to a database server.

Chapter 5, "Oracle Remote Method Invocation"
Describes OC4J support for Remote Method Invocation (RMI) over the proprietary Oracle RMI (ORMI) protocol.

Chapter 6, "J2EE Interoperability"
Describes OC4J support for EJB2.0 interoperation using RMI over the standard Internet Inter-Orb Protocol (IIOP) protocol.

Chapter 7, "Java Transaction API"
Documents Oracle's implementation of the JTA.

Chapter 8, "J2EE Connector Architecture"
Describes how to use the J2EE Connector Architecture in an OC4J application.

Chapter 9, "Java Object Cache"
Details the OC4J Java Object Cache, including its architecture and programming features.

Related Documentation

Refer to the following additional OC4J documents that are available from the Oracle Java Platform Group:

- *Oracle Application Server Containers for J2EE User's 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 Developer's 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 Developer's 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 JavaBeans Developer's 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 Administrator's Guide*
- *Oracle Enterprise Manager Administrator's Guide*
- *Oracle HTTP Server Administrator's Guide*
- *Oracle Application Server 10g Performance Guide*
- *Oracle Application Server 10g Globalization Guide*
- *Oracle Application Server Web Cache Administrator's Guide*
- *Oracle Application Server 10g Upgrading to 10g (9.0.4)*

The following are available from the JDeveloper group:

- Oracle JDeveloper online help
- Oracle JDeveloper documentation on the Oracle Technology Network:

  `http://otn.oracle.com/products/jdev/content.html`

For information about Oracle Application Server 10*g* Personalization, which is the foundation of the Personalization tag library, refer to the following documents from the Oracle Application Server 10*g* Personalization group:

- *Oracle Application Server Personalization Administrator's Guide*
- *Oracle Application Server Personalization API Reference*

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

- OTN Web site for OC4J:

  `http://otn.oracle.com/tech/java/oc4j/content.html`

- OTN OC4J discussion forums, accessible through the following address:

  `http://www.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, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

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

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://otn.oracle.com/documentation/content.html`

Conventions

The following conventions are 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 GUI component such as a link or button to click.
Italics	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.
[ ]	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.

1

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 (JCA)
- 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 Oracle Application Server Containers for J2EE (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", 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 both RMI over the Oracle Remote Method Invocation (ORMI) protocol and over the Internet Inter-ORB Protocol (IIOP) protocol.

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) protocol 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's 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 (JCA)

J2EE Connector Architecture (JCA) 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", 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 that are 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.

2

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
- JNDI Clustering

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.

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`, *looking up* a J2EE or other resource

When OC4J starts up, it constructs a JNDI initial context *for each application* by reading each of the application's configuration XML files that can contain resource references.

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.

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 application's JNDI tree. 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*.

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. ("Application Client Example" on page 2-8 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, the initial context is created by OC4J when the server is started, using the default environment for that application. This constructor is the one that 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. The second form of the `InitialContext` constructor is normally used 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 that are required by JNDI. These properties, defined in the `javax.naming.Context` interface, are listed in Table 2–1.

Table 2–1 InitialContext Properties

Property	Meaning
`INITIAL_CONTEXT_FACTORY`	Value for the `java.naming.factory.initial` property. This property specifies which initial context factory to use when creating a new initial context object.

Table 2–1 InitialContext Properties

Property	Meaning
PROVIDER_URL	Value for the java.naming.provider.url property. This property specifies the URL that the application client code uses to look up objects on the server. Also used by RMIInitialContextFactory and ApplicationClientInitialContextFactory to search for objects in different applications. See Table 2–2, "JNDI-Related Environment Properties" on page 2-7 for details.
SECURITY_PRINCIPAL	Value for the java.naming.security.principal 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 java.naming.security.credential 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 "Application Client Example" on page 2-8 for a code example that sets these properties and gets a new JNDI initial context.

Creating the Initial Context in OC4J

Application clients are defined in section 9.1 of the J2EE 1.3 specification as "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]."

JNDI initial contexts can be used in the following ways:

- 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:** In deciding which JNDI initial context factory to use, 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) being used by the client application. The protocol itself is specified by the JNDI property `java.naming.provider.url`. See Table 2–2, "JNDI-Related Environment Properties" on page 2-7 for details.

Consider an application client that consists of Java code running outside the OC4J server, but that is 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 done in client code, or it can be specified in the `jndi.properties` file that is part of the application's `application-client.jar` file that is 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 that are 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 have to be defined in deployment descriptors in order to be bound to an application's JNDI context.

Environment Properties

`ApplicationClientInitialContextFactory` reads the properties shown in Table 2–2 from the environment if the ORMI protocol is being used.

Table 2–2 JNDI-Related Environment Properties

Property	Meaning
`dedicated.rmicontext`	This property replaces the deprecated `dedicated.connection` setting. When two or more clients in the same process retrieve an `InitialContext`, OC4J returns a cached context. Thus, each client receives the same `InitialContext`, which is assigned to the process. Server lookup, which results in server load balancing, happens only if the client retrieves its own `InitialContext`. If you set `dedicated.rmicontext=true`, then each client receives its own `InitialContext` instead of a shared context. When each client has its own `InitialContext`, then the clients can be load balanced. The `dedicated.rmicontext` property defaults to `false`.
`java.naming.provider.url`	This property specifies the URL to use when looking for local or remote objects. The format is either `[http: | https:]ormi://`*`hostname`*`/`*`appname`* or *`corbaname`*`:`*`hostname`*`:`*`port`*. For details on the *`corbaname`* URL, see "The corbaname URL" on page 6-13. Multiple hosts (for failover) can be supplied in a comma-separated list.
`java.naming.factory.url.pkgs`	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.
`http.tunnel.path`	This property specifies an alternative `RMIHttpTunnelServlet` path. The default path is `/servlet/rmi`, as bound to the target site's Web application. For more information, see "Configuring ORMI Tunneling through HTTP" on page 5-16.
`Context.SECURITY_PRINCIPAL`	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 `java.naming.security.principal`.

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

Property	Meaning
`Context.SECURITY_CREDENTIAL`	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 `java.naming.security.credentials`.

Application Client Example

This is an example of how to configure an application client to access an EJB running inside an OC4J instance in the same machine.

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, we can pass a hash table to the constructor of `InitialContext` instead of supplying a `jndi.properties` file. The code would look 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);
```

Since the application client code refers to the `EmployeeBean` EJB, this must be declared 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. The EJB name used in the application client program must be mapped to the JNDI location where the EJB is actually bound to. This is done 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

Initial context factories can be used 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

To access objects in the same application from servlets, JSP pages, and EJB, you can use J2EE application components.

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 that are 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 virtual machine.

If your application must look up a remote reference, either a resource in another J2EE application in the same JVM or perhaps a resource external to any J2EE application, then you must use RMIInitialContextFactory or IIOPInitialContextFactory. Also see "Objects Not in the Same Application" on page 2-13.

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

The data source location is specified in `data-sources.xml` as:

```
<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's `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

You can access objects not in the same application using one of these context factories:

- RMIInitialContextFactory
- IIOPInitialContextFactory

RMIInitialContextFactory Using either the default server-side `ApplicationInitialContextFactory` or specifying `ApplicationClientInitialContextFactory` works for most application purposes. In some cases, however, an additional context factory must be used:

- 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 same environment properties (described in detail in Table 2–2 on page 2-7) used by `ApplicationClientInitialContextFactory`:

- `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 by the "Application Client Example" on page 2-8.

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, `<ejb-ref>` elements must be declared 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>
```

Also, a mapping from the logical name `EmployeeBean` to the actual JNDI name where the EJB is bound must be provided in `orion-web.xml`:

```
<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 Clustering

JNDI clustering 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 clustering is enabled, you can bind a serializable value into an application context (via 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 Clustering
- JNDI Clustering Limitations

Enabling JNDI Clustering

JNDI clustering is enabled when EJB clustering is enabled.

To take advantage of JNDI clustering, 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 Clustering Limitations

Consider the following limitations when relying on JNDI clustering:

- Multiple Islands on a Given Subnet
- Propagating Changes Across the Cluster
- Binding a Remote Object

Multiple Islands on a Given Subnet

As described in the clustering chapter in the *Oracle Application Server Containers for J2EE User's Guide*, 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 sub-grouping.

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

Propagating Changes Across the Cluster

Re-binding (re-naming) and unbinding are not propagated: they apply locally but are not shared across the cluster.

Bindings to values which 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, that JNDI object will be shared across the cluster but there will be a single point of failure if the first server it is bound to fails.

3

Java Message Service

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 at `http://otn.oracle.com/tech/java/oc4j/demos` on the OTN Web site.

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 a number of 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, SwiftMQ.

Oracle Application Server JMS

Oracle Application Server JMS (OracleAS JMS) is a Java Message Service that provides the following features:

- Compliant with the JMS 1.0.2b specification.
- Choice between in-memory or file-based message persistence.
- Provides an exception queue for undeliverable messages.

This section covers the following:

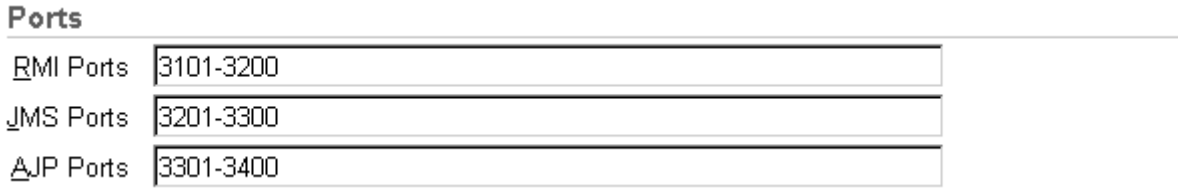
- Configuring OracleAS JMS Ports
- Configuring OracleAS JMS Destination Objects
- Steps for Sending and Receiving a Message

- 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

Figure 3–1 demonstrates how you can configure the port range that OracleAS JMS uses within the Oracle Enterprise Manager The default range is between 3201 and 3300. From the OC4J Home page, select the Administration page. Select Server Properties in the instance Properties column. Then scroll down to the Multiple VM Configuration section.

Figure 3–1 JMS Ports



Configuring OracleAS JMS Destination Objects

OracleAS JMS `Destination` objects, which can be either queues or topics, are configured in the `jms.xml` file. 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.

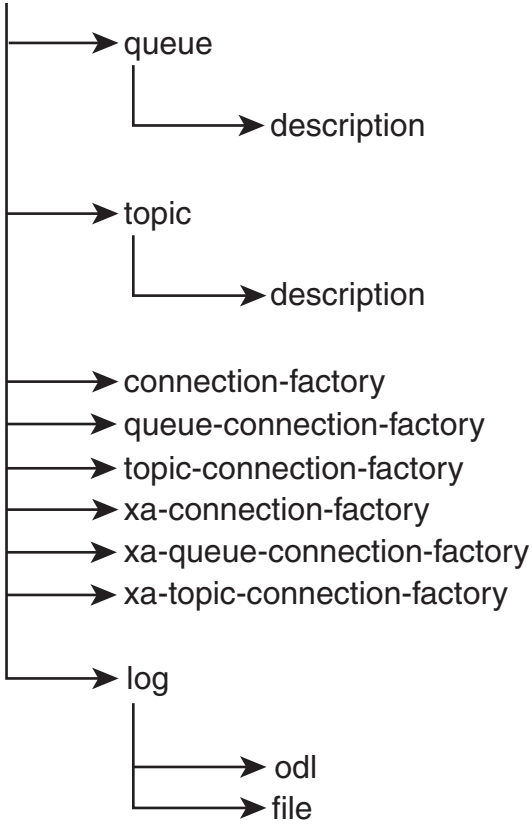
- Oracle Enterprise Manager configuration—To edit the `jms.xml` file directly through Oracle Enterprise Manager, select Advanced Properties under the Instance Properties column on the Administration page. In this section, choose `jms.xml` to modify the straight XML file.

- 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 3–2 demonstrates the order in which the elements in the `jms.xml` file are structured. "OracleAS JMS Configuration File Elements for jms.xml" on page 3-22 provides a complete description of all elements and their attributes in the `jms.xml` file.

Figure 3–2 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 that is 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/dtds
/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: All of these values are defaults, so you do not have to configure them. However, the configuration for the queue, topic, and their connection factories are shown so you understand how to configure your own `Destination` objects and connection factories.

See "OracleAS JMS Configuration File Elements for jms.xml" on page 3-22 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 needing to add 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 ranging over the XA/non-XA and various JMS domains. You can use these connection factories in your code without needing to add 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 non-default 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 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 and Receiving a Message

A JMS client sends or 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.

If you are sending messages, then do the following:

5. Providing 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.

However, if you are receiving messages, then do the following:

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 3–1 demonstrates these steps for sending a JMS message; 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 at `http://otn.oracle.com/tech/java/oc4j/demos` on the OTN Web site.

Note: For simplicity, most of the error code is removed in Example 3–1 and Example 3–2. 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 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 3-46 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 following method—`dosend`—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();

}
```

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

The following method—`dorcv`—sets up a queue to receive messages off of it. After creating the queue receiver, it loops to receive all messages off of 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`, that is used for debugging and information access.

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

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

The OracleAS JMS server must be running. 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
`-host <hostname>`	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.
`-port <port>`	The (remote) port where the OracleAS JMS server is installed. The default JMS port number is 9127.
`-username <username>`	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.
`-password <password>`	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.
`-clientID <ID>`	Use this identifier for all JMS connections. This is only required 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 preceding command with no argument. To display extensive information about the set of commands that are available, the arguments' 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 `-selector` 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 `<other-selector>`.
`knobs`	Display all available system properties (shown in Table 3–5) and their current settings on the OC4J JMS server.

Table 3–2 OC4J JMS Utilities (Cont.)

Utility	Command Description
`stats`	Display all available DMS statistics on the OC4J JMS server (this will include non-JMS statistics as well). (For information on DMS, see the *Oracle Application Server 10g Performance Guide*.)
`destinations`	Print a list of all permanent `Destination` 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 `<topic>`, with a specified name, message selector, whether it is local or not, and a durable subscription client identifier. This replaces existing, inactive durable subscriptions. The name is identified with the `-name` command option. The message selector is identified by the `-selector` command option. Whether the durable subscription is local or not is identified by the `-noLocal` command option. The client identifier is defined with the `-clientID` generic option.
`unsubscribe`	Drop an existing, inactive durable subscription. The durable subscription is identified by a name (`-name` command option) and the client identifier (`-clientID` generic option).
`browse <destination>`	Browse messages on a given destination (queue or topic durable subscription, defined in `jms.xml`).
`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
`-selector <selector>`	Create queue receivers and durable subscribers with the specified JMS message selector.

Table 3–3 JMSUtils Command Options

Command Option	Description
`-noLocal [true|false]`	If set to be true, the subscriber will not see the messages which are published in the same connection. Use when creating a durable subscriber. Default value is FALSE.
`-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 indicated number of messages during the current operation. If the count is negative or zero, all selected messages are processed.

An example for 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). File-based persistence is discussed more in the following sections:

- 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.

Warning: 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, 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 3-22 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 default path is *`J2EE_HOME`*`/persistence/<island>` for Oracle Application Server environments and *`J2EE_HOME`*`/persistence` for standalone environments.

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 allows the possibility that your persistence files 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 that is 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 that are 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 will lead to 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, 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 other's 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 3-15 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 will be cleaned up automatically. However, if OC4J is terminated abnormally, the lock files continue to exist in the file system. Since OC4J cannot distinguish left-over 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 3-19.

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 once recovery is successful, moves to using the new configuration that is specified.

If recovery of the old configuration fails, 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, it creates a new one.

If, for some reason, the `jms.state` file itself is corrupted, 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, OC4J JMS does its best 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 3–5) 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, 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 attempt even to create the relevant persistent JMS `Destination` objects if it detects already existing lock files for them.

The default location of the lock files are 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 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.

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 due to 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 that was sent to a persistent Destination expires, 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 3–5).

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 move all expired messages, regardless of whether or not they are sent to persistent, nonpersistent, or temporary `Destination` objects, by setting the `oc4j.jms.saveAllExpired` administration property (described in Table 3–5) 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 3-14).
- 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 will begin 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).

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 will try to page out as many message bodies as it possibly can 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 will automatically page in the message body (regardless of the memory usage in the JVM) and deliver the correct message header/body to the client. Once 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, the OracleAS JMS server will stop paging out message bodies. The bodies of messages already paged out will not be 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>
```

The JMS configuration elements are defined below:

`jms-server`

The root element of the OC4J JMS server configuration.

Attributes:

- `host`—The host name defined in a `String` (DNS or dot-notation host name) that this OC4J JMS server should bind to. By default, the JMS server binds to `0.0.0.0` (also known as `[ALL]` in the configuration file). Optional attribute.
- `port`—The port defined as an `int` (valid TCP/IP port number) to which this OC4J JMS server should bind. The default setting is 9127. 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. Optional attribute.

`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.

Attributes:

- `name`—This required attributes is the provider-specific name (`String`) 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 `Session.createQueue()` 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.

- `location`—This required attribute states the JNDI location (`String`) 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 `java:comp/env/` JNDI name space, and should be appropriately declared in the relevant deployment descriptors. The `java:comp/env/` 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 `Destination` objects and connection factory elements in `jms.xml`. There is no default.
- `persistence-file`—An optional path and filename (`String`). 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 default path is *`J2EE_HOME`*`/persistence/<island>` for Oracle Application Server environments and *`J2EE_HOME`*`/persistence` for standalone environments.

  See "Recovery" on page 3-16 for more details on the meaning of this attribute. If multiple queue elements with the same name and different locations are declared in `jms.xml`, then all of them should specify the same value for `persistence-file` or should not specify the value at all—if at least one of these multiple declarations specifies a `persistence-file`, that value is used for this queue.

`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.

Attributes:

- `name`—This required attributes is the provider-specific name (`String`) 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 `Session.createTopic()` 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.
- `location`—This required attribute states the JNDI location (`String`) 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 `java:comp/env/` JNDI name space, and should be appropriately declared in the relevant deployment descriptors.

The `java:comp/env/` 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 `jms.xml`. There is no default.

- `persistence-file`—An optional path and filename (`String`). 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 default path is *`J2EE_HOME`*`/persistence/<island>` for Oracle Application Server environments and *`J2EE_HOME`*`/persistence` for standalone environments.

  See "Recovery" on page 3-16 for more details on the meaning of this attribute. If multiple queue or topic elements with the same name and different locations are declared in `jms.xml`, then all of them should specify the same value for `persistence-file` or should not specify the value at all—if at least one of these multiple declarations specifies a `persistence-file`, that value is used for this topic.

`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 3–4 describes all of the attributes for this element.

`queue-connection-factory`

JMS domain connection factory configuration. Table 3–4 describes all of the attributes for this element.

`topic-connection-factory`

JMS domain connection factory configuration.Table 3–4 describes all of the attributes for this element.

`xa-connection-factory`

XA variants of connection factory configuration. Table 3–4 describes all of the attributes for this element.

`xa-queue-connection-factory`

XA variants of connection factory configuration. Table 3–4 describes all of the attributes for this element.

`xa-topic-connection-factory`

XA variants of connection factory configuration. Table 3–4 describes all of the attributes for this element.

`log`

Log configuration element. Enables logging of the JMS activity in either file or ODL format. See the "Enabling OC4J Logging" section in the *Oracle Application Server Containers for J2EE User's Guide* for complete information on logging.

Table 3–4 describes all of the attributes for any connection factory definition.

Table 3–4 Connection Factory Configuration Attributes

Attribute	Type	Mandatory?	Default	Description
`location`	String	yes	(n/a)	The JNDI location that the connection factory is bound to. 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 `java:comp/env/` JNDI name space, and should be appropriately declared in the relevant deployment descriptors. The `java:comp/env/` 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 `Destination` and connection factory elements in `jms.xml`.
`host`	String (DNS or dot notation host name)	no	`[ALL]`	The fixed OC4J JMS host this connection factory will connect to. By default, a connection factory uses the same host as configured for the jms-server element. Nondefault values can be used to force all JMS operations to be directed to a specific OC4J Java virtual machine (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 that this connection factory connects to. By default, a connection factory uses the same port as configured for the `jms-server` 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 3–4 Connection Factory Configuration Attributes (Cont.)

Attribute	Type	Mandatory?	Default	Description
`username`	String	no	(the empty string)	The user name for the authentication of JMS default connections that is 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 that are 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 that are created from this connection factory. If no `clientID` is specified, the default is an empty string, which can also be programmatically overridden by client programs, as per 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 3–4, the property `password` supports password indirection. For more information, refer to the *Oracle Application Server Containers for J2EE Security Guide*.

Examples

The following code samples show connection factory configuration fragments:

The following configures a connection factory of `jms/Cf`, a queue connection factory of `jms/Qcf`, 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="jms/xaTcf"
    username="foo" password="bar" clientID="baz">
</xa-topic-connection-factory>
```

If you wanted to add a topic connection factory, you must use a unique name. For example, you could not name it with 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 samples show 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>
```

There are certain locations that are reserved and cannot be redefined within the `jms.xml` configuration file. The following shows how you cannot use the `jms/Oc4jJmsExceptionQueue` when defining a queue location, as 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 simply 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 their 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 affects basic JMS functionality—they pertain to OC4J JMS specific features, extensions, and performance optimizations.

Table 3–5 provides a brief summary of these administration properties.

Table 3–5 OC4J JMS Administration Properties

JVM System Property	Property Type	Default Value	Server/ Client	Use
`oc4j.jms.serverPoll`	long	`15000`	JMS client	Interval (in milliseconds) that JMS connections ping the OC4J server and report communication exceptions to exception listeners.
`oc4j.jms.messagePoll`	long	`1000`	JMS client	Maximum interval (in milliseconds) that JMS asynchronous consumers wait before checking the OC4J JMS server for new messages.
`oc4j.jms.listenerAttempts`	int	`5`	JMS client	Number of listener delivery attempts, before the message is declared undeliverable.
`oc4j.jms.maxOpenFiles`	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 `Destination` objects than the maximum number of open file descriptors that are allowed by the operating system.
`oc4j.jms.saveAllExpired`	boolean	`false`	OC4J server	Save all expired messages on all `Destination` objects (persistent, nonpersistent, and temporary) to the OC4J JMS exception queue.

Table 3–5 OC4J JMS Administration Properties (Cont.)

JVM System Property	Property Type	Default Value	Server/ Client	Use
`oc4j.jms.socketBufsize`	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.
`oc4j.jms.debug`	boolean	`false`	JMS client	If `true`, enable tracing of `NORMAL` events in JMS clients and the OC4J JMS server. All log events (`NORMAL`, `WARNING`, `ERROR`, and `CRITICAL`) are sent to both `stderr` and, when possible, either *`J2EE_HOME`*`/log/server.log` or *`J2EE_HOME`*`/log/jms.log`. Setting this property to `true` typically generates large amounts of tracing information.
`oc4j.jms.noDms`	boolean	`false`	JMS client	If `true`, disable instrumentation.

Table 3–5 OC4J JMS Administration Properties (Cont.)

JVM System Property	Property Type	Default Value	Server/ Client	Use
`oc4j.jms.forceRecovery`	boolean	`false`	OC4J server	If `true`, 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 `Destination` objects available for use.
`oc4j.jms.pagingThreshold`	double value	1.0	OC4J server	Represents the memory usage fraction above which the OracleAS JMS server will begin 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 3-21 for more information.
`oc4j.jms.usePersistenceLockFiles`	boolean	true	OC4J server	Control whether lock files should be used to protect against 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 false does not create lock files for persistent destinations. Set this property to false 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 3-33 for more information on Oracle JMS.

Configuring a Custom Resource Provider

A custom resource provider can be configured 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 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's JNDI as `"java:comp/resource/JNDIname/"`.

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

- `description` sub-element—A description of the specific resource provider.
- `property` sub-element—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 3-33 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)*.

Oracle JMS is fully described in the following sections:

- 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 3-34.
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 3-34.
3. Create the JMS `Destination` objects in OJMS. "Create JMS Destination Objects" on page 3-35.
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 3-36.
5. Access the resource in your implementation through a JNDI lookup. See "Access the OJMS Resources" on page 3-41.

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. Once you have installed and configured this JMS provider, you must apply additional configuration. This includes the following:

1. You or your DBA should 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 3-34.
2. You or your DBA should create the tables and queues to support the JMS `Destination` objects. See "Create JMS Destination Objects" on page 3-35.

> **Note:** The following sections use SQL for creating queues, topics, their tables, and assigning privileges that is provided within the JMS demo on the OC4J sample code page at `http://otn.oracle.com/tech/java/oc4j/demos` on the OTN Web site.

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 the JTA chapter for the 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 at `http://otn.oracle.com/tech/java/oc4j/demos` on the OTN Web site.

---

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 denotes whether there are multiple consumers or not; 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 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 wanted 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 application's deployment descriptors.

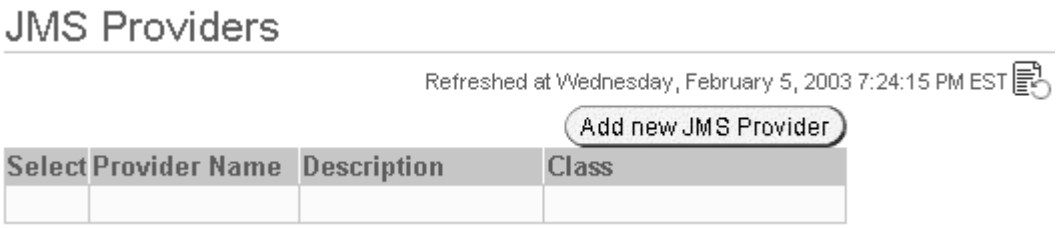
Define the OJMS Resource Provider

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

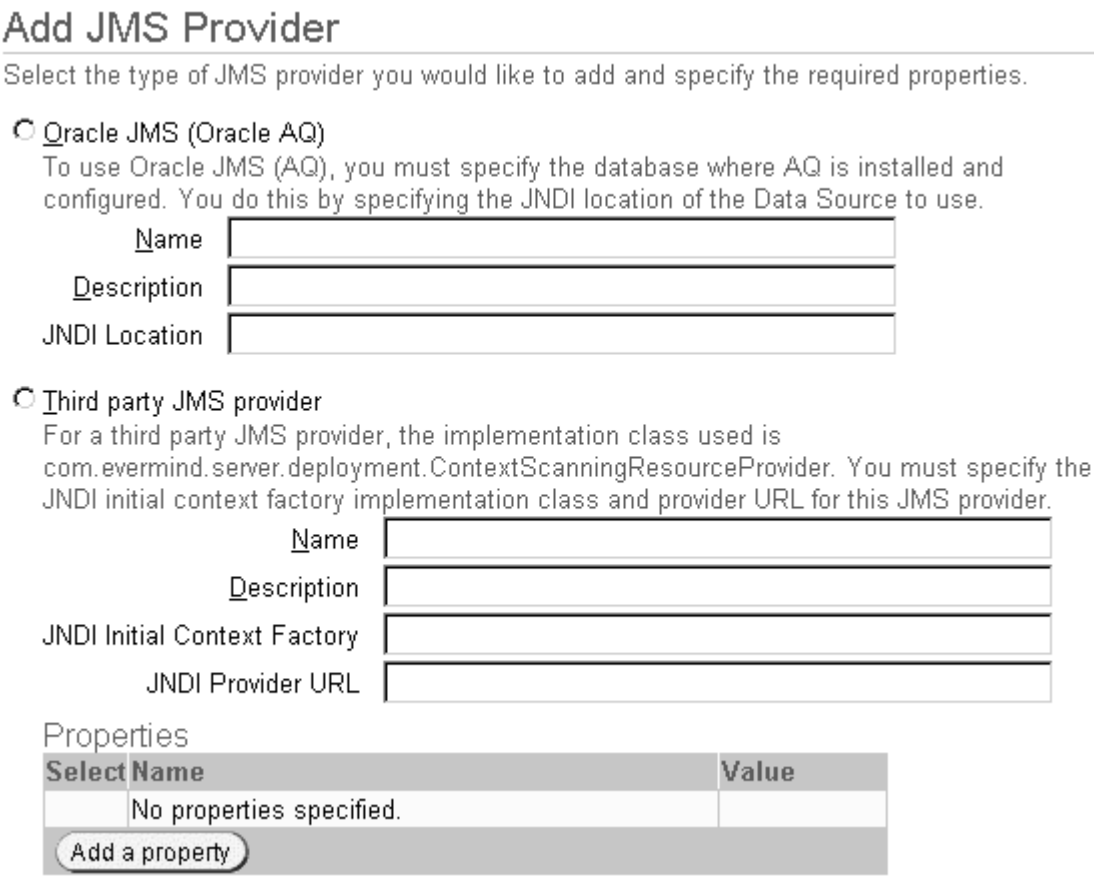
- Configure the OJMS Provider Through the Oracle Enterprise Manager
- Configure the OJMS Provider in the OC4J XML Files

Configure the OJMS Provider Through the Oracle Enterprise Manager The OJMS provider can be configured using Application Server Console in the JMS section. 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, which brings up the following page:



This page enables you to 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, as both OJMS and third-party providers are configured in the same manner.

Once you choose the type of JMS provider, you must provide the following:

- OJMS: Provide the data source name and JNDI location for the database where OJMS is installed and configured.

- Third-party JMS provider: Provide 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 where you can add the name for each JNDI property and its value.

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

To configure a JMS provider that is 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 identifies 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 where 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 non-emulated data sources. For two-phase commit transaction support, you can use only a non-emulated data source. See the JTA chapter for more information.

Example 3–3 Emulated DataSource 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 non-emulated data source. For differences between emulated and non-emulated data sources, see "Defining Data Sources" on page 4-8.

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.

```
<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:@myhost.foo.com:1521:mydb"
/>
```

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 4-8.

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:@hostname:port number:SID">
  </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 and Receiving a Message" on page 3-8. 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: Since 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: Since 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 as "`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 as "`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 at `http://otn.oracle.com/tech/java/oc4j/demos` on the OTN Web site.

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 following method—dosend—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 delineated in "Steps for Sending and Receiving a Message" on page 3-8.

```
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 of a Queue

The following method—`dorcv`—sets up a queue to receive messages off of it. After creating the queue receiver, it loops to receive all messages off of the queue and compares it to the number of expected messages. The steps necessary for setting up the queue and receiving messages are delineated in "Steps for Sending and Receiving a Message" on page 3-8.

```
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 condition 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 vice versa, under the assumption that they are interchangeable. The confusion is due to the Oracle Application Server and the Oracle database both shipping the OJMS client JAR file. Do not copy this file. Use the matrix in Table 3–6 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 3–6 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: 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 3–6 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+
9.0.2	X	X		X	
9.0.3			X		X
10*g* (9.0.4)			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 3-33 used explicit names within the JNDI lookup calls. This section describes how you can use logical names in your client application; thus, 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.

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

- The connection factory is identified in the client's XML deployment descriptor file within a `<resource-ref>` element.
  - The logical name that you want the connection factory to be identified as is defined in the `<res-ref-name>` element.
  - The connection factory class type is defined in the `<res-type>` element as either `javax.jms.QueueConnectionFactory` or `javax.jms.TopicConnectionFactory`.
  - The authentication responsibility (`Container` or `Bean`) is defined in the `<res-auth>` element.
  - The sharing scope (`Shareable` or `Unshareable`) is defined in the `<res-sharing-scope>` element.
- The JMS `Destination`—the topic or queue—is identified in a `<resource-env-ref>` element.
  - The logical name that you want the topic or queue to be identified as is defined in the `<resource-env-ref-name>` element.
  - The `Destination` class type is defined in the `<resource-env-ref-type>` element as either `javax.jms.Queue` or `javax.jms.Topic`.

The following shows an example of 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>
```

Then, 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 in OJMS. However, the mapping is defined in one of the following files:

- For a standalone Java client—the `orion-application-client.xml`
- For an EJB acting as a client—the `orion-ejb-jar.xml`

- For JSPs and servlets acting as a client—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 name for the OracleAS JMS Destination and connection factory are defined within the `jms.xml` file. As shown in Example 3–1, 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/`" and you have the mapping in the `orion-ejb-jar.xml` file as follows:

```
<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 3-41.

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 would access a JMS `Destination` object by providing the following 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
```

Where:

- 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 would access a JMS `Destination` object by providing the following 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
```

Where:

- 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

Once the resources have been defined and the JNDI properties configured, the client sends a JMS message by doing the following:

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 out 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 Message to a Topic

The method of sending a message to a topic is almost the same. 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 should be mapped in the OC4J deployment descriptor.

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

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

//1b. Lookup 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 that is 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 that are supplied by the vendor. (See the documentation that is supplied with your software for instructions.)
2. Configure the resource provider. You can either configure the resource provider through Oracle Enterprise Manager (as shown in "Define the OJMS Resource Provider" on page 3-36) or configure the `<resource-provider>` element in `orion-application.xml`. Use either method to add WebSphere MQ as a

custom resource provider. The following demonstrates an example of configuring WebSphere MQ through the `<resource-provider>` element. You could use the same information to configure through Oracle Enterprise Manager.

```
<resource-provider

class="com.evermind.server.deployment.ContextScanningResourceProvide
r"
      name="MQSeries">
  <description> MQSeries resource provider </description>
  <property
      name="java.naming.factory.initial"
      value="com.sun.jndi.fscontext.RefFSContextFactory">
  </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 either configure the resource provider through Oracle Enterprise Manager (as shown in "Define the OJMS Resource Provider" on page 3-36) or configure the `<resource-provider>` element in `orion-application.xml`. Use either method to add SonicMQ as a custom resource provider as the message provider and the file system as the JNDI store. The following demonstrates an example of configuring SonicMQ through the `<resource-provider>` element. You could use the same information to configure through Oracle Enterprise Manager.

```
<resource-provider

class="com.evermind.server.deloyment.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 that are supplied by the vendor. (See the documentation that is supplied with your software for instructions.)
2. Configure the resource provider. You can either configure the resource provider through Oracle Enterprise Manager (as shown in "Define the OJMS Resource Provider" on page 3-36) or configure the `<resource-provider>` element in `orion-application.xml`. Use either method to add SwiftMQ as a custom resource provider. The following demonstrates an example of configuring SwiftMQ through the `<resource-provider>` element. You could use the same information to configure through Oracle Enterprise Manager.

   ```
   <resource-provider

   class="com.evermind.server.deloyment.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 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.

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, a combination of software, hardware and infrastructure mechanisms make sure that another instance of the server takes over the servicing of requests.

Table 3–7 summarizes the support for high availability in OracleAS JMS and OracleAS JMS.

Table 3–7 High Availability Summary

Feature	OJMS	OracleAS JMS
High availability	RAC + OPMN	OPMN
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 machine only (different machine failover achievable through DNS and shared file system)

JMS clustering provides an environment wherein JMS applications deployed in this type of environment are able to 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 informations 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:

- Oracle Application Server 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

Oracle Application Server JMS High Availability Configuration

Oracle Application Server JMS (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 since the container processes can be spread across multiple nodes/machines. If any of the processes or machines goes down, then the other processes on an alternate machine continue to service messages.

In this section two JMS clustering scenarios are described:

- 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 need to be load balanced across OC4J instances. No configuration changes are required for this scenario.

- 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

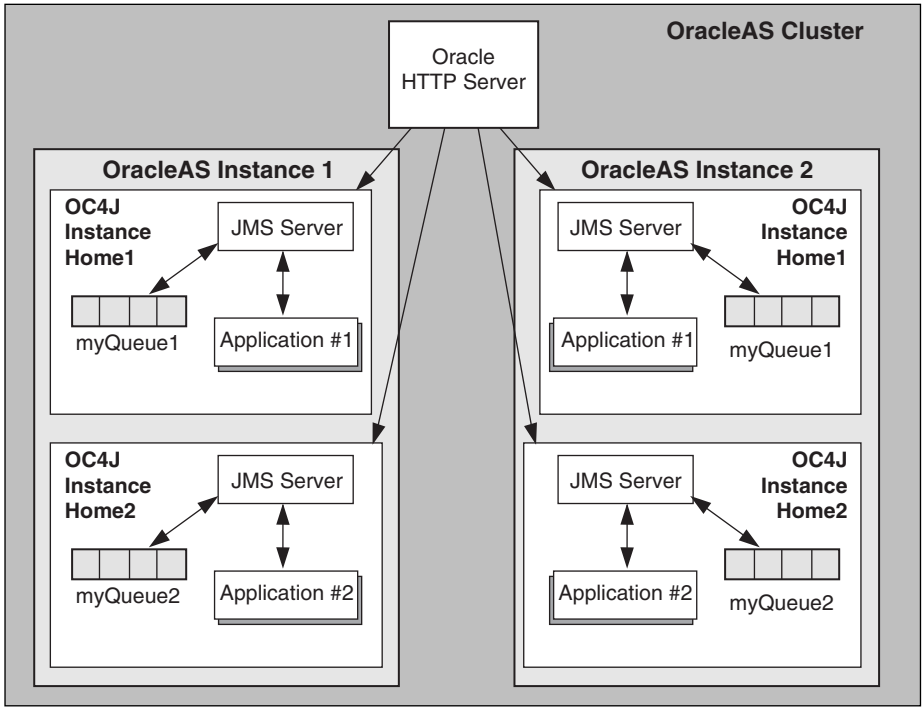
The terms being introduced here are explained in much more detail in the *Oracle Application Server 10g High Availability Guide* and the *Oracle Process Manager and Notification Server Administrator's Guide*.

- *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, where each OC4J instance has 1 to n number of 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 that are in an Oracle Application Server cluster. This of course can scale to more than two Oracle Application Server instances. There are a number of advantages to this type of deployment:

- Since applications and the JMS server are both running inside the same JVM and no inter-process communication is necessary, high throughput is achieved.
- Load balancing promotes high throughput as well as high availability.
- 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.



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 like 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.

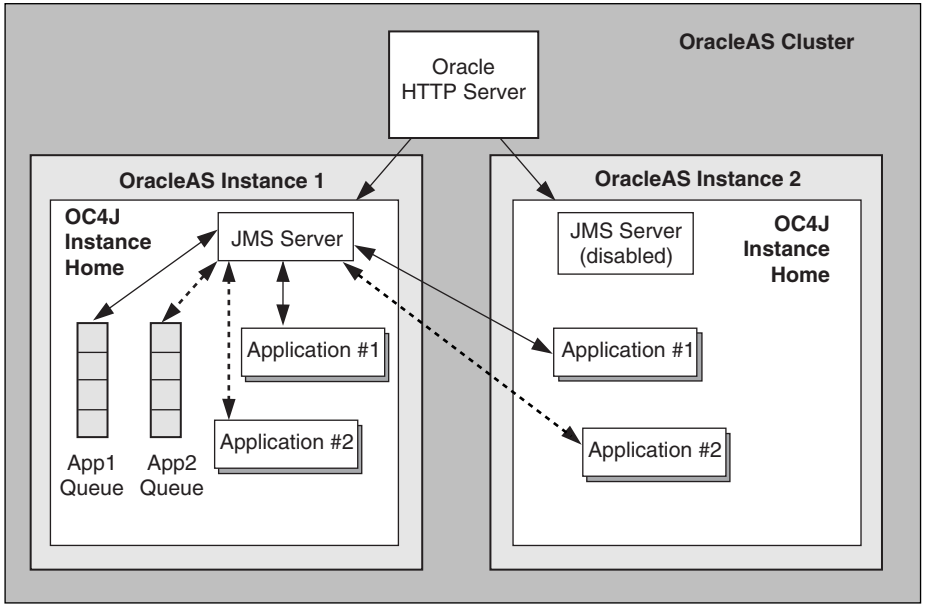
It must be noted that this type of JMS deployment is only suited for specific types of JMS applications. Assuming that message order is not a concern messages are enqueued onto distributed queues of the same name. Given the semantics of JMS point-to-point messaging, messages must not be duplicated across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

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, destinations, and 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.

While this diagram shows the active JMS server in the OC4J `Home` instance, it is recommended that the JMS provider be hosted in its own OC4J instance. For example, while `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. In the `opmn.xml` file example below, you can see that we have created an OC4J instance called `JMSserver`.



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

1. Make sure only one JVM is being started for this OC4J instance (`JMSserver`).
2. Narrow the jms port range for this instance to one value.

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 is necessary to ensure 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), the following Distributed Configuration Management (DCM) command should be run:
>
> `dcmctl updateConfig`
>
> See the *Distributed Configuration Management Reference Guide* for more information.

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

1. Assuming an OC4J instance has been created through Oracle Enterprise Manager called `JMSserver`, then the line denoted by (1) demonstrates where to locate the start of the `JMSserver` definition.
2. The line denoted by (2) is the JMS port range that OPMN works with 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.
3. 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"/>
      <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 of OC4J will have to be setup 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 of the 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 of the 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

It is up to the user to decide where the JMS application(s) will actually be deployed. While 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 JVM's.

High Availability

OPMN provides the failover mechanism to make sure the dedicated JMS server is up and running. If for some reason the JMS server fails, OPMN will detect this and restart the JVM. If there is a hardware failure then the only way to recover messages is to make sure the persisted destinations are 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

High availability is achieved with OJMS by running the following:

- An Oracle database that contains the AQ queues/topics in RAC-mode, which ensures that the database is highly available
- Oracle Application Server in OPMN-mode, which 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 backend Oracle database, which is operating in RAC-mode. JMS operations invoked on objects derived from these resource providers are directed to the RAC database.

If a failure of the application occurs, 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 should recreate their JMS state appropriately and resume operations.

If network failover of a backend database occurs, where the database is a non-RAC database, 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 should throw a `JMSException` until it gets to the point where it can lookup, through JNDI, all JMS administered objects, and proceed from there.

Failover Scenarios When Using a RAC Database With OJMS

An application that uses an RAC (real application clusters) database must handle database failover scenarios. There are two types of failover scenarios, which is detailed fully 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 10g 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 that was 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 rollback transactions and recreate the JMS state (such as connections, session, and messages that were lost).

The following example outlines the logic:

```
getMessage(QueueSesssion session)
{
    try
    {
        QueueReceiver rcvr;
         Message msgRec = null;
        QueueReceiver rcvr = session.createReceiver(rcvrQueue);
         msgRec = rcvr.receive();
    }
    catch(Exception e )
    {
        if (exc instanceof JMSException)
        {
           JMSException  jmsexc = (JMSException) 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:** Transparent application failover (TAF) is discussed fully in Chapter 4, "Data Sources".

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

However, in some cases, an ORA error is thrown when a failure occurs. OJMS passes these errors through to the user as a `JMSException` 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 3-65, you can check to see if the error returned is a fatal error or not through the

`DbUtil.oracleFatalError` method. If it is not a fatal error, the client recovers by sleeping for a short time and then retrying the current operation.

- You can recover from failback and transient errors, which are caused by incomplete failover, by retrying the use of the JMS connection after a short elapse of time. By waiting, the database failover can complete from the failure and reinstate itself.
- In the case of transaction exceptions—that is, exceptions such as "Transaction must roll back" (ORA-25402) or "Transaction status unknown" (ORA-25405)—you must rollback 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 recreate all connections and retry all non-committed 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 (JMSException ex)
{
 if (transientServerFailure)
{ // retry }
else {
 notShutdown = false;
 }
}
```

Clustering Best Practices

1. Minimize JMS client-side state.
   a. Perform work in transacted sessions.
   b. Save/checkpoint intermediate program state in JMS queues/topics for full recoverability.
   c. 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.
2. Do not use non-durable subscriptions on topics.
   a. Non-durable topic subscriptions duplicate messages per active subscriber. clustering/load-balancing creates multiple application instances. If the application creates a non-durable subscriber, it causes the duplication of each message published to the topic: which is either inefficient or semantically invalid.
   b. Use only durable subscriptions for topics. Use queues whenever possible.
3. Do not keep durable subscriptions alive for extended periods of time.

a. Only one instance of a durable subscription can be active at any given time. clustering/load-balancing creates multiple application instances. If the application creates a durable subscription, only one instance of the application in the cluster will succeed—all others will fail with a `JMSException`.

b. Create, use, and close a durable subscription in small time/code windows, minimizing the duration when the subscription is active.

c. Write application code to be aware of the fact that durable subscription creation can fail due to clustering (that is, some other instance of the application running in a cluster in 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.

4

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.

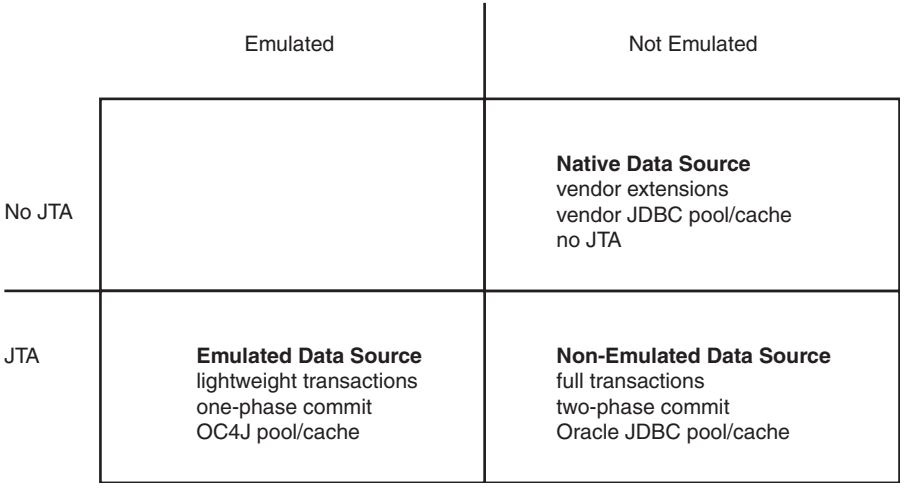
Types of Data Sources

In OC4J, Data Sources are classified as follows:

- Emulated Data Sources
- Non-emulated Data Sources
- Native Data Sources

Figure 4–1 summarizes the key differences between each data source type.

Figure 4–1 OC4JData Source Types

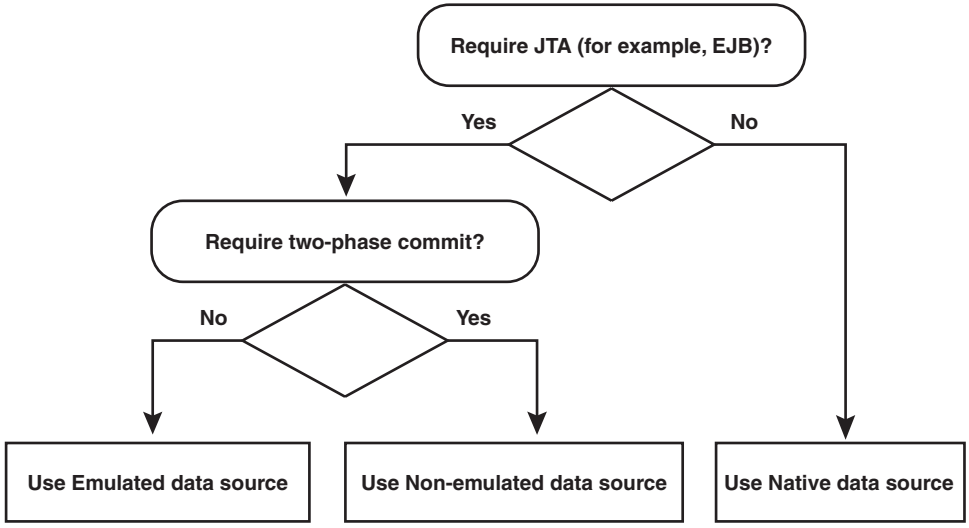


Note: If you access a non-emulated data source by the `ejb-location`, then you are using the OC4J pool and cache. If you use `OracleConnectionCacheImpl`, you can access both OC4J and Oracle JDBC pool and cache.

Note that if you access a non-emulated data source by the `ejb-location`, then you are using the OC4J pool and cache. If you use `OracleConnectionCacheImpl`, you have access to both OC4J and Oracle JDBC pool and cache.

Figure 4–2 summarizes the decision making tree that should guide you when choosing a data source type.

Figure 4–2 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 where emulated XA is preferred (such as transactions that don't require two-phase commit.)

Connections that are 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. We recommend 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"
    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 via JNDI you should look it up by the value that was 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. All three values must be specified for emulated data sources, but 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 non-emulated 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");
```

Non-emulated Data Sources

Non-emulated data sources provide full (non-emulated) JTA services, including two-phase commit capabilities for global transactions. Non-emulated 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".

We recommend that you use these data sources for distributed database communications, recovery, and reliability. Non-emulated data sources share physical connections for logical connections to the same database for the same user.

The following is a `data-sources.xml` configuration entry for a non-emulated 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
</data-source>
```

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

Here are the expected attribute definitions:

- `location` is the JNDI name that this data source is bound to 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. One must 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 only be performed via 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, be aware of the following issues:

- Only emulated and non-emulated data sources support JTA transactions.

  You cannot enlist connections that are obtained from Native data sources in a JTA transaction.

- Only non-emulated 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 the connections must use non-emulated 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, you will receive the `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 an XML file known as `data-sources.xml`.

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 4–1 summarizes the configuration requirements for each type of data source.

Table 4–1 Data Source Configuration Summary

Configuration	Non-emulated	Emulated	Native
Data source class	OrionCMTDataSource	DriverManagerData Source	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

Table 4–2 summarizes the characteristics for each type of data source.

Table 4–2 Data Source Characteristics

Characteristic	Non-emulated	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
JCA support	No	Yes	Yes

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

To define a new data source object:

1. Decide on a location for the `data-sources.xml` file (see "Configuration Files" on page 4-9)
2. Familiarize yourself with data source attributes (see "Data Source Attributes" on page 4-10)
3. Define a data source either by using the Oracle Enterprise Manager (see "Defining Data Sources in Oracle Enterprise Manager" on page 4-14) or by manually editing configuration files see "Defining Data Sources in the XML Configuration File" on page 4-15)

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` file 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, but most are optional. The required attributes are marked below. The attributes are specified in a `<data-source>` tag.

Table 4–3 lists the attributes and their meanings.

In addition to the `data-source` attributes described in Table 4–3, you can also add `property` sub-nodes 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 4–3 Data Source Attributes

Attribute Name	Meaning of Value	Default Value
`class`	Names the class that implements the data source. For non-emulated, this can be `com.evermind.sql.OrionCMTDataSource`. For emulated, this should be `com.evermind.sql.DriverManagerDataSource`. (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 non-emulated data sources. See also Table 4–1, "Data Source Configuration Summary" on page 4-8	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, some data sources that deal with `java.sql.Connection` need. For most data sources, the driver should be `oracle.jdbc.driver.OracleDriver`. Applicable only for emulated data sources where the `class` attribute is `com.evermind.sql.DriverManagerDataSource`.	None
`username`	Default user name used when getting data source connections.	None
`password`	Default password used when getting data source connections. See also "Password Indirection" on page 4-15	None
`URL`	The URL for database connections.	None
`xa-location`	The logical name of an XA data source.Emulated data sources only. See also Table 4–1, "Data Source Configuration Summary" on page 4-8	None

Table 4–3 Data Source Attributes

Attribute Name	Meaning of Value	Default Value
`ejb-location`	Use this for JTA single-phase commit transactions or looking up emulated data sources. If you use it to retrieve the data source, you can map the returned connection to `oracle.jdbc.OracleConnection`. See also Table 4–1, "Data Source Configuration Summary" on page 4-8	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 where `connection-driver` is `oracle.jdbc.driver.OracleDriver and class` is `com.evermind.sql.DriverManagerDataSource.`	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 `DataSource.getConnection` method is invoked.	0
`wait-timeout`	The number of seconds to wait for a free connection if the pool has reached `max-connections` 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

Table 4–3 Data Source Attributes

Attribute Name	Meaning of Value	Default Value
`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 or not the system is enabled for Real Application Clusters (RAC). For information on using this flag with an infrastructure database, see *Oracle High Availability Architecture and Best Practices*.) and with a user database, see "Using DataDirect JDBC Drivers" on page 4-30 and "High Availability Support for Data Sources" on page 4-33 If the data source points to an RAC database, you should set this property to `true`. This lets OC4J manage its connection pool in a way that performs better during RAC instance failures.	false
`schema`	This optional attribute specifies the `database-schema` 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 4-18.	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:@jsnyder-us:1521:jsnyder"
      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 4-2.

Defining Data Sources in Oracle Enterprise Manager

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

How to define data sources is explained in detail in the Data Sources Primer chapter of the *Oracle Application Server Containers for J2EE User's Guide*.

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 Oracle Enterprise Manager information.

This section provides a brief overview of these procedures.

Use the Oracle Enterprise Manager 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 non-emulated data sources, until the step in which you define the JNDI location. There the screen shot shows one field, **Location**, to be filled out. That is for a non-emulated data source. For 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. 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 4-2.

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"
  inactivity-timeout="30"
/>
```

See "Data Source Attributes" on page 4-10 for details on all data source attributes.

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 allows 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 4-3 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
- Configuring an Indirect Password Manually

Configuring an Indirect Password with Oracle Enterprise Manager

To configure an indirect password using the Oracle Enterprise Manager:

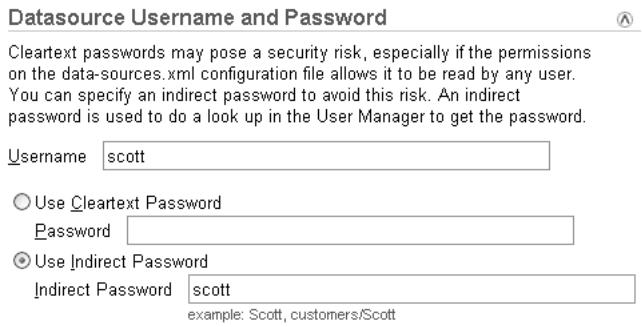
1. Log into the Oracle Enterprise Manager
2. Select a target of type OC4J.
3. Select **Administer**.

   The Oracle Enterprise Manager 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 4–3.

Figure 4–3 Edit Data Source Page



8. In the **Datasource Username and Password** area, click on **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 by either the username or by the realm and username separated by a slash ("/").

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

   This will cause 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 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 (like `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 4–4.

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 4–4 database-schema.xml File Attributes

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

The `database-schema` element may contain any number of the following sub-elements:

- type-mapping
- disallowed-field

type-mapping This sub-element 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 sub-element identifies a name that must not be used 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` 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:1521:DEBU"
    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>
```

4. Deploy your ejb and the appropriate tables will be created properly.

Using Data Sources

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

- Portable Data Source Lookup
- Retrieving a Connection from a Data Source
- Retrieving Connections with a Non-emulated 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 or SQLJ 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 that is 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 that is 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 4-27 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 either SQLJ or JDBC.

Refer to Retrieving Connections with a Non-emulated Data Source on page 4-23 for information on handling common connection retrieval error conditions.

Retrieving Connections with a Non-emulated Data Source

The physical behavior of a non-emulated data source object changes depending on whether you retrieve a connection from the data source that is 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 non-emulated 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 that is 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 non-emulated data source and you are involved in a global JTA transaction, all physical connections that are 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 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 user name that is provided is ignored. Instead, the `scott` user 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".

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:

- Specify one commit coordinator for all applications using the global `application.xml` in the `J2EE_HOME/config` directory.
- 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>
```

Note: 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*.

Note: Two-phase commit may only be configured for non-emulated data sources. For more information on data source types, see "Types of Data Sources" on page 4-2.

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 that is 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"
   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"
     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"
     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 database caching scheme to use within the data source definition. There are three types of caching schemes: `DYNAMIC_SCHEME`, `FIXED_WAIT_SCHEME`, and `FIXED_RETURN_NULL_SCHEME`. The Connection Pooling and Caching chapter of the *Oracle9i JDBC Developer's Guide and Reference*, found on OTN at the following location, describe these schemes:

`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 4–5 shows the supported values.

Table 4–5 Database Caching Schemes

Value	Cache Scheme
1	`DYNAMIC_SCHEME`
2	`FIXED_WAIT_SCHEME`
3	`FIXED_RETURN_NULL_SCHEME`

Note: The 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:DB SID
  inactivity-timeout="30">
  <property name="cacheScheme" value="1" />
</data-source>
```

In the preceding, 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, the ejb-location, xa-location, and pooled-location attributes must not be specified. Only the location attribute should be specified. Accessing the data source using any other attribute with JNDI will cause 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 (thick) 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:oci8:@`

  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:oci8:@`*`TNS service name`*

  The TNS service name is an entry in the instance `tnsnames.ora` file.

- `jdbc:oracle:oci8:@`*`full_TNS_listener_description`*

For more TNS information, see the *Oracle Net Administrator's Guide*.

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*.

Once 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 4-31.

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 version 9.04 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:sqlserver//hostname:port;User=test;Password=secret"
   inactivity-timeout="30"
 />
```

DB2

For a DB2 database, here is a data source configuration sample:

```
<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:db2://hostname:port;LocationName=jdbc;CollectionId=default;"
  inactivity-timeout="30"
/>
```

Sybase

For a Sybase database, here is a data source configuration sample:

```
<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:sybase://hostname:port;User=test;Password=secret"
  inactivity-timeout="30"
/>
```

High Availability Support for Data Sources

Introduction

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:

- Redundant middle-tier or application tier (Oracle Application Server), network, and storage infrastructure.
- Oracle Data Guard to protect from human errors and data failures and recover from site failures.
- Real Application Clusters (RAC) at each site to protect from host and instance failures.
- Sound operational best practices (such as using fast-start check pointing to control the amount of time required to recover from an instance failure).

For more information about MAA, see
http://otn.oracle.com/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 by providing log transport services, managed recovery, switchover, and failover features.

Real Application Clusters (RAC) RAC uses two or more nodes or machines, 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 a 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. On-going 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 only available when using the thick JDBC 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 that was in

progress. This reconnect happens automatically from within the Oracle Call Interface (OCI) library.

TAF provides a best effort failover mechanism for on-going operations on a database connection created against a database instance which is part of a RAC cluster. It also attempts to ensure that existing connections (which 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 which 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 rollback the current transaction before the database connection can be used again. Your application will also need to replay all the operations past the last committed transaction to get into 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 4-40)

High Availability Support in OC4J

Oracle Application Server Containers for J2EE can be integrated with RAC, Data Guard, and TAF as part of your HA architecture.

The remainder of this section describes configuration issues specific to Oracle Application Server Containers for J2EE that relate directly to HA. Use this information in conjunction with MAA recommendations and procedures.

Oracle Application Server Containers for J2EE HA configuration issues include:

- 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 will randomly choose one of the alternatives if the current host is unreachable.

   For details on data source configuration, see "Defining Data Sources" on page 4-8.

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 4-38.
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 the `url` element `failover` is `on` and `failover_mode` is defined. As long as two or more hosts are specified, the JDBC client will randomly choose one of the alternatives if the current host is unreachable. For a description of `failover_mode` options, see Table 4–6, "TAF Configuration Options" on page 4-38.

For details on data source configuration, see "Defining Data Sources" on page 4-8.

Note: Only data sources configured to use the thick JDBC client can be configured for use with TAF.

---

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>
```

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 sub-parameters described in Table 4–6.

Table 4–6 TAF Configuration Options

Subparameter	Description
BACKUP	Specify a different net service name for backup connections. A backup should be specified when using the PRECONNECT METHOD to pre-establish connections.
TYPE	Specify the type of failover. Three types of Oracle Net failover functionality are available by default to Oracle Call Interface (OCI) applications: ■ SESSION: Set to failover the session. If a user's connection is lost, 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.
METHOD	Determines how fast failover occurs from the primary node to the backup node: ■ 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.

Table 4–6 TAF Configuration Options

Subparameter	Description
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 *Oracle10i 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 will issue a simple commit (instead of Two-Phase Commit) which will make the transaction suspect. If your application will encounter 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, 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 an `SQLException` with error code of 3113 or 3114.

When an exception occurs while using a user connection handle, it is useful for OC4J to detect if 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 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 a`java.sql.SQLException` is thrown, all un-allocated connections are removed from the pool.

In a RAC environment, when a`java.sql.SQLException` is thrown, first the states of all un-allocated connections are checked. If they are alive, they 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 via 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. JDBC Connection failed/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) call backs and failover events, your application can customize TAF operation 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 fail-over using OCI callback capabilities. Table 4–7 describes the fail-over events defined in the OCI API.

Table 4–7 OCI API Fail-Over Events

Symbol	Value	Meaning
FO_BEGIN	1	A lost connection has been detected and fail over is starting.
FO_END	2	A successful completion of fail-over.
FO_ABORT	3	An unsuccessful fail-over with no option of retrying.
FO_REAUTH	4	A user handle has been re-authenticated.
FO_ERROR	5	A fail-over was temporarily unsuccessful but the application has the opportunity to handle the error and retry.
FO_RETRY	6	Retry fail-over.
FO_EVENT_UNKNOWN	7	A bad/unknown fail-over event.

For more information, see the Oracle Call Interface Programmer's Guide.

SQL Exception Handling

Depending on the driver type used, SQLExceptions 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` that is thrown to the caller.

Table 4–8 summarizes these issues by driver type.

Table 4–8 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 4-40), replay on surviving node works.	After application sends acknowledgement to TAF (see "Acknowledging TAF Exceptions" on page 4-40), replay on surviving node works.	If application sends acknowledgement to TAF (see "Acknowledging TAF Exceptions" on page 4-40), then OC4J proceeds transparently.

5

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 10*g* (9.0.4) 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 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 will resend 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 ensures that each client (single- or multi-threaded) will have one connection to the remote server.

Co-located Object Support

For co-located objects, RMI/ORMI will detect the co-located scenario and avoid the extra, unnecessary socket call.

The same is true when the JNDI registry is co located.

Client-Side Requirements

In order to access EJBs, you must do the following on the client-side:

1. Download the `oc4j_client.zip` file from `http://otn.oracle.com/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 JARs contain the classes necessary for client interaction. You only need to add `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*.

Configuring OC4J for RMI

You can configure OC4J for RMI in one of two ways:

- Configuring RMI Using Oracle Enterprise Manager
- Configuring RMI Manually

Oracle recommends that you configure OC4J using the Oracle Enterprise Manager.

Once OC4J is configured for RMI, you must specify RMI properties as described in "RMI Configuration Files" on page 5-10.

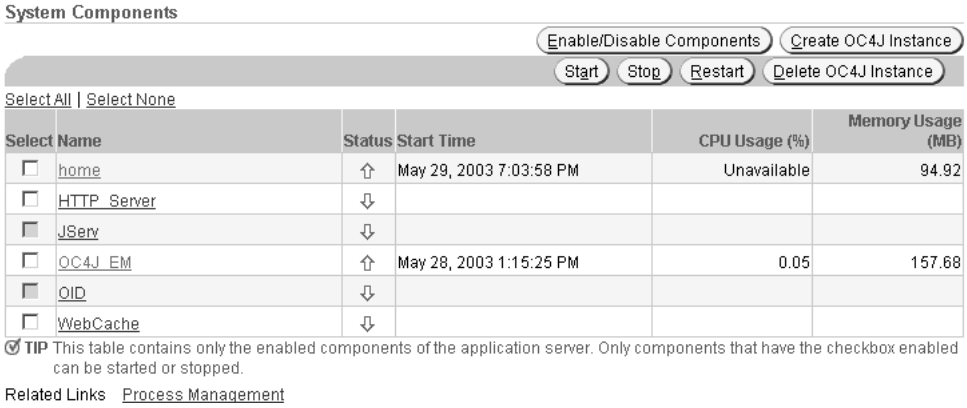
Configuring RMI Using Oracle Enterprise Manager

Oracle recommends that you configure OC4J to use RMI by using Oracle Enterprise Manager as follows:

1. Navigate to an OC4J instance in which you want to allow access to applications through RMI.

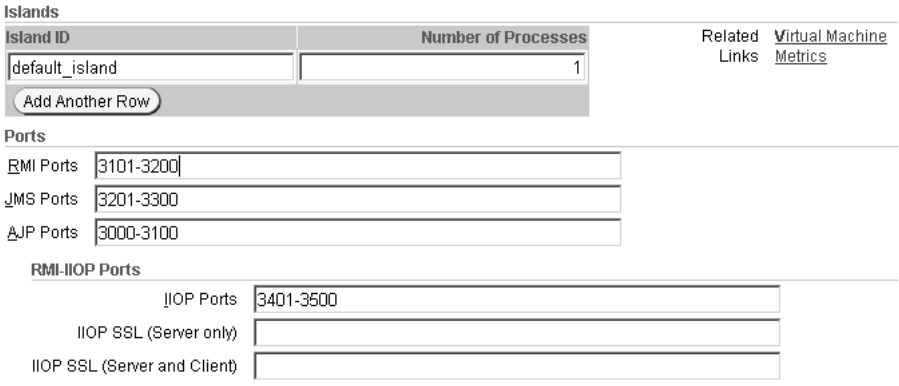
   Figure 5–1 shows an OC4J instance called `home`.

Figure 5–1 Oracle Enterprise Manager System Components



2. Click the OC4J instance name.
3. Click the **Administration** tab.
4. Click **Server Properties**.
5. 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.

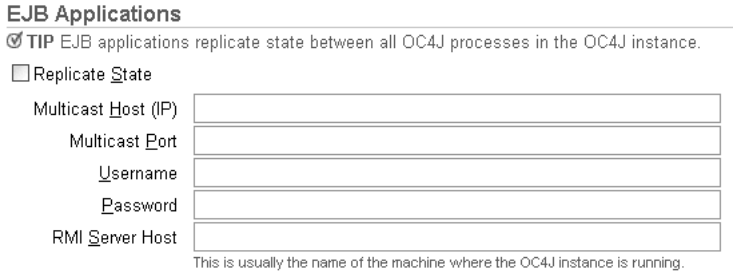
Figure 5–2 Oracle Enterprise Manager Server Properties Port Configuration



6. Click **Apply**.
7. Click the **Back** button on your browser.
8. Click **Replication Properties**.
9. 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 Replication Properties



10. 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 *Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide* section "Configure the Multicast Address for EJB Clustering".

---

11. Click **Apply**.

Configuring RMI Manually

Oracle recommends that you configure OC4J using the Oracle Enterprise Manager as described in "Configuring RMI Using Oracle Enterprise Manager" on page 5-4. If you choose to manually configure RMI, you must:

1. Edit property file `server.xml` (see "Editing server.xml" on page 5-7).
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 5-7) only.
   - In an Oracle Application Server environment, edit both the `rmi.xml` file (see "Editing rmi.xml" on page 5-7) and the `opmn.xml` file (see "Editing opmn.xml" on page 5-10).

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 5-10); 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>
   ```

   The user-replaceable attributes of the `<rmi-server>` element are:

   - *`hostname`*: the host name or IP address from which the RMI server accepts RMI requests. If you omit this attribute, the RMI server will accept 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 5-10); 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. The user-replaceable attributes of the `server` element are:

   - *`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).

The user-replaceable attributes of the `log` element are:

- *`odlpathname`*—is 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`*—is the maximum size in bytes of each individual log file.
- *`num`*—is the maximum number of log files.
- *`logfilepathname`*—is the path name of a log file (*`logfilepathname`*) to which the server writes all RMI requests.

The `<odl>` element is new in the OC4J 10*g* (9.0.4) 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 `log`*`N`*`.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 is opened to continue the logging, `log2.xml`. 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 10g Administrator's Guide*.

2. Apply changes by running the following on the Oracle Application Server command line:

```
dcmctl 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	`server.xml`	The `<sep-config>` element in this file specifies the path name, normally `internal-settings.xml`, for the server extension provider properties. Example: `<sep-config path="./internal-settings.xml">`
Application	`jndi.properties`	This file specifies the URL of the initial naming context used by the client. See "JNDI Properties for RMI" on page 5-11

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 5-11)
- `java.naming.factory.initial` (see "Context Factory Usage" on page 5-14)

Naming Provider URL

Set the`java.naming.provider.url` using the following syntax:

```
<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`	Use `opmn:ormi` for Oracle Application Server applications. Use `ormi` for standalone OC4J applications. Use `http:ormi` for applications that use HTTP tunneling (see "Configuring ORMI Tunneling through HTTP" on page 5-16). Use `corbaname` for applications that must interoperate with non-OC4J containers (see "The corbaname URL" on page 6-13).
`host`	For Oracle Application Server applications, the name of the OPMN host as defined in the `opmn.xml` file. Although OPMN is often located on the same machine as the OC4J instance. Specify the host name in case it is located on another machine. For standalone OC4J applications, the port number defined by the `rmi.xml` file `rmi-server` element `host` attribute.

Table 5–2 Naming Provider URL

Variable	Description
port	In Oracle Application Server release 10*g* (9.0.4) when the opmn:ormi prefix is used, specify the request port on which the opmn process is listening and the opmn 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 5-12). If omitted, the default request port value 6003 is used. In Oracle Application Server releases before 10*g* (9.0.4) when the ormi prefix is used, you must specify the RMI port that opmn selected for your OC4J instance (see "Using opmnctl to Show the Selected RMI Port" on page 5-13). For standalone OC4J applications when the ormi prefix is used, you must specify the port number defined by the rmi.xml file rmi-server element port attribute. For applications that use HTTP tunneling and use the http:ormi prefix, see "Configuring ORMI Tunneling through HTTP" on page 5-16 for information on what port to specify. For applications that must interoperate with non-OC4J containers and use the corbaname prefix, see "The corbaname URL" on page 6-13 for information on what port to specify.
oc4j_instance	For Oracle Application Server applications, the name of the OC4J instance as defined in the Enterprise Manager. For standalone OC4J applications, this is not applicable.
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 release 10*g* (9.0.4), 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 will forward 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 and 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: 10g (9.0.4) Release" on page 5-16.

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 and 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 stand-alone 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.

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 of the OC4J container, use the `ApplicationClientInitialContextFactory` class.
- If the client is an EJB or servlet client within the OC4J container, 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, use the `RMIInitialContextFactory` class.
- If the client is going to use DNS load balancing, use the `RMIInitialContextFactory` class.

Example Lookups

This section provides examples that show how to lookup an EJB in:

- OC4J Standalone
- OC4J in Oracle Application Server: Releases Before 10g (9.0.4)
- OC4J in Oracle Application Server: 10g (9.0.4) Release

OC4J Standalone

The following example shows how to lookup an EJB called `MyCart` in the J2EE application `ejbsamples` deployed in a standalone OC4J instance. The application is located on a machine 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 10*g* (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 10*g* (9.0.4), 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, you have to limit the port ranges for RMIs to a specific number, for example: 3101-3101.

The following example shows how to lookup an EJB called `MyCart` in the J2EE application `ejbsamples` in an Oracle Application Server environment in releases before 10*g* (9.0.4). The application is located on a machine 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: 10*g* (9.0.4) Release

In Oracle Application Server release 10*g* (9.0.4), you can use the following type of lookup in the URL to lookup 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 lookup EJB called `MyCart` in the J2EE application `ejbsamples` in an Oracle Application Server environment in release 10*g* (9.0.4). The EJB application is located on a machine named `localhost`. The differences between this invocation and the stand-alone 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 5-11). 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 5-17) 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 used to start 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 maps an OC4J instance to URLs that start with a specified path name. This mapping is specified in the OC4J `mod-oc4j.conf` file by Oracle Enterprise Manager at deployment time. This 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`.

An OC4J mount point is used to direct tunneling requests to an OC4J instance other than the home instance of OC4J.

The first part of "Configuring ORMI Tunneling through HTTP" on page 5-16 showed the following URL:

```
http:ormi://hostname:http_port/appName@oc4j_mount
```

In this URL, *appName* is the name of the application (defined by `default-web-site.xml` attribute `name`), 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" />
```

Therefor, 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*.

Warning. Do not manually modify these configuration files. The Oracle Enterprise Manager internally makes these changes when an application is deployed. Manual modification of these files may risk putting the repository out of synchronization.

6

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) protocol.

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 6-4 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 the first three of these features.

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 10*g* (9.0.4) 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

In order to access EJBs, you must do the following on the client-side:

1. Download the `oc4j_client.zip` file from `http://otn.oracle.com/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 JARs contain the classes necessary for client interaction. You only need to add `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 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 5-3.

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
 -deployment_name 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 were 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 6-13.

> **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 (the Interface Description Language) describing its interfaces. See "Configuring OC4J for Interoperability" on page 6-16 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 "CSIv2 Security Properties (orion-ejb-jar.xml)" on page 6-22 and "EJB Server Security Properties (internal-settings.xml)" on page 6-17 for details.
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.

   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`

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 6-13.

   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 (the Interface Description Language), describing its interfaces. See "Configuring OC4J for Interoperability" on page 6-16 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:

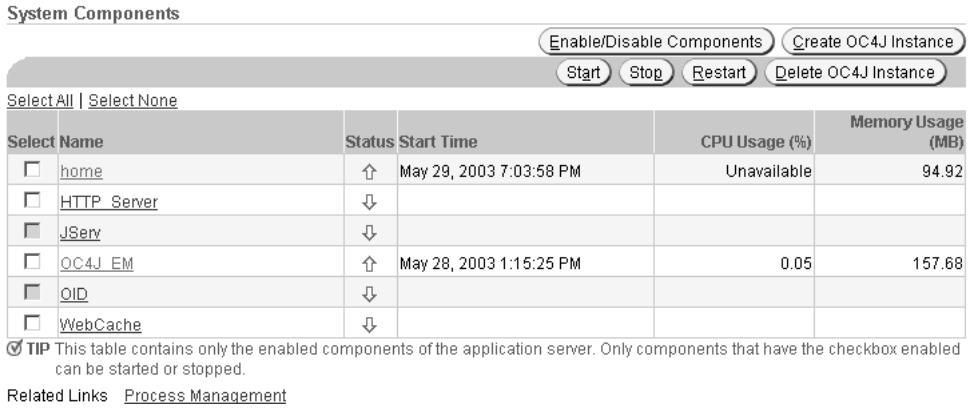
- Configuring for Interoperability Using Oracle Enterprise Manager
- Configuring for Interoperability Manually

Configuring for Interoperability Using Oracle Enterprise Manager

You can configure an EJB to be accessible by means of RMI/IIOP in an Oracle Application Server environment by using Oracle Enterprise Manager. Follow these steps:

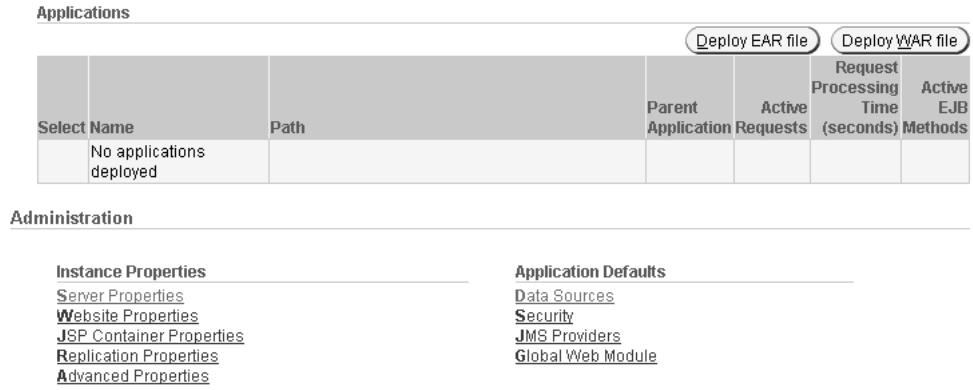
1. Navigate to an OC4J instance in which you want to allow access to applications through RMI/IIOP. Figure 6–1 shows an OC4J instance called `home`.

Figure 6–1 Oracle Enterprise Manager System Components



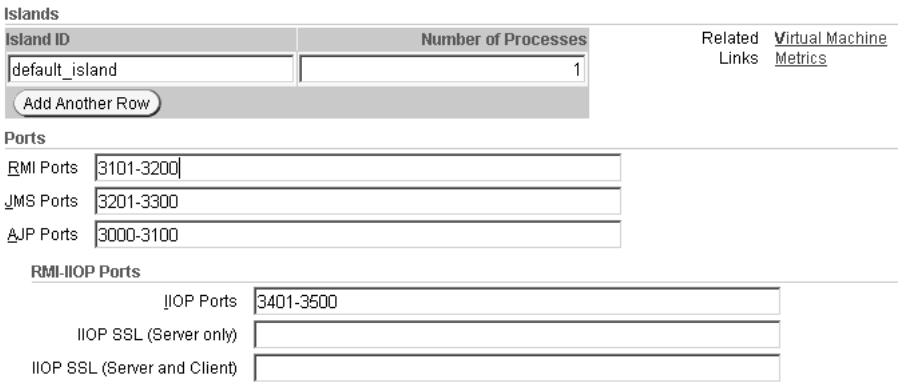
2. Click on Server Properties in the Administration section for this OC4J instance. This is shown in Figure 6–2.

Figure 6–2 Oracle Enterprise Manager Server Properties



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. Click Apply.

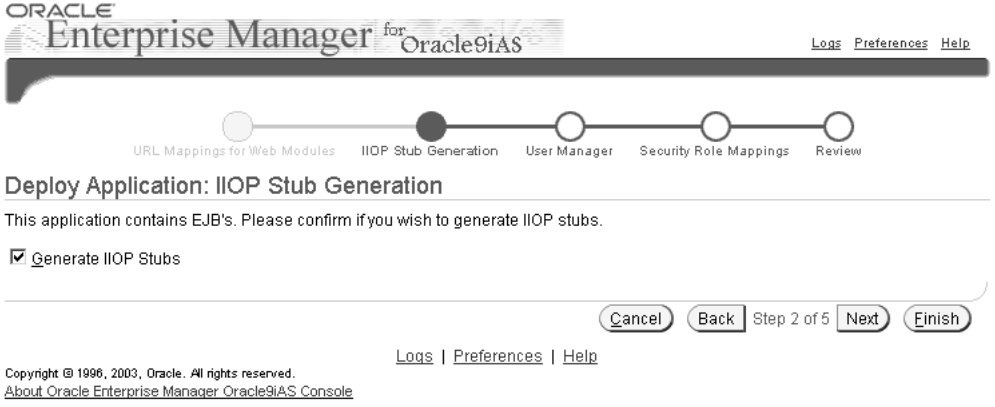
Figure 6–3 Oracle Enterprise Manager Port Configuration



Deploy your application following the Oracle Enterprise Manager deployment wizard.

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 Stub Generation



Finish deploying your application following the Oracle Enterprise Manager 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.xm 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, 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
```

and then:

```
opmnctl startall
```

For information on Oracle Enterprise Manager, 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 6-13. For details on the OPMN URL, see "The OPMN URL" on page 6-14.

   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 (the Interface Description Language), describing its interfaces. See "Configuring OC4J for Interoperability" on page 6-16 for a discussion of command-line options.

Advanced Interoperability in Oracle Application Server Environment

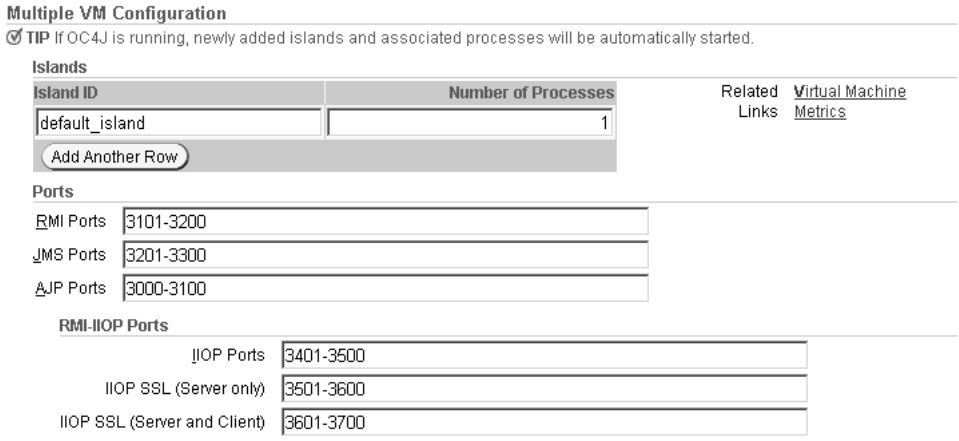
You can access an EJB using RMI/IIOP in an Oracle Application Server environment in two ways:

- Configuring for Interoperability Using Oracle Enterprise Manager
- Configuring for Interoperability Manually

Configuring for Interoperability Using Oracle Enterprise Manager

The advanced configuring for interoperability using Oracle Enterprise Manager differs from the simple configuring described under "Configuring for Interoperability Using Oracle Enterprise Manager" on page 6-6 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.xm` file of OC4J. This is shown in Figure 6–5.

Figure 6–5 Oracle Enterprise Manager Port Specifications



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.

1. Specify CSIv2 security policies for the bean in `orion_ejb_jar.xml` and in `internal_settings.xml`. See "CSIv2 Security Properties (orion-ejb-jar.xml)" on page 6-22 and "EJB Server Security Properties (internal-settings.xml)" on page 6-17 for details.
2. 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, your client cannot connect to iiops1 or iiops2 ports because OPMN-allocated ports are not reported to OC4J.

---

3. Using opmnctl or Oracle Enterprise Manager, 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
```

and then:

```
opmnctl startall
```

For information on Oracle Enterprise Manager, 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 6-13. For details on the OPMN URL, see "The OPMN URL" on page 6-14.

   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 (the Interface Description Language), describing its interfaces. See "Configuring OC4J for Interoperability" on page 6-16 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 is most used by EJB developers. 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 specific to RMI/IIOP. For general information about the OPMN URL, see "JNDI Properties for RMI" on page 5-11.

In an Oracle Application Server environment, IIOP ports for all OC4J processes within each Oracle Application Server instance are dynamically managed by OPMN. Because the ports are dynamically allocated by OPMN, it might 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
```

Note: NOTE: For the OC4J 10*g* (9.0.4) implementation, load balancing and failover are supported only for ORMI, not IIOP.

Note: If you choose to 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
`javax.transaction.TransactionRolledbackException`	`TRANSACTION_ROLLEDBACK`
`javax.transaction.TransactionRequiredException`	`TRANSACTION_REQUIRED`
`javax.transaction.InvalidTransactionException`	`INVALID_TRANSACTION`
`java.rmi.NoSuchObjectException`	`OBJECT_NOT_EXIST`
`java.rmi.AccessException`	`NO_PERMISSION`
`java.rmi.MarshalException`	`MARSHAL`
`java.rmi.RemoteException`	`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.

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 `<sep-config>` element in this file specifies the path name, normally `internal-settings.xml`, for the server extension provider properties. For example: <sep-config path="./internal-settings.xml">
	`internal-settings.xml`	This file specifies server extension provider properties that are specific to RMI/IIOP. See "EJB Server Security Properties (internal-settings.xml)" on page 6-17 for details.
Application	`orion-ejb-jar.xml`	The `<ior-security-config>` subentity of the `<session-deployment>` and `<entity-deployment>` entities specifies Common Secure Interoperability Version 2 (CSIv2) security properties for the server. See "CSIv2 Security Properties" on page 6-19 for details.

Table 6–2 Interoperability Configuration Files

Context	File	Description
	`ejb_sec.properties`	This file specifies client-side security properties for an EJB. See "EJB Client Security Properties (ejb_sec.properties)" on page 6-24 for details.
	`jndi.properties`	This file specifies the URL of the initial naming context used by the client. See "JNDI Properties for Interoperability (jndi.properties)" on page 6-25 for details.

EJB Server Security Properties (internal-settings.xml)

You specify server security properties in the `internal-settings.xml` file.

Note: You cannot edit `internal-settings.xml` with the Oracle Enterprise Manager.

Note: If you choose to configure your client's JNDI property `java.naming.provider.url` to use an `OPMN` URL, your client cannot connect to `ssl-port` and `ssl-client-server-auth-port` ports because OPMN-allocated ports are not reported to OC4J.

This file specifies certain properties as values within `<sep-property>` entities. Table 6–3 contains a list of properties.

The table refers to *keystore* and *truststore* files, which use the Java Key Store (JKS), a JDK-specified format, to store keys and certificates. A keystore stores a map of private keys and certificates. A truststore stores trusted certificates for the certificate authorities (CAs, such as VeriSign and Thawte).

Table 6–3 EJB Server Security Properties

Property	Meaning
`port`	IIOP port number (defaults to `5555`).
`ssl`	`true` if IIOP/SSL is supported, `false` otherwise.

Table 6–3 EJB Server Security Properties (Cont.)

Property	Meaning
`ssl-port`	IIOP/SSL port number (defaults to `5556`). This port is used for server-side authentication only. If your application uses client and server authentication, you also must set `ssl-client-server-auth-port`.
`ssl-client-server-auth-port`	Port used for client and server authentication (defaults to `5557`). This is the port on which OC4J listens for SSL connections that require both client and server authentication. If not set, OC4J will listen on `ssl-port` + 1 for client-side authentication.
`keystore`	Name of keystore (used only if `ssl` is `true`).
`keystore-password`	The keystore password (used only if `ssl` is `true`).
`trusted-clients`	Comma-separated list of hosts whose identity assertions can be trusted. Each entry in the list can be an IP address, a host name, a host name pattern (for instance, `*.example.com`), or `*`. An `*` alone means that all clients are trusted. The default is to trust no clients.
`truststore`	Name of truststore. If you do not specify a truststore for a server, OC4J uses the keystore as the truststore (used only if `ssl` is `true`).
`truststore-password`	Truststore password (can be set only if `ssl` is `true`).

Note: In Table 6–3, the properties `keystore-password` and`truststore-password` support password indirection. For more information, refer to the *Oracle Application Server Containers for J2EE Security Guide*.

If OC4J is started by the Oracle Process Management Notification service (OPMN) in an Oracle Application Server (as opposed to standalone) environment, then ports specified in `internal-settings.xml` are ignored. If OPMN is configured to disable IIOP for a particular OC4J instance, then, even though IIOP may be enabled

through `internal-settings.xml` (as pointed to by `server.xml`), IIOP is not enabled.

The following example shows a typical `internal-settings.xml` file:

```
<server-extension-provider name="IIOP"
        class="com.oracle.iiop.server.IIOPServerExtensionProvider">
    <sep-property name="port" value="5555" />
    <sep-property name="host" value="localhost" />
    <sep-property name="ssl" value="false" />
    <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="123456" />
    <sep-property name="truststore" value="truststore.jks" />
    <sep-property name="truststore-password" value="123456" />
    <sep-property name="trusted-clients" value="*" />
</server-extension-provider>
```

> **Note:** Although the default value of `port` is one less than the default value for `ssl-port`, this relationship is not required.

Here is the DTD for `internal-settings.xml`:

```
<!-- A server extension provider that is to be plugged in to the server.
-->
<!ELEMENT server-extension-provider (sep-property*) (#PCDATA)>
<!ATTLIST server-extension-provider name class CDATA #IMPLIED>
<!ELEMENT sep-property (#PCDATA)>
<!ATTLIST sep-property name value CDATA #IMPLIED>
<!-- This file contains internal server configuration settings. -->
<!ELEMENT internal-settings (server-extension-provider*)>
```

CSIv2 Security Properties

CSIv2 is an Object Management Group (OMG) standard for a secure interoperable wire protocol that supports authorization and identity delegation. You configure CSIv2 properties in three different locations:

- `internal_settings.xml`
- `orion-ejb-jar.xml`
- `ejb_sec.properties`

"CSIv2 Security Properties (internal-settings.xml)" on page 6-20, "CSIv2 Security Properties (orion-ejb-jar.xml)" on page 6-22, and "EJB Client Security Properties (ejb_sec.properties)" on page 6-22 discusses these configuration files.

CSIv2 Security Properties (internal-settings.xml)

This section discusses the semantics of the values you set within the `<sep-property>` element in `internal_settings.xml`. For details of syntax, see "EJB Server Security Properties (internal-settings.xml)" on page 6-17.

To use the CSIv2 protocol with OC4J, you must both set `ssl` to `true` and specify an IIOP/SSL port (`ssl-port`).

- If you do not set `ssl` to `true`, then CSIv2 is not enabled. Setting `ssl` to `true` permits clients and servers to use CSIv2, but does not require them to communicate using SSL.
- If you do not specify an `ssl-port`, then no CSIv2 component tag is inserted by the server into the IOR, even if you configure an `<ior-security-config>` entity in `orion-ejb-jar.xml`.

When IIOP/SSL is enabled on the server, OC4J listens on two different sockets—one for server authentication alone, and one for server and client authentication. You specify the server authentication port within the `<sep-property>` element. The server and client authentication listener uses the port number immediately following.

For SSL clients using server authentication alone, you can specify:

- Truststore only
- Both keystore and truststore.
- Neither

If you specify neither keystore nor truststore, then the handshake may fail if there are no default truststores established by the security provider.

SSL clients using client-side authentication must specify both a keystore and a truststore. The certificate from the keystore is used for client authentication.

CSIv2 Security Properties (ejb_sec.properties)

If the client does not use client-side SSL authentication, you must set `client.sendpassword` in the `ejb_sec.properties` file for the client runtime

to insert a security context and send the user name and password. You must also set `server.trustedhosts` to include your server.

> **Note:** Server-side authentication takes precedence over a user name and password.

If the client does use client-side SSL authentication, the server extracts the `DistinguishedName` from the client's certificate and then looks it up in the corresponding user manager. It does not perform password authentication.

Trust Relationships

Two types of trust relationships exist:

- Clients trusting servers to transmit user names and passwords using non-SSL connections
- Servers trusting clients to send *identity assertions*, which delegate an originating client's identity

Clients list trusted servers in the EJB property `oc4j.iiop.trustedServers`. See Table 6–4, "EJB Client Security Properties" on page 6-24 for details. Servers list trusted clients in the `trusted-client` property of the `<sep-property>` element in `internal-settings.xml`. See "EJB Server Security Properties (internal-settings.xml)" on page 6-17 for details.

Conformance level 0 of the EJB standard defines two ways of handling trust relationships:

- *presumed trust*, in which the server presumes that the logical client is trustworthy, even if the logical client has not authenticated itself to the server, and even if the connection is not secure
- *authenticated trust*, in which the target trusts the intermediate server, based on authentication either at the transport level, or in the `trusted-client` list, or both

> **Note:** You can also configure the server to both require SSL client-side authentication and also specify a list of trusted client (or intermediate) hosts that are allowed to insert identity assertions.

OC4J offers both kinds of trust. You configure trust using the bean's <ior-security-config> element in orion-ejb-jar.xml. See "CSIv2 Security Properties (orion-ejb-jar.xml)" on page 6-22 for details.

CSIv2 Security Properties (orion-ejb-jar.xml)

This section discusses the CSIv2 security properties for an EJB. You configure each individual bean's CSIv2 security policies in its orion-ejb-jar.xml. The CSIv2 security properties are specified within <ior-security-config> elements. Each element contains a <transport-config> element, an <as-context> element, and an <sas-context> element.

The <transport-config> Element

This element specifies the transport security level. Each element within <transport-config> must be set to supported, required, or none. None means that the bean neither supports nor uses that feature; supports means that the bean permits the client to use the feature; required means that the bean insists that the client use the feature. The elements are:

- <integrity>: Is there a guarantee that all transmissions are received exactly as they were transmitted?
- <confidentiality>: Is there a guarantee that no third party was able to read transmissions?
- <establish-trust-in-target>: Does the server authenticate itself to the client?
- <establish-trust-in-client>: Does the client authenticate itself to the server?

Notes: If you set <establish-trust-in-client> to required, this overrides specifying username_password in <as-context>. If you do this, you must also set the <required> node value in the <as-context> section to false; otherwise, access permission issues will arise.

Setting any of the <transport-config> properties to required means that the bean will use RMI/IIOP/SSL to communicate.

The <as-context> element

This element specifies the message-level authentication properties.

- `<auth-method>`: must be set to either `username_password` or `none`. If set to `username_password`, beans use user names and passwords to authenticate the caller.
- `<realm>`: must be set to `default` at the current release.
- `<required>`: if set to `true`, the bean requires the caller to specify a user name and password.

The <sas-context> element

This element specifies the identity delegation properties. It has one element, `<caller-propagation>`, which can be set to `supported`, `required`, or `none`. If the `<caller-propagation>` element is set to `supported`, then this bean accepts delegated identities from intermediate servers. If it is set to `required`, then this bean requires all other beans to transmit delegated identities. If set to `none`, this bean does not support identity delegation.

An example:

```
<ior-security-config>
  <transport-config>
    <integrity>supported</integrity>
    <confidentiality>supported</confidentiality>
    <establish-trust-in-target>supported</establish-trust-in-target>
    <establish-trust-in-client>supported</establish-trust-in-client>
  </transport-config>
  <as-context>
    <auth-method>username_password</auth-method>
    <realm>default</realm>
    <required>true</required>
  </as-context>
  <sas-context>
    <caller-propagation>supported</caller-propagation>
  </sas-context>
</ior-security-config>
```

DTD The DTD for the `<ior-security-config>` element is:

```
<!ELEMENT ior-security-config (transport-config?, as-context?
sas-context?) >
<!ELEMENT transport-config (integrity, confidentiality,
establish-trust-in-target, establish-trust-in-client) >
```

```
<!ELEMENT as-context (auth-method, realm, required) >
<!ELEMENT sas-context (caller-propagation) >
<!ELEMENT integrity (#PCDATA) >
<!ELEMENT confidentiality (#PCDATA)>
<!ELEMENT establish-trust-in-target (#PCDATA) >
<!ELEMENT establish-trust-in-client (#PCDATA) >
<!ELEMENT auth-method (#PCDATA) >
<!ELEMENT realm (#PCDATA) >
<!ELEMENT required (#PCDATA)> <!-- Must be true or false -->
<!ELEMENT caller-propagation (#PCDATA) >
```

EJB Client Security Properties (ejb_sec.properties)

Any client, whether running inside a server or not, has EJB security properties. Table 6–4 lists the EJB client security properties controlled by the `ejb_sec.properties` file. By default, OC4J searches for this file in the current directory when running as a client or in `J2EE_HOME/config` when running in the server. You can specify this file's location explicitly with `-Dejb_sec_properties_location=`*`pathname`*.

Table 6–4 EJB Client Security Properties

Property	Meaning
`# oc4j.iiop.keyStoreLoc`	The path name for the keystore.
`# oc4j.iiop.keyStorePass`	The password for the keystore.
`# oc4j.iiop.trustStoreLoc`	The path name for the truststore.
`# oc4j.iiop.trustStorePass`	The password for the truststore.
`# oc4j.iiop.enable.clientauth`	Whether the client supports client-side authentication. If this property is set to `true`, you must specify a keystore location and password.

`oc4j.iiop.ciphersuites`	Which cipher suites are to be enabled. Here are the valid cipher suites:
	`TLS_RSA_WITH_RC4_128_MD5`
	`SSL_RSA_WITH_RC4_128_MD5`
	`TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA`
	`SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA`
	`TLS_RSA_EXPORT_WITH_RC4_40_MD5`
	`SSL_RSA_EXPORT_WITH_RC4_40_MD5`
	`TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA`
	`SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA`

Table 6–4 EJB Client Security Properties (Cont.)

Property	Meaning
`nameservice.useSSL`	Whether to use SSL when making the initial connection to the server.
`client.sendpassword`	Whether to send user name and password in clear form (unencrypted) in the service context when not using SSL. If this property is set to `true`, the user name and password are sent only to servers listed in the `trustedServer` list.
`oc4j.iiop.trustedServers`	A list of servers that can be trusted to receive passwords sent in clear form. Has no effect if `client.sendpassword` is set to `false`. The list is comma-separated. Each entry in the list can be an IP address, a host name, a host name pattern (for instance, `*.example.com`), or `*`. An `*` alone means that all servers are trusted.

Note: The properties marked with a # can be set either in `ejb_sec.properties` or as system properties. The settings in `ejb_sec.properties` always override settings that are specified as system properties.

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 6-13. For details on the OPMN URL, see "The OPMN URL" on page 6-14.
- `contextFactory` can be either `ApplicationClientInitialContextFactory` or the class `IIOPInitialContextFactory`.

  If your application has an `application-client.xml`, then leave `contextFactory` set to `ApplicationClientInitialContextFactory`. If your application does not have an `application-client.xml`, then change `contextFactory` to `IIOPInitialContextFactory`.

Context Factory Usage

`com.evermind.server.ApplicationClientInitialContextFactory` is used when looking up remote objects from stand-alone 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.iiop.server.IIOPInitialContextFactory` is used when looking up remote objects between different containers using the IIOP protocol.

7

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

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.

Code examples are available for download from the OTN OC4J sample code site:

```
http://otn.oracle.com/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 4-20 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, you can use single-phase commit.
- Two-Phase Commit (2pc): If more than one resource is enlisted, 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.

Use the default data source (`data-sources.xml`) that comes with a standard OC4J installation if you can 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:@myhost:myport:mySID"
  inactivity-timeout="30"
/>
```

In the preceding code, *myhost*, *myport*, and *mySID* are entries that you must change. The values that actually appear likely are not valid for your installation.

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". Basically, 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();
```

If you are using JDBC, you can start preparing and executing statements against the database. If you are using SQLJ, create a default context to specify in the `#sql` statement.

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()`—that is, use the method with no arguments.
- Use `ds.getConnection(`*`username`*`, `*`password`*`)`—that is, 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.

Example 7–1 shows a small portion of an employee session bean that uses container-managed transactions (CMTs) and uses SQLJ for updating the database.

Example 7–1 Retrieving a Connection Using Portable JNDI Lookup

```
int empno = 0;
double salary = 0.0;
DataSource remoteDS;
Context ic;

//Retrieve the initial context. No JNDI properties are necessary here
ic = new InitialContext ();

//Look up the DataSource using the <resource-ref> definition
```

```
remoteDS = (DataSource)ic.lookup ("java:comp/env/jdbc/OracleMappedDS");

//Retrieve a connection to the database represented by this DataSource
Connection remoteConn = remoteDS.getConnection ("SCOTT", "TIGER");
// Use remoteDS.getConnection () if the data source definition contains
// the user name and password that you want

//Since this implementation uses SQLJ, create a default context for this
//connection.
DefaultContext dc = new DefaultContext (remoteConn);

//Perform the SQL statement against the database, specifying the default
//context for the database in brackets after the #sql statement.
#sql [dc] { select empno, sal  from emp where ename = :name  };
```

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 7–2 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 7–2 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 7–1). 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, the bean uses the invoker's transaction. If the invoker is not involved in a transaction, 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, the invoker's transaction context is used by the bean. If the invoker is not involved in a transaction, 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 `RemoteException` 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 7–3 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 7–3 <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 of 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 and 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.

The following code example assumes that there are no errors. You can download this example from the OC4J sample code OTN site:

`http://otn.oracle.com/sample_code/tech/java/oc4j/htdocs/oc4jsamplecode/oc4j-demo-ejb.html`

This example demonstrates the combination of demarcating a transaction and enlisting the database resources in the following manner:

1. It retrieves the `UserTransaction` object from the bean context.
2. It starts the transaction with the `begin` method.
3. It enlists the database.

This example is the same as in Example 7–1 on page 7-5, but it is surrounded by `UserTransaction begin()` and `commit()` methods.

```
DataSource remoteDS;
Context ic;
```

```
int empno = 0;
double salary = 0.0;
//Retrieve the UserTransaction object. Its methods are used for txn demarcation
UserTransaction ut = ctx.getUserTransaction ();

//Start the transaction
ut.begin();

//Retrieve the initial context. No JNDI properties are necessary here
ic = new InitialContext ();

//Lookup the OrionCMTDataSource that was specified in the data-sources.xml
remoteDS = (DataSource)ic.lookup ("java:comp/env/jdbc/OracleCMTDS");

//Retrieve a connection to the database represented by this DataSource
Connection remoteConn = remoteDS.getConnection ("SCOTT", "TIGER");

//Since this implementation uses SQLJ, create a default context for this
//connection.
DefaultContext dc = new DefaultContext (remoteConn);

//Perform the SQL statement against the database, specifying the default
//context for the database in brackets after the #sql statement.
#sql [dc] { select empno, sal  from emp where ename = :name  };

//Assuming everything went well, commit the transaction.
ut.commit();
```

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 "Non-emulated Data Sources" on page 4-5.

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 going through the steps for configuring your JTA two-phase commit environment.

Configuring Two-Phase Commit Engine

When a global transaction involves 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 Oracle9*i* 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 Oracle9*i* 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 opens 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:

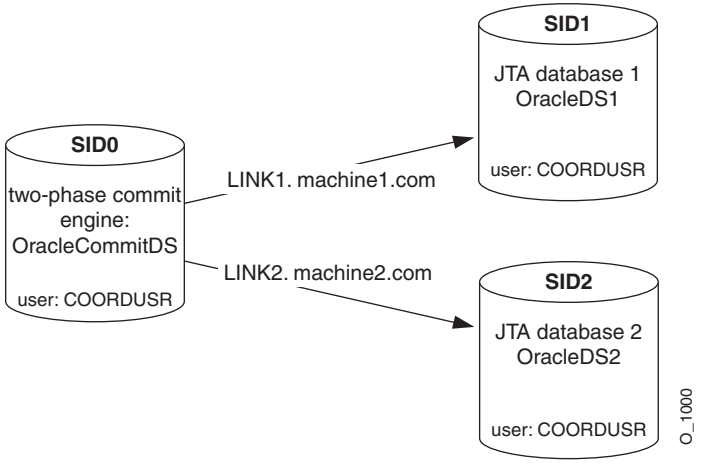
```
CONNECT SYSTEM/MANAGER;
CREATE USER COORDUSR IDENTIFIED BY COORDUSR;
GRANT CONNECT, RESOURCE, CREATE SESSION TO COORDUSR;
```

```
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 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, when you have defined the database that is to act as the two-phase commit engine, configure it as follows:

   a. 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:@machine0:port0:SID0"
  inactivity-timeout="30"
/>
```

b. Refer to the two-phase commit engine data source in either the global `application.xml` file or local `orion-application.xml` file. The global XML file exists in the `config` directory. The local XML file exists in the application EAR file.

Configure the two-phase commit engine in the `orion-application.xml` file 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.

Note: The container prioritizes the user name and password that is defined in the orion-application.xml file over the user name and password that is defined in the data-sources.xml file.

---

* Specify the <commit-class>. This class is always OracleTwoPhaseCommitDriver for two-phase commit engines.

The two-phase commit coordinator can be specified at the application level by defining the <commit-coordinator> element in the application.xml file.

* The OracleTwoPhaseCommitDriver class is defined in the <commit-class> element.
* 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 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:@machine1:port1:SID1""
  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:@machine2:port2:SID2""
  inactivity-timeout="30">
  <property name="dblink"
   value="LINK2.machine2.COM"/>
</data-source>
```

Note: If you change the two-phase commit engine, 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 7-3 for more information.

Limitations of Two-Phase Commit Engine

The following `data-sources.xml` configuration is supported for two-phase commit in the Oracle Application Server Containers for J2EE (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:@hostname:port number:SID"
/>
```

Two-phase commit works only with a nonemulated data source configuration, as shown in the preceding. The URLs of all participating nonemulated data sources must point to an Oracle database instance. Only multiple Oracle resources participating in a global transaction will 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 seem to work, but is not supported, as there is no recovery. The ACID properties of the transaction will not be guaranteed and may cause problems for an application.

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.setTransactionTimeout(`*`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 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 trying to accomplish 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, 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. Thus, when they receive and 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, 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 you should 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 `REQUIRED` and `NOT_SUPPORTED`. If you specify another attribute, such as `SUPPORTS`, then the default of `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, 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 a 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 it not supported.

If you have no two-phase commit support, then there is no guarantee that when a transaction commits, that 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. Since 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 trying to accomplish 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 incurred. 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 find out if the exception indicates an error that you can retry. If this method returns true, then you should 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`. Also, 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 JMSException)
 {
  JMSException  jmsexc = (JMSException) 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
    // 1a. Look up the Queue Connection Factory
    QueueConnectionFactory qcf = (QueueConnectionFactory)
              ctx.lookup ("java:comp/resource/" + resProvider +
              "/QueueConnectionFactories/myQCF");
    // 1b. Lookup the Queue
    Queue queue = (Queue) ctx.lookup ("java:comp/resource/" + resProvider +
                                      "/Queues/rpTestQueue");

    //2 & 3. Retrieve a connection and a session on top of the connection
    //2a. Create queue connection using the connection factory.
    QueueConnection qconn = qcf.createQueueConnection();
    //2a. We're receiving msgs, so start the connection.
    qconn.start();

    // 3. create a session over the queue connection.
    QueueSession qsess = qconn.createQueueSession(false,
                                                  Session.AUTO_ACKNOWLEDGE);

     //4. 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);

  }
 }
}
```

8

J2EE Connector Architecture

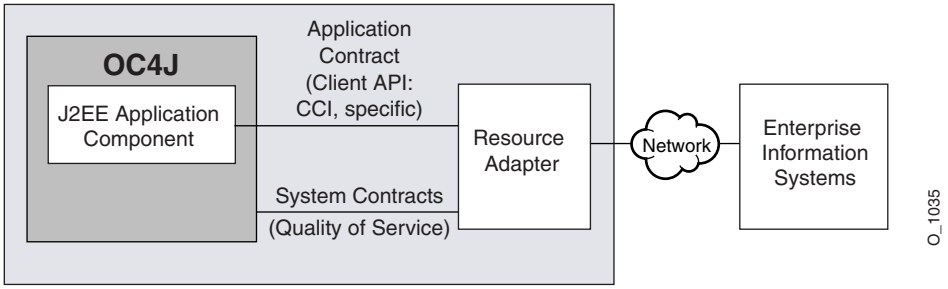
This chapter describes how to use the J2EE Connector Architecture (JCA) 1.0 in an Oracle Application Server Containers for J2EE (OC4J) application. This chapter covers the following topics:

- 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 8-3 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 8-14.

  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 8-15.

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 8-14.

- 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 8-15.

- 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 10g 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 10g 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 8-8).
- `-grantAllPermissions`: grants all runtime permissions requested within the RAR file (see also the `<security-permission>` element in "The oc4j-connectors.xml Descriptor" on page 8-8).

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.

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 10g 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
connectors_dir	connector-directory	The root directory for all standalone resource adapters.	connectors
applications_dir	applications-directory	The root directory for all applications.	applications
application_deployments_dir	deployment-directory	The root directory for all files generated at deployment.	application-deployments

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.

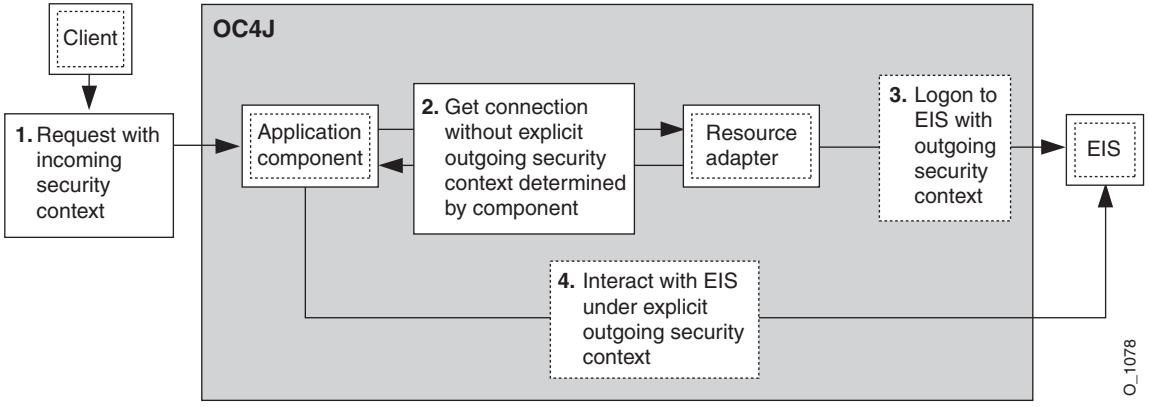
> **Note:** The remainder of this section assumes that you are familiar with chapter 7 of the J2EE Connector Architecture 1.0 specification. The specification uses the terms *initiating principal*, *caller principal,* and *resource principal*. As used in this section, the incoming security context refers 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:

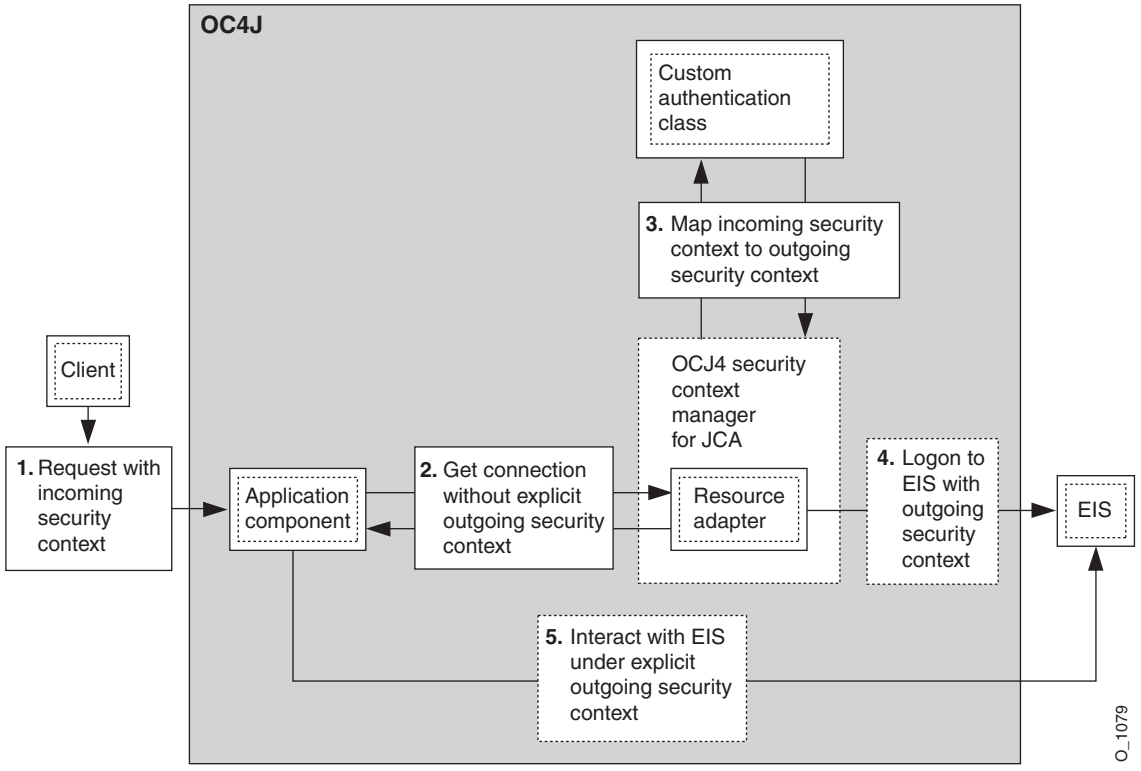
```
Context initctx = new InitialContext();
// perform JNDI lookup to obtain a connection factory
javax.resource.cci.ConnectionFactory cxf =
        (javax.resource.cci.ConnectionFactory)initctx.lookup("java:com/env/eis/MyEIS");

// 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
`public void init(java.util.Properties prop)`	Called by OC4J to initialize the settings for the `PrincipalMapping` implementation class. OC4J passes the properties specified in the `<config-property>` elements in `oc4j-ra.xml` to this method. The implementation class can use the properties for setting default user name and password, LDAP connection information, or default mapping.
`public void setManagedConnectionFactory (ManagedConnectionFactory mcf)`	Used by OC4J to provide the implementation class with the `ManagedConnectionFactory` instance that is needed to create a `PasswordCredential`.
`public void setAuthenticationMechanisms (java.util.Map authMechanisms)`	Called by OC4J to pass the authentication mechanisms supported by the resource adapter to the `PrincipalMapping` implementation class. The key of the map passed is a string containing the supported mechanism type, such as `BasicPassword` or `Kerbv5`. The value is a string containing the corresponding credentials interface as declared in `ra.xml`, such as `javax.resource.spi.security.PasswordCredential`. The map can contain multiple elements if the resource adapter supports multiple authentication mechanisms.
`public javax.security.auth.Subject mapping (javax.security.auth.Subject initiatingSubject)`	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 JCA 1.0 specifications. The implementation may return null 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 (Kerbv5) 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
`public abstract void init (java.util.Properties prop)`	This method must be implemented by the subclasses. See `PrincipalMapping` interface, described in Table 8–3, for details.
`public void setManagedConnectionFactory (ManagedConnectionFactory mcf)`	Stores the `ManagedConnectionFactory` instance that is passed in. Subclasses need not implement this method, and can make use of the `getManagedConnectionFactory` object saved by this method.
`public void setAuthenticationMechanisms (java.util.Map authMechanisms)`	Stores the map of authentication mechanisms. Subclasses need not implement this mechanism. Instead, they can make use of the `isBasicPasswordSupported` or `isKerbv5Supported` methods to determine which authentication mechanism is supported by the resource adapter. The method `getAuthenticationMechanisms` can be used to retrieve the authentication mechanisms as well.
`public javax.security.auth.Subject mapping (javax.security.auth. Subject initiatingSubject)`	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.
`public abstract javax.security.auth.Subject mapping (javax.security.auth. Subject initiatingSubject)`	This method must be implemented by the subclasses. See `PrincipalMapping` interface, described in Table 8–3, for details.
`public ManagedConnectionFactory getManagedConnectionFactory()`	Utility method provided by this abstract class to return the `ManagedConnectionFactory` instance that might be required to create a `PasswordCredentials` object.

Table 8–4 Method Description for oracle.j2ee.connector.AbstractPrincipalMapping Class

Method Signature	Description
`public java.util.Map getAuthenticationMechanisms()`	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 `BasicPassword` or `Kerbv5`. The value is a string containing the corresponding credentials interface as declared in `ra.xml`, such as `javax.resource.spi.security.PasswordCredential`.
`public boolean isBasicPasswordSupported()`	Utility method to allow subclass to determine whether the `BasicPassword` authentication mechanism is supported by this resource adapter.
`public boolean isKerbv5Supported()`	Utility method to allow subclass to determine whether the `Kerbv5` authentication mechanism is supported by this resource adapter.
`public java.security.Principal getPrincipal (javax.security.auth. Subject subject)`	Utility method provided to extract the `Principal` 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 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 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.

9

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 10*g* 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 10*g*.

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 10*g* 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.

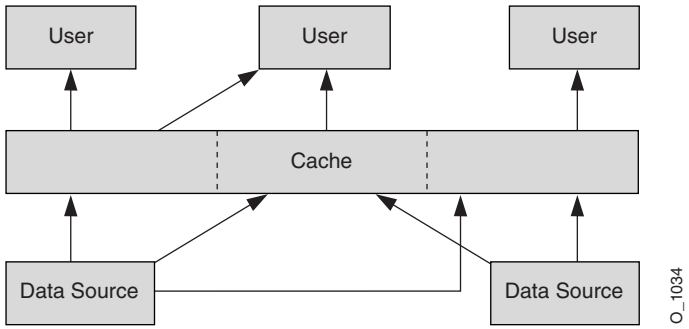
The Java Object Cache is an in-process, process-wide caching service for general application use. That is, objects are cached within the process memory space and the Java Object Cache is a single service that is shared by all threads running in the process, as opposed to a service that runs in another 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 doesn't 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, the cache interacts via the CacheLoader with the data source. This gives a clean division between object creation and object use.

Figure 9–1 Java Object Cache Basic Architecture



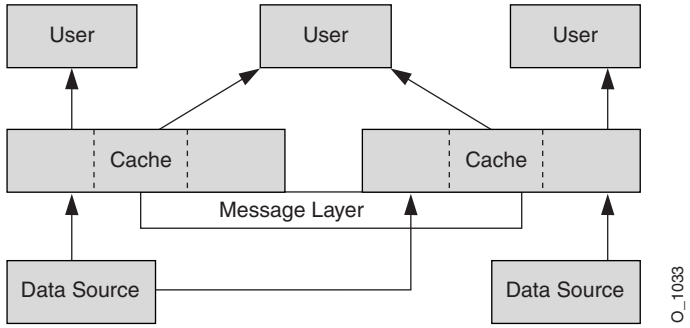
Distributed Object Management

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 process. 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, it is also updated on 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

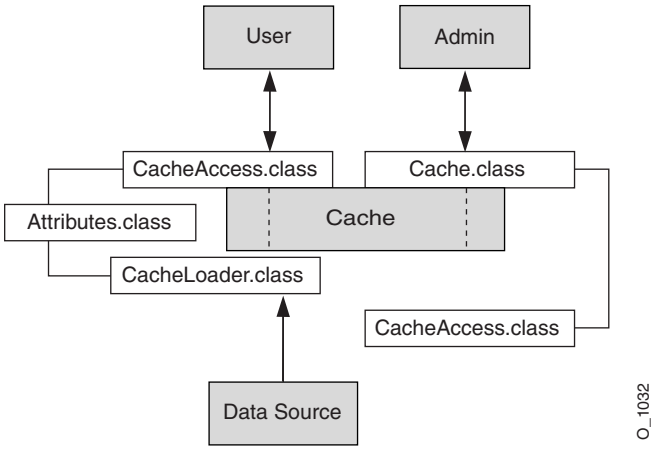
The Java Object Cache manages Java objects within a process, across processes, and on a local disk. The Java Object Cache provides a powerful, flexible, and easy-to-use service that significantly improves Java performance by managing local copies of Java objects. There are very few restrictions on the 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 shows 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.

Table 9–1 Cache Organizational Construct (Cont.)

Cache Construct	Description
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.
`StreamAccess` object	An object that is loaded using a Java `OutputStream` and accessed using a Java `InputStream`. 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, including:

- 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 Oracle9*i* 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

A `StreamAccess` object is a special cache object 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

A *pool object* is a special class of 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. 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 due to 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 need to 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. A group defines 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. 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 10*g* (9.0.4) version of the Java Object Cache, the maximum size of a region or group can be specified 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. The size of a region or group can be set 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, the default policy is used. The default policy follows: If a nonreferenced object of lesser or equal priority is found, it is invalidated in favor of the new object. If the priority attribute has not been set for an object, 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 30 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, 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 30 before the capacity of region A is reached, 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

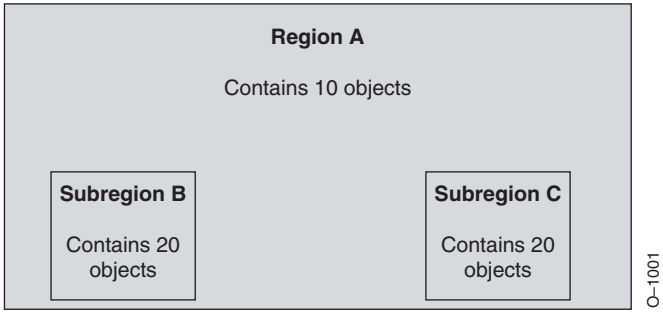
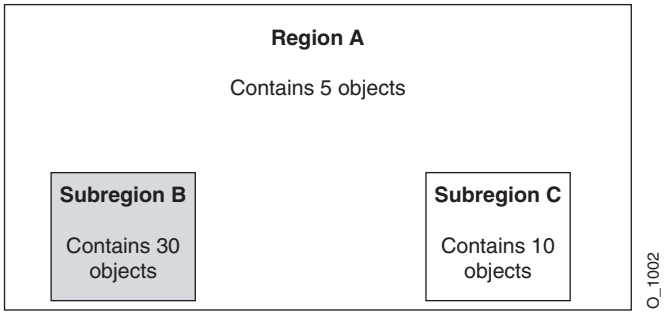


Figure 9–5 Capacity Policy Example, Part 2



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`	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. **Object Types:** When set on a region, subregion, or a group, this attribute sets the default value for the `DISTRIBUTE` attribute for the objects within the region, subregion, or group, unless the objects explicitly set their own `DISTRIBUTE` attribute. Because pool objects are always local, this attribute does not apply to pool objects. **Default Value:** All objects are local.
`GROUP_TTL_DESTROY`	Indicates that the associated object, group, or region should be destroyed when the `TimeToLive` expires. **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 `TimeToLive` expires. **Default Value:** Only group member objects are invalidated when the `TimeToLive` expires.

Table 9–2 Java Object Cache Attributes–Set at Object Creation (Cont.)

Attribute Name	Description
LOADER	Specifies the CacheLoader associated with the object. **Object Types:** When set on a region or group, the specified CacheLoader becomes the default loader for the region, subregion, or group. The LOADER attribute is specified for each object within the region or the group. **Default Value:** Not set.
ORIGINAL	Indicates that the object was created in the cache, rather than loaded from an external source. ORIGINAL objects are not removed from the cache when the reference count goes to zero. ORIGINAL objects must be explicitly invalidated when they are no longer useful. **Object Types:** When set on a region or group, this attribute sets the default value for the ORIGINAL attribute for the objects within the region, subregion, or group, unless the objects set their own ORIGINAL attribute. **Default Value**: Not set.
REPLY	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 DISTRIBUTE attribute is not set or the cache is started in non-distributed mode, REPLY is ignored. **Object Types:** When set on a region or group, this attribute sets the default value for the REPLY attribute for the objects within the region, subregion, or group, unless the objects explicitly set their own REPLY attribute. For memory, StreamAccess, and disk objects, this attribute applies only when the DISTRIBUTE attribute is set to the value DISTRIBUTE. Because pool objects are always local, this attribute does not apply for pool objects. **Default Value:** No reply is sent. When DISTRIBUTE is set to local the REPLY attribute is ignored.
SPOOL	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. **Note**: An object must be serializable to be spooled. **Object Types:** When set on a region, subregion, or group, this attribute sets the default value for the SPOOL attribute for the objects within the region, subregion, or group, unless the objects set their own SPOOL attribute. **Default Value:** Memory objects are not stored to disk.

Table 9–2 Java Object Cache Attributes–Set at Object Creation (Cont.)

Attribute Name	Description
SYNCHRONIZE	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 `CacheAccess.getOwnership()` method. The "owner" of an object is the `CacheAccess` object. Setting the `SYNCHRONIZE` attribute does not prevent a user from reading or invalidating the object. **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. **Default Value:** Updates are not synchronized.
SYNCHRONIZE_DEFAULT	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 `SYNCHRONIZE` attribute. Ownership of the object must be obtained before the object can be loaded or updated. Setting the `SYNCHRONIZE_DEFAULT` 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 `SYNCHRONIZE` attribute set. **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. **Default Value:** Updates are not synchronized.
ALLOWNULL	Specifies that the cache accepts null as a valid value for the affected objects. Null objects that are returned by a `cacheLoader` object are cached, rather than generating an `ObjectNotFoundException`. **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. **Default Value:** OFF. (Nulls are not allowed.)
MEASURE	Indicates 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. **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. **Default Value:** OFF. (The size of an object is not automatically calculated.)

Table 9–2 Java Object Cache Attributes–Set at Object Creation (Cont.)

Attribute Name	Description
`CapacityPolicy`	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. **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. **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.)

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 9-13, and can be reset using the `CacheAccess.resetAttributes()` method.

Table 9–3 Java Object Cache Attributes

Attribute Name	Description
`DefaultTimeToLive`	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. **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`. **Default Value:** No automatic invalidation.
`IdleTime`	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. **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`. **Default Value:** No automatic `IdleTime` invalidation.

Table 9–3 Java Object Cache Attributes (Cont.)

Attribute Name	Description
`CacheEventListener`	Specifies the `CacheEventListener` associated with the object. **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. **Default Value:** No `CacheEventListener` is set.
`TimeToLive`	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 `GROUP_TTL_DESTROY` is not set), the `TimeToLive` value is reset. **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 `TimeToLive`. **Default Value:** No automatic invalidation.
`Version`	An application can set a `Version` for each instance of an object in the cache. The `Version` is available for application convenience and verification. The caching system does not use this attribute. **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 `Version`. **Default Value:** The default `Version` is `0`.
`Priority`	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 `CapacityPolicy` object used to control the size of the cache, region, or group. The larger the number that is, 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. **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. **Default Value:** `integer.MAX_VALUE`.

Table 9–3 Java Object Cache Attributes (Cont.)

Attribute Name	Description
`MaxSize`	Specifies the maximum number of bytes available for a region or group. If this attribute is specified for an object, it is ignored. **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. **Default Value:** No limit.
`MaxCount`	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. **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. **Default Value:** No limit.
User-defined attributes	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 `CapacityPolicy` object, although they can be defined as needed by the cache user. **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. **Default Value:** No user-defined attributes are set by default.

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:

- Importing Java Object Cache
- Defining a Cache Region
- 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. A `CacheAccess` object 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.

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
caccess.defineGroup(GROUP_NAME_);
// Close the CacheAccess object
caccess.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
caccess.defineSubRegion(SUBREGION_NAME_);
// Close the CacheAccess object
caccess.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 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 or 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 request. 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
`setAttributes()`	Sets the attributes for the object being loaded.
`netSearch()`	Searches other available caches for the object to load. Objects are uniquely identified by the region name, subregion name, and the object name.
`getName()`	Returns the name of the object being loaded.
`getRegion()`	Returns the name of the region associated with the object being loaded.
`createStream()`	Creates a `StreamAccess` object.
`createDiskObject()`	Creates a disk object.
`exceptionHandler()`	Converts noncache exceptions into `CacheExceptions`, with the base set to the original exception.
`log()`	Records messages in the cache service log.

Example 9–5 shows a `CacheLoader` object using the `cacheLoader.netSearch()` method to check if 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 might involve an HTTP connection to a remote Web site or a connection to the Oracle9*i* 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 scnds
            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, `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, 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

By default, the Java Object Cache is initialized automatically upon OC4J startup. The OC4J runtime automatically 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 previous versions of Java Object Cache, configuration was done through the file `javacache.properties`; the 10*g* (9.0.4) release uses `javacache.xml`.

Note: If you install both a 10*g* (9.0.4) release and a pre-10*g* (9.0.4) release on the same host, you must ensure that the javacache.xml discovery-port attribute and javacache.properties discoveryAddress attribute are not configured to the same port. If they are, 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>
    <coordinator 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 `TimeToLive` or `IdleTime` attributes that are associated with the object. (Table 9–3 describes these attributes are described.) **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

Table 9–5 Java Object Cache Configuration Properties (Cont.)

Configuration XML Element	Description	Type
`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 9-32. **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 (`cache.xsd`). 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 9-34. Also see "Examples" on page 9-32. **Note**: The file path of the declarative cache XML schema is *`ORACLE_HOME`*`/javacache/admin/cache.xsd`. Refer to the XML schema when writing a declarative cache file. **Default value:** Not use a declarative cache.	`String`
`communication`	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. Updates and invalidation for objects that have the `distribute` property set are propagated to other caches known to the Java Object Cache. If the `isDistributed` subelement of the `communication` element is set to `false`, all objects are treated as local, even when the attributes set on objects are set to distribute. See "Examples" on page 9-32. **Default value:** Cache is not distributed (`isDistributed` subelement set to `false`).	complex (has subelements)

Table 9–5 Java Object Cache Configuration Properties (Cont.)

Configuration XML Element	Description	Type
`logging`	Specifies the logger attributes such as log file name and log level. The available options of the log level are `OFF`, `FATAL`, `ERROR`, `DEFAULT`, `WARNING`, `TRACE`, `INFO`, and `DEBUG`. See "Examples" on page 9-32. **Default values:** **Log file name:** `$ORACLE_HOME/javacache/admin/logs/javacache.log` on UNIX or `%ORACLE_HOME%\javacache\admin\logs\javacache.log` on Windows **Log level:** `DEFAULT`	complex (has subelements)
`persistence`	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 10MB. The unit of the disk cache size is megabytes. See "Examples" on page 9-32. **Default value:** Disk caching is not available.	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 shows the use of the `<region-name-separator>` element:

- Set the separator to be `_S_`:

```
<region-name-separator>_S_</region-name-separator>
```

The following example shows 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 show 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 machine:

```
<communication>
  <isDistributed>true</isDistributed>
  <coordinator 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>
<coordinator 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>
  <coordinator ip="192.10.10.10" discovery-port="7000">
  <coordinator ip="192.11.11.11" discovery-port="7000">
  <coordinator ip="192.22.22.22" discovery-port="7000">
  <coordinator ip="192.22.22.22" discovery-port="8000">
</communication>
```

The following examples show 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 20MB:

```
<persistence>
  <location>/app/9iAS/javacache/admin/diskcache</location>
  <disksize>20</disksize>
</persistence>
```

The following examples show 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>
```

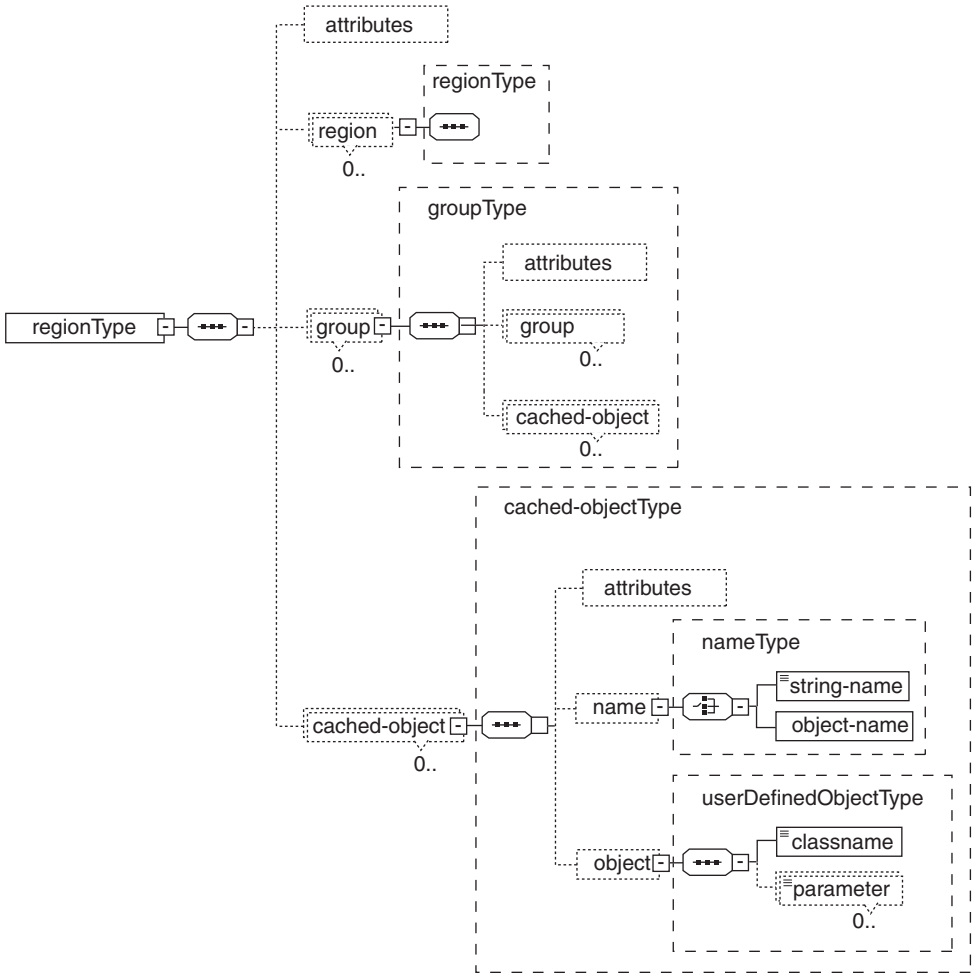
Declarative Cache

With the 10*g* (9.0.4) 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 9-69 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 for automatically loading 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 10*g*. 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 9-40 for code that shows usage for most elements.

Table 9–6 Description of Declarative Cache Schema (cache.xsd)

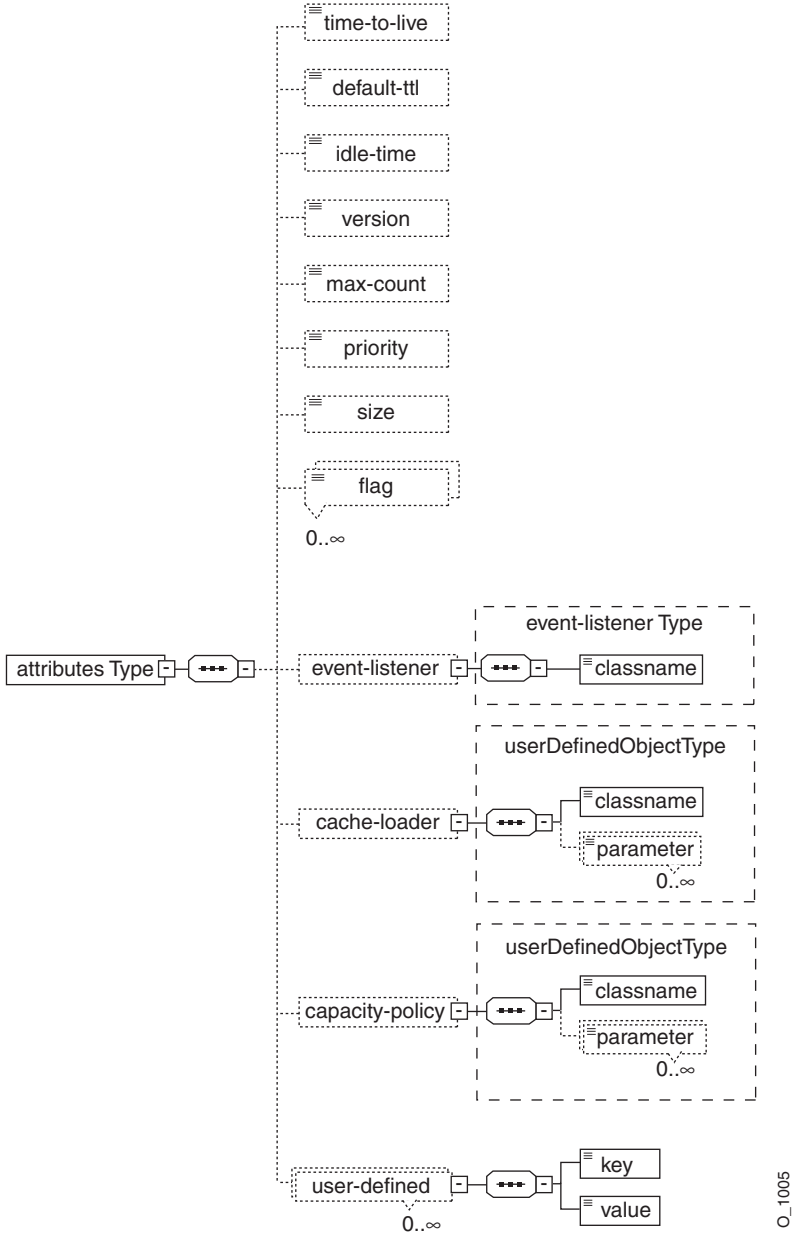
Element	Description	Children	Type
region	Declare a cache region or subregions.	<attributes> <region> <group> <cached-object>	regionType
group	Declare a cache group or subgroup.	<attributes> <group> <cached-object>	groupType
cached-object	Declare a cache object.	<attributes> <name> <object>	objectType
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

Table 9–6 Description of Declarative Cache Schema (cache.xsd) (Cont.)

Element	Description	Children	Type
object	Declare a user-defined Java object. The class of the specified object must implement the declarable interface of the `oracle.ias.cache` 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 `oracle.ias.cache` package. See the Javadoc of `Attributes.class` 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 `CacheEventListener` object.	<classname>	event-listenerType
cache-loader	Declare a `CacheLoader` object.	<classname> <parameter>	userDefinedObjectType
capacity-policy	Declare a `CapacityPolicy` 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, such 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 `cache.jar`

- `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`

- `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 reach, 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, 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, as 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 to keep in mind when using the Java Object Cache.

- The `CacheAccess` object should not be shared 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 can result in nondeterministic behavior.
- A `CacheAccess` object holds a reference to only one cached object at a time. If multiple cached objects are being accessed concurrently, then multiple `CacheAccess` objects should be used. 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, you should keep the cache reference open as long as the cached object is being used.
- A `CacheAccess` object should always be closed 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 can 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 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. The cache configuration should be defined during the initialization of the application.

- If a `CacheAccess.waitForResponse()` or `CacheAccess.releaseOwnership()` method call times out, then it must be called again until it returns successfully. If `CacheAccess.waitForResponse()` doesn't succeed, then `CacheAccess.cancelResponse` must be called to free resources. If `CacheAccess.releaseOwnership()` doesn't succeed, then `CacheAccess.releaseOwnership` with a timeout value of -1 must be called 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, 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, 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 need to 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, the write is done 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 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 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, 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

Creating Pool Objects

To create a pool object, use `CacheAccess.createPool()`. The `CreatePool()` method takes as arguments a `PoolInstanceFactory`, and an `Attributes` object, plus 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, 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, the new user is not given access to the object. The interface `PoolAffinityFactory` extends the interface `PoolInstanceFactory`, adding the callback method

`affinityRelease`. This method should return `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 should then be done 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

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 9-29 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 `coordinator` 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 `coordinator` 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 coordinator 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 9-29.

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 coordinator 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 9-67.

Using the REPLY Attribute with Distributed Objects

When updating, invalidating, or destroying objects across multiple caches, it might 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 allows the application to update a number of 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 also 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 of 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 should call `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 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, should guarantee 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 applications 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 to only 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.

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`.

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="region-name-separator" type="xs:string" 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: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:sequence>
  </xs:complexType>
  <xs:complexType name="communicationType">
    <xs:sequence>
      <xs:element name="isDistributed" type="xs:boolean" default="false"
minOccurs="0"/>
      <xs:element name="coordinator" type="coordinatorType" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="coordinatorType">
```

```
    <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:schema>
```

XML schema for attribute declaration

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
targetNamespace="http://www.oracle.com/oracle/ias/cache/configuration/declarativ
e" 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="CacheType">
    <xs:sequence maxOccurs="unbounded">
      <xs:element name="region" type="regionType"/>
    </xs:sequence>
  </xs:complexType>
  <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: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:schema>
```

Index

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X