Oracle® Containers for J2EE

Services Guide

10*g* Release 3 (10.1.3)

B14427-01

January 2006

Oracle Containers for J2EE Services Guide, 10*g* Release 3 (10.1.3)

B14427-01

Primary Author: Alfred Franci

Contributing Author: Bonnie Vaughan, Brian Wright, Dan Hynes, Frances Zhao, Jerry Bortvedt, Kirk Bittler, Tom Collier, Qiang Liu, Vivekananda Maganty, Bob Nettleton, Shengsong Ni, Debabrata Panda, Paul Parkinson, J. J. Snyder, John Speidel, Jerry Steidl.

Contributor: John Lang, Rajkumar Irudayaraj, Lixin Zheng, Peter Wu, Valarie Moore, Anthony Lai, Ashok Banerjee, Cheuk Chau, Editor Ellen Siegal, Erik Bergenholtz, Gary Gilchrist, Irene Zhang, Jon Currey, Jyotsna Laxminarayanan, Krishna Kunchithapadam, Kuassi Mensah, Lars Ewe, Lelia Yin, Mike Lehmann, Mike Sanko, Min-Hank Ho, Nickolas Kavantzas, Olga Peschansky, Rachel Chan, Raymond Ng, Sastry Malladi, Stella Li, Sunil Kunisetty, Thomas Van Raalte.

Contents

Preface

Oracle Application Server 10*g* Release 3 (10.1.3) includes a J2EE environment known as Oracle Containers for J2EE (OC4J). This book describes the services provided by OC4J.

This preface contains these topics:

- Intended Audience
- Documentation Accessibility
- Related Documents
- Conventions

Intended Audience

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

Documentation Accessibility

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

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

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

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

TTY Access to Oracle Support Services

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

Related Documents

For more information, see the following Oracle resources.

Additional OC4J documents:

- *Oracle Containers for J2EE Developer's Guide*

  This discusses items of general interest to developers writing an application to run on OC4J—issues that are not specific to a particular container such as the servlet, EJB, or JSP container. (An example is class loading.)

- *Oracle Containers for J2EE Deployment Guide*

  This covers information and procedures for deploying an application to an OC4J environment. This includes discussion of the deployment plan editor that comes with Oracle Enterprise Manager 10*g*.

- *Oracle Containers for J2EE Configuration and Administration Guide*

  This discusses how to configure and administer applications for OC4J, including use of the Oracle Enterprise Manager 10*g* Application Server Control Console, use of standards-compliant MBeans provided with OC4J, and, where appropriate, direct use of OC4J-specific XML configuration files.

- *Oracle Containers for J2EE Servlet Developer's Guide*

  This provides information for servlet developers regarding use of servlets and the servlet container in OC4J, including basic servlet development and use of JDBC and EJBs.

- *Oracle Containers for J2EE Support for JavaServer Pages Developer's Guide*

  This book provides information about JavaServer Pages development and the JSP implementation and container in OC4J. This includes discussion of Oracle features such as the command-line translator and OC4J-specific configuration parameters.

- *Oracle Containers for J2EE JSP Tag Libraries and Utilities Reference*

  This book provides conceptual information as well as detailed syntax and usage information for tag libraries, JavaBeans, and other Java utilities provided with OC4J. There is also a summary of tag libraries from other Oracle product groups.

- *Oracle Containers for J2EE Services Guide*

  This book provides information about standards-based Java services supplied with OC4J, such as JTA, JNDI, JMS, JAAS, and the Oracle Application Server Java Object Cache.

- *Oracle Containers for J2EE Security Guide*

  This document (not to be confused with the *Oracle Application Server Security Guide*) describes security features and implementations particular to OC4J. This includes information about using JAAS, the Java Authentication and Authorization Service, as well as other Java security technologies.

- *Oracle Containers for J2EE Enterprise JavaBeans Developer's Guide*

This book provides information about Enterprise JavaBeans development and the EJB implementation and container in OC4J.

- *Oracle Containers for J2EE Resource Adapter Administrator's Guide*

  This book provides an overview of J2EE Connector Architecture features and describes how to configure and monitor resource adapters in OC4J.

- *Oracle Application Server Web Services Developer's Guide*

  This book describes Web services development and configuration in OC4J.

- *Oracle Application Server Advanced Web Services Developer's Guide*

  This book describes topics beyond basic Web service assembly. For example, it describes how to diagnose common interoperability problems, how to enable Web service management features (such as reliability, auditing, and logging), and how to use custom serialization of Java value types.

  This book also describes how to employ the Web Service Invocation Framework (WSIF), the Web Service Provider API, message attachments, and management features (reliability, logging, and auditing). It also describes alternative Web service strategies, such as using JMS as a transport mechanism.

Oracle TopLink documents:

- *Oracle TopLink Getting Started Guide*
- *Oracle Application Server TopLink Mapping Workbench User's Guide*
- *Oracle TopLink Developer's Guide*

Java-related documents for Oracle Database:

- *Oracle Database Java Developer's Guide*
- *Oracle Database JDBC Developer's Guide and Reference*
- *Oracle Database JPublisher User's Guide*

Additional Oracle Application Server documents:

- *Oracle Application Server Administrator's Guide*
- *Oracle Application Server Security Guide*
- *Oracle Application Server Certificate Authority Administrator's Guide*
- *Oracle Application Server Performance Guide*
- *Oracle HTTP Server Administrator's Guide*
- *Oracle Process Manager and Notification Server Administrator's Guide*
- *Oracle Application Server Globalization Guide*
- *Oracle Application Server Web Cache Administrator's Guide*
- *Oracle Application Server Upgrade and Compatibility Guide*

Oracle JDeveloper documentation:

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

  `http://www.oracle.com/technology/products/jdev/index.html`

Additional Oracle Database documents:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle Database Application Developer's Guide - Fundamentals*
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Database PL/SQL User's Guide and Reference*
- *Oracle Database SQL Reference*
- *Oracle Database Net Services Administrator's Guide*
- *Oracle Database Advanced Security Administrator's Guide*
- *Oracle Database Reference*

The following Web site for Java servlets and JavaServer Pages is also available:

`http://www.oracle.com/technology/tech/java/servlets/index.html`

For further servlet information, refer to the *Java Servlet Specification* at the following location:

`http://java.sun.com/products/servlet/download.html#specs`

Resources from Sun Microsystems:

- Web site for Java servlet technology:

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

- Web site for JavaServer Pages technology:

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

- J2EE 1.4 Javadoc, including the servlet packages `javax.servlet` and `javax.servlet.http`:

  `http://java.sun.com/j2ee/1.4/docs/api/index.html`

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1 Introduction to OC4J Services

Oracle 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)
- Using Remote Method Invocation in OC4J
- Data Sources
- OC4J Transaction Support J
- Java Object Cache
- XML Query Service

This chapter gives a brief overview of each technology in the preceding list and a link to the relevant chapter.

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

Java Naming and Directory Interface (JNDI)

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

See Chapter 2, "Oracle JNDI", 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.

In past releases, Oracle has used the terms "OracleAS JMS" and "OJMS" when describing the In-Memory, File-Based, Database persistence options. "OracleAS JMS" referred to the In-Memory and File-Based options while "OJMS" referred to JMS interface to Streams Advanced Queuing (AQ). To avoid any confusion regarding JMS, the term "OEMS JMS" is used instead of the terms "OracleAS JMS" and "OJMS". This reflects the fact that Oracle offers a single JMS interface to the three message persistence options. Your JMS application code will not have to change if you decide to change message persistence between any of the three quality of service choices.

See Chapter 3, "Oracle Enterprise Messaging Service (OEMS)", for details.

Data Sources

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

See Chapter 4, "Data Sources", for details.

OC4J Transaction Support J

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

See Chapter 5, "OC4J Transaction Support", for details.

Using Remote Method Invocation in OC4J

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

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

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

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

For details on RMI/ORMI and interoperability (RMI/IIOP), see Chapter 6, "Using Remote Method Invocation in OC4J".

Java Object Cache

The Java Object Cache (formerly OCS4J) is a set of Java classes that manage Java objects within a process, across processes, and on a local disk. The primary goal of the Java Object Cache is to provide a powerful, flexible, easy-to-use service that significantly improves server performance by managing local copies of objects that are expensive to retrieve or create. There are no restrictions on the type of object that can be cached or the original source of the object. The management of each object in the cache is easily customized. Each object has a set of attributes associated with it to control such things as how the object is loaded into the cache, where the object is stored (in memory, on disk, or both), how it is invalidated (based on time or by explicit request), and who should be notified when the object is invalidated. Objects can be invalidated as a group or individually. See Chapter 7, "Java Object Cache", for details.

XML Query Service

XML Query Service (XQS), a new service in the OC4J 10.1.3 implementation built upon XQuery (the XML query language) that provides a convenient user model for the retrieval, analysis, integration, and transformation of enterprise data. Generally, without a service such as XQS, XQuery is limited to accessing XML documents. With XQS, you can also retrieve data from non-XML documents, relational databases, and other possibly non-XML enterprise information systems, through access mechanisms such as SOAP or SQL. See Chapter 8, "XML Query Service", for details.

Third-Party Licenses

This appendix lists the licensing agreements that pertain to the third-party products included with Oracle Application Server. See Appendix A, "Third Party Licenses", for details.

2 Oracle JNDI

This chapter describes the Java Naming and Directory Interface (JNDI) service as implemented by Oracle Containers for J2EE (OC4J). The chapter focuses on setting up the initial contexts for using JNDI and describing how to perform JNDI lookups.

This chapter covers the following topics:

- What You Need To Know About Oracle JNDI
- Configuring JNDI for Deployment
- Browsing the JNDI Context
- Looking Up Objects from J2EE Application Components
- Looking Up Objects from J2EE Application Clients
- JNDI State Replication

To use the information in 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`

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`

JNDI Tasks

This chapter discusses the following common JNDI tasks:

- Configuring JNDI for Deployment
- Enabling JNDI State Replication
- Browsing the JNDI Context
- Constructing a JNDI Context
- Creating JNDI bindings and using JNDI to look up bindings, as described in the following sections:
  - Looking Up Objects from J2EE Application Components
  - Looking Up Objects from J2EE Application Clients

What's New for 10.1.3

The following OC4J JNDI features and behaviors are new for this release:

- **JNDI Tree Browser** - The Application Server Control Console provides a JNDI tree browser as described in Browsing the JNDI Context.
- **Global JNDI Lookup** - In the default OC4J configuration, lookups within an application are bound to be available within the current application's namespace. It is now possible to configure JNDI to perform inter-application lookups. Note that for global lookup to work properly, the target application's classes must be in the classpath of the application attempting the lookup. The simplest way to achieve this is to copy the target EJB jar into the `J2EE_HOME/applib` directory. This will cause OC4J to load this jar as a system library that is imported by all deployed applications. If finer-grained control of system libraries is desired, it is also possible to deploy the target classes as a named system library, and then explicitly import these into each application.

  Please consult the OC4J documentation on configuring a system library for the steps required to accomplish this. For configuration instructions, see "Configuring JNDI for Deployment" on page 2-3.
- **Relative Java Context Lookups** - Relative lookups are now attempted when all other lookup attempts fail. For example, if a client looks up its resource using `context.lookup("ejb/EJBName")`, rather than (`"java:comp/env/ejb/EJBName"`), this lookup will now succeed. This lookup attempt is always relative to the "`java:comp/env`" namespace, and has the same effect as calling `context.lookup()` with the "`java:comp/env`" prefix attached to the binding name. There is no configuration required for this feature, since this feature is enabled by default in OC4J.
- **Associating a Principal with a Thread**

  For work that requires authentication or access control, you can now associate a principal, or user, with the thread that performs the work. This is described in "JNDI Contexts and Threads" on page 2-7.
- **MBean Support for JNDI** - The following JNDI-related MBeans are now registered with OC4J and are available for use within the Oracle Enterprise Manager 10*g* Application Server Control Console:
  - `JNDIResource` - This JSR-77 MBean allows for queries on the JNDI bindings of a given application. A JNDIResource MBean is registered for each deployed application in a given OC4J instance. When an application is undeployed, the associated JNDIResource MBean is deregistered. This MBean offers two ways to access the JNDI bindings for a given application:
    - `getBindingsAsXMLString()` - returns the bindings tree as an XML document.
    - `getBindingsAsString()` - used mainly for debugging.
  - `JNDINamespace` - This MBean allows a client to iterate over all of the JNDIResource MBeans registered on a given OC4J instance. This means that this namespace bean can be used to view all of the JNDI bindings for a given application server instance, separated by application name. The method `getAllBindingsAsXMLString()` returns an XML document that represents the entire bindings tree.

  Both MBeans return XML documents that can be parsed to determine the current bindings of the applications deployed to an OC4J instance. This XML document uses the `oc4j-jndi-bindings-10_0.xsd` schema for validation.

  Path to JNDI-Related MBeans from the Application Server Control Page

OC4J:Home > Administration tab > Task Name.System MBean Browser. Go To Task > Drill down: J2EEDomain:oc4j, J2EEServer:standalone > JNDI Namespace and JNDI Resource

- **Deprecated**

  The following item is deprecated in this release:

  – The previous package structure for context factories provided by OC4J is deprecated, and is replaced by a more consistent naming structure. See the note below Table 2–1, " InitialContext Properties" on page 2-6.

- **No Longer Supported**

  The following items are no longer supported in this release:

  – The `dedicated.connection` environment property is no longer supported as of release 10.1.3.

  – The `dedicated.rmicontext` environment property is no longer supported as of release 10.1.3. See Table 2–2, " JNDI-Related Environment Properties" on page 2-12.

Additional Documentation

The How-To documents at the following site provide information about OC4J 10g Release 3 (10.1.3) features, including feature overviews and code excerpts relevant to the feature.

`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`

What You Need To Know About Oracle JNDI

The Java Naming and Directory Interface (JNDI) is a core part of the J2EE specification. JNDI provides naming and directory functionality for J2EE applications and components. JNDI is defined independently of any specific naming or directory service implementation. This enables J2EE applications and components 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.

A J2EE-compatible application uses JNDI to obtain naming contexts. A naming context enables the application to retrieve J2EE resources including data sources, local and remote Enterprise Java Beans (EJBs), and JMS-administered objects such as topics and queues.

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

Configuring JNDI for Deployment

The only configuration necessary for JNDI is to enable global JNDI lookup.

To enable global lookups, make the following configuration changes:
Set the `global-jndi-lookup-enabled` attribute to `true` in `server.xml`. This causes a lookup to be attempted across all known applications in the OC4J instance. If the lookup fails within the current application, the lookup is attempted on each deployed application's context. The first successful lookup of the given name is returned. This feature is disabled by default. When this feature is enabled, each

resource binding name must be unique across all applications deployed in an OC4J instance. When a lookup across all applications is performed, the order of applications is not guaranteed. If two applications in the same instance have a binding of the same name pointing to different objects, then a lookup might return an unexpected object when using this new feature.

For reference documentation of `server.xml`, see *Oracle Containers for J2EE Configuration and Administration Guide* Appendix B - Configuration Files Used in OC4J, Section - "Overview of the OC4J Server Configuration File (server.xml)".

Initial Context

The concept of the initial context is central to JNDI. This section discusses the following topics:

Creating and Using the Initial Context

The two most frequently used JNDI operations in J2EE applications are:

- Creating a new `javax.naming.InitialContext` instance
- Using the `InitialContext` to look up a resource

When OC4J starts up, it constructs a JNDI initial context for each application by reading the resource references in each application's deployment descriptor.

- For lookups from within EJBs, the resource references are specified in `ejb-jar.xml`. Resource references specified in `ejb-jar.xml` are then mapped to actual JNDI locations in `orion-ejb-jar.xml`.
- When a servlet creates an initial context, the JNDI implementation maps bindings to the resource references specified in `web.xml` to the actual JNDI locations specified in `orion-web.xml`.
- An application client's `application-client.xml` descriptor's references are bound when the context is created from a remote application client. These references are bound to the actual JNDI locations specified in `orion-application-client.xml`.

Persistence

After the initial configuration, the JNDI namespace for each application is purely memory-based. Additions made to the context at run time are not persisted.

When OC4J is shut down, any bindings made programmatically (by making a call to `Context.bind`, for example) are no longer available.

Bindings that are specified declaratively through J2EE deployment descriptors are persisted beyond a shutdown of the application server.

Constructing a JNDI Context

Upon initialization, OC4J 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 and bindings from the global application and can override property values defined in the global application, specify new values for properties, and define new properties as required. Lookups in a user application's context are made in the following order:

- First, look in the local application's namespace.

- If the binding is not found locally, then look for the binding in the parent application.
- If the binding is not found locally or in the parent application, then one of the following occurs:
  - If global lookup is enabled, then OC4J will attempt to resolve the lookup over all known application contexts currently deployed in the OC4J instance.
  - If global lookup is not enabled (which is the default behavior), then a `NameNotFoundException` is thrown.

For more information about configuring the OC4J server and its contained applications, see the *Oracle Containers for J2EE Configuration and Administration Guide*.

Environment and Constructors

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

- An environment specified explicitly in a `java.util.Hashtable` instance passed to the JNDI initial context constructor. ("Example: Application Client Looking Up an EJB" on page 2-13 includes an example of this constructor.)
- System property values, as set either by the OC4J server or by the application container.
- A `jndi.properties` file contained in the application EAR file (as part of `application-client.jar`).

The JNDI `InitialContext` has two constructors. You can use either of the following constructors to create the initial context:

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

InitialContext()

The `InitialContext()` constructor creates a `Context` object using the default context environment. If you use this constructor in an OC4J server-side application, then the application context built by OC4J upon startup is returned. This constructor is typically used in code that runs on the server side, such as JSPs, EJBs, and servlets.

InitialContext(Hashtable env)

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

Table 2–1 InitialContext Properties

Property	Value	Meaning
`java.naming.factory.initial`	`INITIAL_CONTEXT_FACTORY`	This property specifies which initial context factory to use when creating a new initial context object. The eligible settings are: ■ `oracle.j2ee.rmi.RMIInitialContextFactory` ■ `oracle.j2ee.naming.ApplicationClientInitialContextFactory` ■ `oracle.j2ee.iiop.IIOPInitialContextFactory` See the Note following this table for information about deprecated context factories.
`java.naming.provider.url`	`PROVIDER_URL`	This property specifies the URL that the client-side JNDI code uses to look up objects on the server. See Table 2–2, " JNDI-Related Environment Properties" on page 2-12 for details.
`java.naming.security.principal`	`SECURITY_PRINCIPAL`	This property specifies the user name for the current security credentials. Required in application client code to authenticate the client. Not required for server-side code, because the authentication has already been performed.
`java.naming.security.credential`	`SECURITY_CREDENTIALS`	This property specifies the password for the current security principal. Required in application client code to authenticate the client. Not required for server-side code, because the authentication has already been performed.

See "Example: Application Client Looking Up an EJB" on page 2-13 for a code example that sets these properties and gets a new JNDI initial context.

Note: The use of the 10*g* package names for OC4J initial context factories is deprecated as of 10.1.3.

The following context factories are deprecated:

- `com.evermind.server.rmi.RMIInitialContextFactory`
- `com.evermind.server.ApplicationClientInitialContextFactory`
- `com.oracle.iiop.server.IIOPInitialContextFactory`

For the new context factory names that replace the deprecated ones, see the `java.naming.factory.initial` item in Table 2–1, " InitialContext Properties".

Example: Looking Up An EJB

This example shows code to use on the server side in a typical Web or EJB application:

```
Context ctx = new InitialContext();
HelloHome 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` is the name of a reference to a session bean that is declared in `web.xml` (if this code executes in a servlet), or in `ejb-jar.xml` (if this code executes in an EJB business method). This reference is defined in an `<ejb-ref>` tag. The EJB reference would then be mapped to an actual JNDI location in either `orion-web.xml` or `orion-ejb-jar.xml`, depending on the caller that is executing this code.

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>
```

JNDI Contexts and Threads

By default, when you create a JNDI Context with a username and password, a principal or user is bound to a context instance when establishing an initial context with OC4J. Thus, the specified principal or user is used during authentication and access control when looking up a named object in the OC4J namespace, obtaining a reference to a remote object, and invoking operations on the remote object. In this scenario, multiple threads within an application client may share the context.

You can also associate a principal or user with the thread that will perform the work. To do so, create a context with the appropriate principal and credential property values as described below. To disassociate a principal from the thread, close the context.

When a thread is associated with a principal, that principal becomes the default for the thread. If any contexts are subsequently created without principal or credential properties, the principal associated with the thread will remain unchanged.

While it is technically possible to share a single context among multiple threads, passing a context to another thread does not associate the principal used in creating the context with the new thread. The only way to associate a principal with a new thread is to create a new context within that thread. In addition, a context can only be closed within the thread with which it was created. For these reasons, Oracle recommends that all work performed with a given context be handled in the same thread.

A thread may be associated with only one principal at any given time. If multiple contexts are created within a single thread without closing any contexts, the thread will be associated with the principal used in the last context created. Principal information is stored with the thread in a stack. If the last context is closed, the thread becomes associated with the principal used in creating the previous context, and so on.

To enable this feature in 10.1.3, set the system property `-DAssociateUserToThread=true` on the command line. By default, this feature is not enabled (set to `false`).

Browsing the JNDI Context

Using the JNDI Browser, you can view the entire JNDI namespace to verify that a given set of objects is actually bound in an application.

The JNDI Browser is available within the Oracle Enterprise Manager 10*g* Application Server Control Console as follows:

Path to JNDI Browser

OC4J: Home > Administration tab > Task Name: Services > JNDI Browser > Click Go To Task icon > Expand All

To Get JNDI Bindings as a String

You can also get a string representation of the bindings in a JNDI context. This can be useful in debugging.

OC4J: Home > Administration tab > Task Name: System MBean Browser. Go To Task > Drill down: J2EEDomain:oc4j, J2EEServer:standalone, JNDIResource > Select application > Operations tab > `getBindingsAsString` or `getBindingsAsXMLString` > Invoke

Looking Up Objects from J2EE Application Components

This section describes how to use the JNDI to look up bound objects from J2EE application components, such as servlets, JSP pages, and EJBs.

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

- Looking Up Objects In the Same Application
- Looking Up Objects in Another Application

Looking Up Objects In the Same Application

When code is running in a server, it is by definition part of an application. Because the code is part of an application, OC4J establishes defaults for properties that JNDI uses. Application code, such as a servlet or an EJB business method, does not need to provide any property values when constructing a JNDI `InitialContext` object.

Note: If your application must look up a remote reference, such as a resource in another J2EE application in the same JVM or a resource external to any J2EE application, then you must use `RMIInitialContextFactory` or `IIOPInitialContextFactory`. See "Looking Up Objects in Another Application" on page 2-9.

Example: Servlet Looking Up a Data Source

In this example, a servlet retrieves a data source to perform a JDBC operation on a database.

Use the Application Server Control Console to specify data source location. Specify the location in the JNDI Location field of the Create Data Source page. 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 corresponds to the `"jndi-name"` attribute 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);
}
...
```

When looking up objects within the same application, no initial context factory specification is necessary, because OC4J sets appropriate defaults when the application starts. For most lookups within the same application, only the `no-args` constructor for `javax.naming.InitialContext` to create an InitialContext is required.

Also, it is not necessary 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.

Looking Up Objects in Another Application

Use one of the following context factories to access objects in another application, or to access J2EE resources from a standalone java client:

- `oracle.j2ee.rmi.RMIInitialContextFactory`
- `oracle.j2ee.iiop.IIOPInitialContextFactory`
- `oracle.j2ee.naming.ApplicationClientInitialContextFactory`

RMIInitialContextFactory

The RMIInitialContexFactory provides a JNDI context implementation that uses the Oracle Remote Method Invocation (RMI) protocol for distributed lookups. This context factory is used by remote clients, as well as applications that attempt lookups for bindings deployed in other OC4J instances. The JNDI environment properties used by the `RMIInitialContextFactory` are described in Table 2–2,

" JNDI-Related Environment Properties" on page 2-12. For information on RMI, see the "Using Remote Method Invocation in OC4J" chapter on page 6-1.

Example: Servlet Looking Up an EJB Remotely Using RMI

In this example, a servlet accesses an EJB running on another OC4J instance on a different machine. The EJB in this example is the `EmployeeBean` that is used in the "Example: Application Client Looking Up an EJB" on page 2-13.

Here is an excerpt of the servlet code:

```
Hashtable env = new Hashtable();
env.put("java.naming.factory.initial",
"oracle.j2ee.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("EmployeeBean");
```

Example: Servlet Looking Up an EJB Remotely in a Multiple Instance Environment

When running OC4J within IAS, a remote JNDI service can specify that the request should use the `"opmn:ormi"` protocol, which allows a client to attempt a lookup without hard-coding the ORMI port information. The OC4J JNDI code contacts the opmn process to determine the proper ORMI port for this IAS install. The lookup is similar to the above ORMI example, except that the `"java.naming.provider.url"` property is set to a URL that begins with `"opmn:ormi"`. Here is an excerpt of the same lookup code, using `"opmn:ormi"`:

Here is an excerpt of the servlet code:

```
Hashtable env = new Hashtable();
env.put("java.naming.factory.initial",
"oracle.j2ee.rmi.RMIInitialContextFactory");
env.put("java.naming.provider.url","opmn: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("EmployeeBean");
```

There are also ways to specify which home instance the "opmn" URL should refer to. For more information on using "opmn", see Using Remote Method Invocation in OC4J on page 6-1.

IIOPInitial ContextFactory

The `IIOPInitialContexFactory` provides a JNDI Context implementation that uses the Internet Inter-ORB Protocol (IIOP) for distributed lookups. For information on RMI, see the "Using Remote Method Invocation in OC4J" chapter on page 6-1.

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

Note: You can use `IIOPInitialContextFactory` only for looking up EJBs. The same holds true for using a `"corbaname"` URL with the `ApplicationClientInitialContextFactory`.

Example: Servlet Looking Up an EJB Remotely Using IIOP

Here is an example.

```
Hashtable env = new Hashtable();
env.put("java.naming.factory.initial",
"oracle.j2ee.iiop.IIOPInitialContextFactory");
env.put("java.naming.provider.url","corbaname::remotehost:PORT_NUMBER#APPLICATION_
NAME");
env.put("java.naming.security.principal","SCOTT");
env.put("java.naming.security.credentials","TIGER");
Context context = new InitialContext(env);
Object homeObject =
context.lookup("EmployeeBean");
```

In this example, a `corbaname` URL is used to specify the location of the EJB. In the `corbaname` string provided in the above example, `PORT_NUMBER` is the IIOP port number that OC4J is configured to use, `remotehost` is the name of the server that OC4J is running on, and `APPLICATION_NAME` is the name of the application that contains the EJB.

The application used in this example must be deployed with IIOP enabled for this lookup to succeed. The remote client must also include the IIOP client jar generated by OC4J. For more information on this, see the OC4J IIOP documentation at "Switching from ORMI to IIOP Transport" on page 6-18.

Looking Up Objects from J2EE Application Clients

This section discusses how to configure an application client to access objects running inside an OC4J instance.

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

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

When an application client must look up a resource that is available in a J2EE server application, the client uses the `ApplicationClientInitialContextFactory` in the `oracle.j2ee.naming` package to construct the initial context.

Note: If your application is a J2EE application client (that is, it has an `application-client.xml` file), then you must always use `ApplicationClientInitialContextFactory` regardless of the protocol (ORMI or IIOP) that the client application is using. The protocol itself is specified by the JNDI property `java.naming.provider.url`. See Table 2–2, " JNDI-Related Environment Properties" on page 2-12 for details.

Remote and local access of components is essentially the same from the point of view of the remote client. Clients can use ORMI or IIOP, depending on the provider URL.

Consider an application client that consists of Java code that connects to an OC4J server. 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 remote client must specify the value of the property `java.naming.factory.initial` as `"oracle.j2ee.naming.ApplicationClientInitialContextFactory"`. This can be specified in client code, or it can be specified in the `jndi.properties` file that is part of the application's `application-client.jar` file included in the EAR file.

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

Environment Properties

If the ORMI protocol is being used, then `ApplicationClientInitialContextFactory` reads the properties listed in Table 2–2 from the environment.

Table 2–2 JNDI-Related Environment Properties

Property	Meaning
`dedicated.rmicontext`	This property is no longer used by OC4J. This property previously was used for two reasons: ■ To enable load balancing. ■ As a workaround for known ORMI/JNDI bugs. In Version 10.1.3, the known ORMI /JNDI bugs that required this flag have been resolved. To enable client-side ORMI load-balancing in 10.1.3, please use the properties described in the "Load Balancing" section. The properties described in the "Load Balancing" section on page 2-13 can be used in the few cases that would still require this flag.

`java.naming.provider.url`	This property specifies the URL to use when looking for local or remote objects. The format is either
	`[opmn:|http:|https:]ormi://hostname/appname`
	or
	`corbaname:hostname:port`
	For details on the *`corbaname`* URL, see "Specifying the corbaname URL" on page 6-21.
	You can supply multiple hosts (for failover) in a comma-separated list when using the ORMI protocol.

`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.
`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 (Cont.) JNDI-Related Environment Properties

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`.

Load Balancing

In situations where heavy request volume is expected, load balancing of requests across OC4J instances may be desired. Load balancing is configurable using the `oracle.j2ee.rmi.loadBalance` property, which can be set in the client's `jndi.properties` file or in a Hashtable in the client code. The values for this the `oracle.j2ee.rmi.loadBalance` property are outlined in the following table.

Table 2–3 oracle.j2ee.rmi.loadBalance Property Values

Value	Description
`client`	If specified, the client interacts with the OC4J process that was initially chosen at the first lookup for the entire conversation.
`context`	Used for a Web client (servlet or JSP) that will access EJBs in a clustered OC4J environment. If specified, a new `Context` object for a randomly-selected OC4J instance will be returned each time `InitialContext()` is invoked.
`lookup`	Used for a standalone client that will access EJBs in a clustered OC4J environment. If specified, a new `Context` object for a randomly-selected OC4J instance will be created each time the client calls `Context.lookup()`.

For more information on load balancing see "Configuring ORMI Request Load Balancing" on page 6-11.

Example: Application Client Looking Up an EJB

This section shows how to configure an application client to access an EJB running inside an OC4J instance.

First, the EJB is deployed into OC4J. with the name `EmployeeBean`. The name is defined as follows 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 `bmpapp/EmployeeBean` in `orion-ejb-jar.xml` as follows:

```
<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 environment naming context (java:comp/env). The ejb reference 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.
 Employee rec = home.create(empNo, empName, salary);
 // 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=oracle.j2ee.naming.ApplicationClientInitialContextFactory
java.naming.provider.url=ormi://localhost/bmpapp
java.naming.security.principal=SCOTT
java.naming.security.credentials=TIGER
```

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

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
"oracle.j2ee.naming.ApplicationClientInitialContextFactory");
env.put("java.naming.provider.url","ormi://localhost/bmpapp");
env.put("java.naming.security.principal","SCOTT");
env.put("java.naming.security.credentials","TIGER");
Context initial = new InitialContext(env);
```

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

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

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

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

For details on the `application-client.xml` file and the `orion-application-client.xml` file, see *Oracle Containers for J2EE Developer's Guide*, Appendix A, OC4J-Specific Deployment Descriptors.

Example: Application Client Looking Up an EJB Using IIOP

In the previous example, the application client used ORMI as the underlying protocol for remote lookups of resources. Alternatively, you can configure the application client to use IIOP as the remote protocol. The code for looking up the `EmployeeBean` is the same as the previous example, with the following differences:

The `jndi.properties` file must be changed to reflect the fact that IIOP is being used as the remote protocol. This is configured in the `java.naming.provider.url` system property in `jndi.properties`. The following is the `jndi.properties` file for the same example, configured to use IIOP:

```
java.naming.factory.initial=oracle.j2ee.naming.ApplicationClientInitialContextFactory
java.naming.provider.url=corbaname::REMOTE_HOST:IIOP_PORT#bmpapp
java.naming.security.principal=SCOTT
java.naming.security.credentials=TIGER
```

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

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
"oracle.j2ee.naming.ApplicationClientInitialContextFactory");
env.put("java.naming.provider.url","corbaname::REMOTE_HOST:IIOP_PORT#bmpapp");
env.put("java.naming.security.principal","SCOTT");
env.put("java.naming.security.credentials","TIGER");
Context initial = new InitialContext(env);
```

If an application client sets the JNDI properties through a hash table, as shown in this example, the security credentials must be set as system properties. The IIOP interceptors require access to the client's credentials through system properties. This is achieved automatically when using the `jndi.properties` file, since the JNDI framework automatically sets these properties as system properties.

If the properties are set in the code itself, the following lines must be added to the client code:

```
System.setProperty("java.naming.security.principal","SCOTT");
System.setProperty("java.naming.security.credentials ","TIGER");
```

In both the `jndi.properties` file and the programmatic example,

- `REMOTE_HOST` must be set to the name of the server running the OC4J server.
- `IIOP_PORT` must be set to the port number used by OC4Jfor serving IIOP requests.

When using IIOP with the application client, no changes need to be made to the application client code or the deployment descriptors. The client must be configured to use IIOP properly, including the use of the `jndi.properties` file.

The remote application client must also include the deployed application's IIOP client jar in its classpath, in order to have access to the IIOP stubs for each EJB deployed.

See "Switching from ORMI to IIOP Transport" on page 6-18 for more information on this.

JNDI State Replication

This section discusses JNDI state replication.

Note: JNDI State Replication supports clustering in a multiple-OC4J environment.

If you are using Oracle Application Server in standalone mode as an environment for developing applications to be used in a multiple-OC4j environment, the information in this section is provided to support your planning and coding process.

The section covers the following topics:

- What Is JNDI State Replication
- Enabling JNDI State Replication
- Limitations of JNDI State Replication

What Is JNDI State Replication

JNDI state replication causes changes made to the context on one OC4J instance of an OC4J cluster to be replicated to the name space of every other OC4J instance in the cluster.

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

Enabling JNDI State Replication

JNDI state replication is enabled on a per-application basis. When an application is deployed, a `<cluster>` tag can be added to `orion-application.xml`, which is found in the application's `META-INF` directory. This type of configuration allows for control over the clustering status of each application. Adding this `<cluster>` tag enables HTTP/EJB session replication as well as JNDI State replication.

The `<cluster>` element serves as the single mechanism for clustering configuration. You can add the `<cluster>` element to either of the following files:

- The `ORACLE_HOME/j2ee/home/config/application.xml` file to configure clustering at the global level.
- An application-specific `orion-application.xml` file for per-application clustering configuration.

Configuring the `<cluster>` element mainly involves specifying the following subelements:

- The `<replication-policy>` subelement controls when replication occurs and what data is replicated.
- The `<protocol>` subelement defines what mechanism to use for replication - `multicast` (default), `peer-to-peer`, or `database`.

Here are two example OC4J clustering and JNDI state replication configuration.

Example 1:

This example configures an application to communicate over a multicast network. It is important to make sure that all nodes in this cluster use the same multicast port.

```
<cluster>
    <protocol>
        <multicast ip="230.230.0.30" port="45678" />
    </protocol>
</cluster>
```

Example 2:

This example uses a peer-to-peer protocol for JNDI state replication. This example requires that all nodes running in the cluster be part of a managed IAS instance. This Oracle Application Server instance is controlled by OPMN.

```
<cluster>
```

```
    <replication-policy trigger="onRequestEnd" scope="modifiedAttributes" />
    <protocol>
        <peer>
            <opmn-discovery/>
        </peer>
    </protocol>
</cluster>
```

For detailed information on OC4J clustering, see the *Oracle Containers for J2EE Configuration and Administration Guide*.

Limitations of JNDI State Replication

Consider the following limitations when relying on JNDI state replication:

- Propagating Changes Across the Cluster
- Binding a Remote Object

Propagating Changes Across the Cluster

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

Binding a Remote Object

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

3

Oracle Enterprise Messaging Service (OEMS)

The Oracle Enterprise Messaging Service (OEMS) provides a robust messaging platform for building and integrating distributed applications. It provides the framework for Oracle's messaging and message integration solutions.

The following key features make up OEMS:

- **Standardized interface**
  - Java Message Service (JMS) and J2EE Connector Architecture (J2CA)
- **Quality of Service choice for message persistence**
  - In-Memory, File-Based, or Oracle Database
- **Seamless Integration with non-Oracle messaging systems**
  - JMS Connector, JMS Router, and Messaging Gateway (MGW)

All of these areas are covered in this chapter. The only OEMS feature not covered here is MGW, which is documented in the *Oracle Streams Advanced Queuing User's Guide and Reference* document.

Note:

In past releases, Oracle has used the terms "OracleAS JMS" and "OJMS" when describing the In-Memory, File-Based, Database persistence options. "OracleAS JMS" referred to the In-Memory and File-Based options while "OJMS" referred to JMS interface to Streams Advanced Queuing (AQ).

To avoid any confusion regarding JMS, the "OracleAS JMS" and "OJMS" nomenclature will not be used. The "OEMS JMS" reference will be used instead. This reflects the fact that Oracle offers a single JMS interface to the three message persistence options. Your JMS application code will not have to change if you decide to change message persistence between any of the three quality of service choices.

This chapter discusses the following topics:

- JMS Tasks
- New JMS Features
- About JMS

- JMS Configuration Overview
- OEMS JMS In-Memory and File-Based Persistence
- OEMS JMS Database Persistence
- JMS Connector
- Resource Providers
- Sending and Receiving JMS Messages
- Using High Availability and Clustering for OEMS JMS
- JMS Router

JMS Tasks

This chapter discusses the following JMS tasks:

- Configuring Destination Objects and Connection Factories.
- Configuring Ports
- Sending and Receiving JMS Messages
- Declaring the OEMS JMS Database Reference
- Enabling File-Based Persistence in the Application Server Control Console
- Using Logical Names to Reference Resources
- Using Third-Party JMS Providers
- Using Message-Driven Beans
- Using High Availability and Clustering for OEMS JMS

New JMS Features

The following OC4J JMS features and behaviors are new for this release:

- **JMS 1.1 and J2EE 1.4 Compliance** - OEMS JMS is JMS 1.1 compliant and J2EE 1.4 compliant.

  You can access the JMS 1.1 specification at: `http://java.sun.com/products/jms/docs.html`.

  You can access the J2EE 1.4 specification at: `http://java.sun.com/j2ee/1.4/docs/index.html`

- **Domain Unification** - You can use the same JMS connection and session to operate on OEMS JMS queues and topics. This is a JMS 1.1 feature.
- **JMS Connector** - The JMS Connector is a generic JMS J2CA resource adapter used to integrate the OEMS JMS In-Memory, File-Based, and Database resource providers, as well as non-Oracle JMS providers, in to OC4J. Supported non-Oracle JMS providers that can be integrated are WebSphereMQ, Tibco, and SonicMQ. All of these providers are now fully integrated with OC4J. The JMS Connector supports MDBs. Other significant advantages are:
  – Support for global transactions
  – MDBs that react to changing loads
  – JMS connection pooling (not implemented for applications running in a non-managed environment)

  - Performance enhancement features: Lazy enlistment of global transactions and lazy evaluation of JMS operations. Certain operations, such as enlisting resources and looking up connection factories and destinations are not performed until necessary.

  For discussion of the Resource Adapter, see *Oracle Containers for J2EE Resource Adapter Administrator's Guide*.

- Support for **Message Driven Beans** (MDB) for the following JMS resource providers:
  - OEMS JMS In-Memory, File-Based, and Database
  - IBM WebSphere MQ-based JMS
  - TIBCO Enterprise for JMS
  - SonicMQ
- **Transactions across Domains** - You can engage queues and topics in the same transaction. This is a JMS 1.1 feature.
- **JMS Router** - The JMS Router provides message-bridging routing services. The JMS Router is discussed in the "JMS Router" section starting on page 3-76.
- **Oracle and Third-party Support** - The following JMS message providers are officially supported through the JMS Connector bundled with OC4J:
  - OEMS JMS In-Memory, File-Based, and Database
  - IBM WebSphere MQ for JMS versions 6.0 and 5.3 resource provider
  - TIBCO Enterprise for JMS version 3.1.0
  - SonicMQ JMS 6.0

  For information on the third-party providers, see "Resource Providers" on page 3-16. The demos that demonstrate configuration and usage of the JMS providers through the Resource Adapter are available at `http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`.
- **Start-order independence** - See the "JMS Connector" section on page 3-51 section for details.
- **Demo Code** - Demos for various JMS configuration functions are available at `http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`.

  For a list of the How-To documents and demo sets and their URLs, see "JMS How-To Documents and Demo Sets" on page 3-4.

About JMS

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.

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

`http://java.sun.com/products/jms/index.htm`

JMS provides two messaging domains, each associated with a JMS destination type, and a domain-specific set of Java interfaces:

- Point-to-Point—Messages are sent to a single consumer using a JMS queue.
- Publish/Subscribe—Messages are broadcast to all registered listeners using a JMS topic.

JMS destination objects are bound in the JNDI environment and made available to J2EE applications.

In addition to providing two sets of messaging interfaces, one for each messaging domain, JMS (starting with JMS 1.1) also provides a set of common interfaces for implementing domain-independent application code. This set of common interfaces maintains the distinct behavior of the two messaging domains (where the behavior is governed by the messaging domain used, as associated with the JMS destination type), while providing common programming interfaces for both messaging domains. The interfaces belonging to this set of common interfaces, as well as how they relate to the domain-specific interfaces, are detailed in Table 2-1 in the JMS 1.1 specification document.

Backward Compatibility

Oracle recommends that newer JMS applications be deployed using the JMS Connector, which is based on the J2CA 1.5 specification and mandated by the J2EE 1.4 standard. This path provides the new features introduced in OracleAS 10.1.3. However, Oracle will continue to support JMS applications deployed using the older proprietary OC4J Resource Provider supported in OracleAS 9.0.4/10.1.2.

The Oracle JMS Connector is discussed in the "JMS Connector" section starting on page 3-51.

JMS How-To Documents and Demo Sets

How-To documents and a set of examples, including commented configuration files, are available at the How-To website:
`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`

The JMS documents and demo sets are listed under the Messaging (JMS) heading. The documents and deployment descriptor files in the demo set are organized by resource provider and include the configuration variations called for by the various supported resource providers. Unzip the files that apply to the relevant resource provider.

Table 3–1, " JMS How-To Documents and Demo Sets" lists the JMS topics, the associated How-To documents, and their demo set ZIP files.

Table 3–1 JMS How-To Documents and Demo Sets

JMS Topic	How-To Document URL	Demo Set ZIP File
How To Use the JMS Router for OracleAS JMS and Oracle OJMS	`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/how-to-use-JMS-router/doc/How-to-Use-JMS-Router.html`	`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/how-to-use-JMS-router/how-to-use-JMS-router.zip`
How-To Configure and Use MQ Series® JMS with OC4J 10g (10.1.3) JCA 1.5 Resource Adapters	`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/how-to-mq-jms/doc/how-to-mq-jms.html`	`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/how-to-mq-jms/how-to-mq-jms.zip`
How to Configure and Use Oracle's Generic JMS Resource Adapter with OracleAS JMS	`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/how-to-gjra-with-oracleasjms/doc/how-to-gjra-with-oracleasjms.html`	`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/how-to-gjra-with-oracleasjms/how-to-gjra-with-oracleasjms.zip`
How to Configure and Use Oracle's Generic JMS Resource Adapter with OJMS	`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/how-to-gjra-with-ojms/doc/how-to-gjra-with-ojms.html`	`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/how-to-gjra-with-ojms/how-to-gjra-with-ojms.zip`
How to Configure and Use Oracle's Generic JMS Resource Adapter with IBM WebSphere MQ JMS	`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/how-to-gjra-with-mqseries/doc/how-to-gjra-with-mqseries.html`	`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/how-to-gjra-with-mqseries/how-to-gjra-with-mqseries.zip`
How to Configure and Use Oracle's Generic JMS Resource Adapter with Tibco Enterprise for JMS	`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/how-to-gjra-with-tibco/doc/how-to-gjra-with-tibco.html`	`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/how-to-gjra-with-tibco/how-to-gjra-with-tibco.zip`
How to Configure and Use Oracle's Generic JMS Resource Adapter with SonicMQ JMS	`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/how-to-gjra-with-sonic/doc/how-to-gjra-with-sonic.html`	`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/how-to-gjra-with-sonic/how-to-gjra-with-sonic.zip`

Additional Documentation

The How-To documents at the following site provide additional information about OC4J 10g Release 3 (10.1.3) features, including feature overviews and code excerpts relevant to the feature.

`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`

JMS Configuration Overview

This section gives an overview of the following JMS configuration topics:

- JMS Configuration Sequence
- JMS Configuration File Structure

This OEMS document and the associated demos and How-To documents describe using the JMS Connector with the various supported resource providers, with emphasis on the OEMS JMS In-Memory and File-Based persistence options.

Additional discussions of these topics are available in other parts of this document. Also, a demo set of examples, including How-To documents and commented configuration files, is available at: `http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`. For a list of the How-To documents and demo sets and their URLs, see "JMS How-To Documents and Demo Sets" on page 3-4.

JMS Configuration Sequence

This section outlines preparing the following components for JMS operation:

- Developing and Assembling the Application
- Configuring the Resource Provider
- Configuring the JMS Connector

You can, but are not required to, create matching sets of connection factories and destination objects, one set on the resource provider and the matching set on the JMS Connector. Alternatively, you can use "automatic destination wrapping" to avoid having to make a matching set of JMS Connector destinations.

Developing and Assembling the Application

The tasks for developing and assembling your application to use JMS messaging are as follows:

- Write Code to Send and Receive Messages
- Declare Logical Names for JMS Resources
- Use Logical Names for JMS Resources
- Create and Declare an MDB Class
- Declare Message Destinations
- Link to Message Destinations
- Define the onMessage Transaction Attribute
- List the Application Modules

For details, see the How-To documents included in the demo set. For a list of the How-To documents and demo sets and their URLs, see "JMS How-To Documents and Demo Sets" on page 3-4.

Configuring the Resource Provider

Configuring the resource provider usually requires several rounds, arising out of the need for application component developers and application assemblers to have some connection factories and destinations to use for development. The development connection factories and destinations are often not the same connection factories and destinations used for deployment (since the development servers and production servers are often separate machines, and may be configured using different organization strategies, and may even use different resource providers). This document focuses on the production deployment.

When configuring a resource provider, you have to decide the following:

- How many and what type of resource provider connection factories will be needed to satisfy the application.
- How many and what type of resource provider destinations are needed to satisfy the application.

- Create RP Connection Factories
- Create RP Destinations
- Declare a Resource Provider Reference

See the How-to documents in the demo set for details. For a list of the How-To documents and demo sets and their URLs, see "JMS How-To Documents and Demo Sets" on page 3-4.

Configuring the JMS Connector

The introduction of the JMS Connector functionality in OEMS provides some degree of insulation from the specifics of the various resource providers. Based on the J2CA 1.5 specification, the JMS Connector acts as a compatibility layer and a value-added wrapper for the resource provider.

The tasks for configuring the JMS Connector are as follows:

- Settings in `ra.xml`.
- Create a JMS Connector Instance
- Create JMS Connector Connection Factories
- Create JMS Connector Destinations

For details, see the How-To documents included in the demo set. For a list of the How-To documents and demo sets and their URLs, see "JMS How-To Documents and Demo Sets" on page 3-4.

Additional Information and Examples

For detailed examples of the `oc4j-connectors.xml`, the `oc4j-ra.xml`, and the `ra.xml` files, go to:
`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`

For detailed reference information on the JMS Connector XML files, go to Appendix A, OC4J Resource Adapter Configuration Files of the *Oracle Containers for J2EE Resource Adapter Administrator's Guide*.

The following links point to the document "How to Configure and Use Oracle's Generic JMS Resource Adapter with OracleAS JMS" and to the ZIP file containing the corresponding set of demo files. In these documents, the term "Generic JMS Resource Adapter" refers to the JMS Connector.

- "How to Configure and Use Oracle's Generic JMS Resource Adapter with OracleAS JMS" is available at:

  `http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/how-to-gjra-with-oracleasjms/doc/how-to-gjra-with-oracleasjms.html`

- The ZIP file containing the corresponding set of demo files is available at:

  `http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/how-to-gjra-with-oracleasjms/how-to-gjra-with-oracleasjms.zip`

JMS Configuration File Structure

This section points out the consistencies that must exist between the connection-factory and destination references and definitions in the JMS configuration files. Figure 3–1, "JMS Infrastructure" illustrates the references and definitions that must agree with each other.

Figure 3–1 JMS Infrastructure

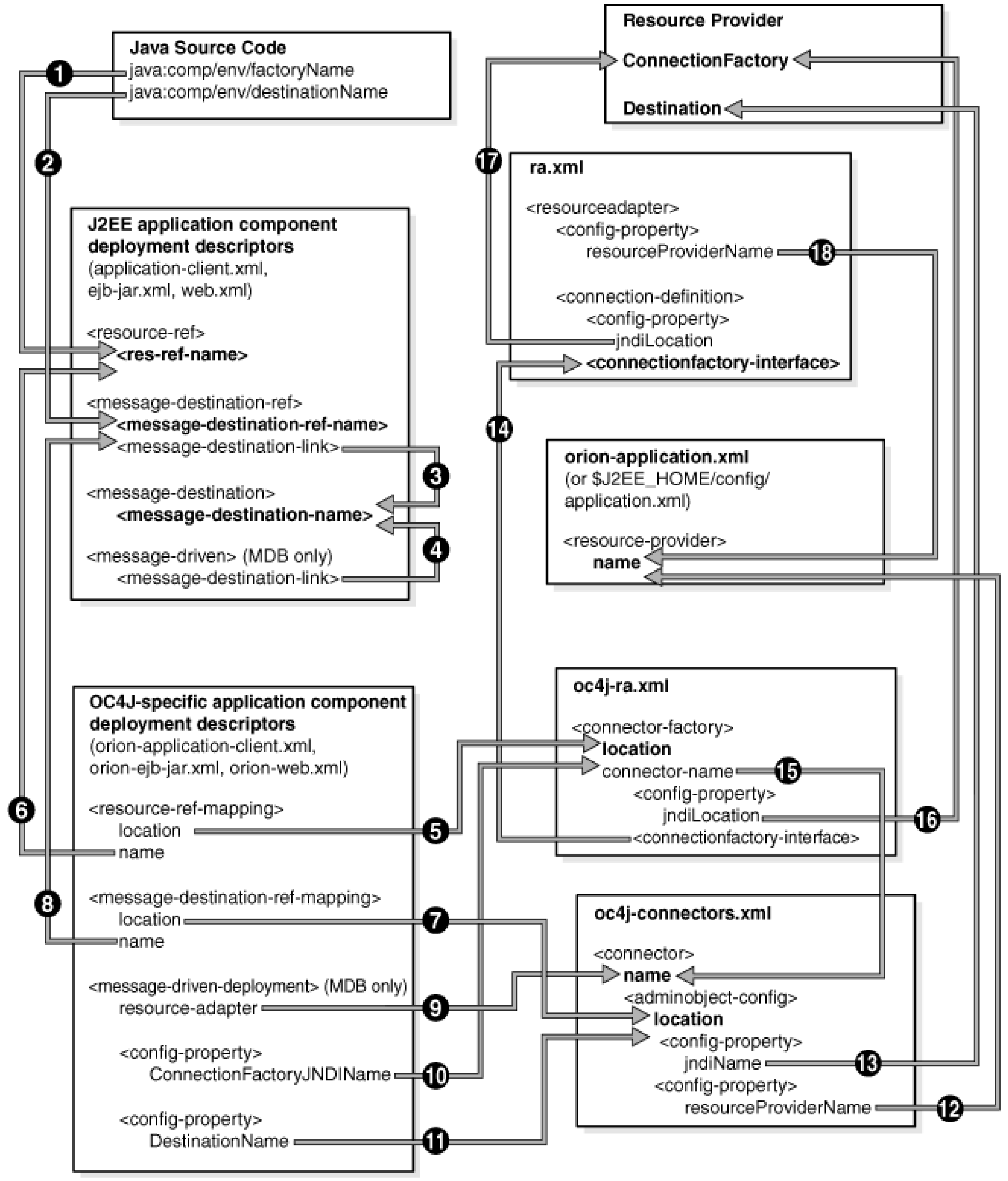

Java Source Code
java:comp/env/factoryName
java:comp/env/destinationName
Resource Provider
ConnectionFactory
Destination
ra.xml
<resourceadapter>
<config-property>
resourceProviderName
<connection-definition>
<config-property>
jndiLocation
<connectionfactory-interface>
J2EE application component deployment descriptors
(application-client.xml, ejb-jar.xml, web.xml)
<resource-ref>
<res-ref-name>
<message-destination-ref>
<message-destination-ref-name>
<message-destination-link>
<message-destination>
<message-destination-name>
<message-driven> (MDB only)
<message-destination-link>
orion-application.xml
(or $J2EE_HOME/config/application.xml)
<resource-provider>
name
OC4J-specific application component deployment descriptors
(orion-application-client.xml, orion-ejb-jar.xml, orion-web.xml)
<resource-ref-mapping>
location
name
<message-destination-ref-mapping>
location
name
<message-driven-deployment> (MDB only)
resource-adapter
<config-property>
ConnectionFactoryJNDIName
<config-property>
DestinationName
oc4j-ra.xml
<connector-factory>
location
connector-name
<config-property>
jndiLocation
<connectionfactory-interface>
oc4j-connectors.xml
<connector>
name
<adminobject-config>
location
<config-property>
jndiName
<config-property>
resourceProviderName
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

Figure 3–1 depicts the various types of links between the java source code, the application deployment descriptors, the resource adapter deployment descriptors, and the resource provider. At the tail of each arrow is a "link-reference", and at the head of each arrow is the "link-key" (name, JNDI location or Java interface) of the item being referenced. The textual representation of the link-key at the head and link-reference at the tail of any given arrow is always identical except where otherwise noted.

The files are:

- `ra.xml`
- `oc4j-ra.xml`
- `application.xml`
- `orion-application.xml`
- `oc4j-connectors.xml`
- `ejb-jar.xml`
- `orion-ejb-jar.xml`
- `application-client.xml`
- `orion-application-client.xml`
- `web.xml`
- `orion-web.xml`

The demo set includes expanded explanations of the relationships depicted in Figure 3–1, "JMS Infrastructure". For a list of the How-To documents and demo sets and their URLs, see "JMS How-To Documents and Demo Sets" on page 3-4.

Download and unzip the relevant How-To-gjra-with-*xxx*.zip file, where *xxx* is the name of the relevant resource provider.

Open the relevant how-to-gjra-with-*xxx*.html document. The explanations in this section point to the section in the how-to document that explains the relationship.

The demo set also includes examples of Java code and deployment descriptor XML files.

Java Source Code

In J2EE applications, Java source code typically uses logical names to reference JMS resources. Logical names (references declared with `<resource-ref>` and `<message-destination-ref>` elements) are types of environment entries, and all environment entries are placed in the `java:comp/env/` JNDI subcontext. For details, see "Application Component Provider Task #3: Use Logical Names for JMS Resources" in the relevant How-To document.

Figure 3–1 depicts the two types of links from the Java source code to the J2EE application component deployment descriptors. The number of each description in the following list corresponds to a numbered link in the figure.

- **1** - The first link in a typical outbound connection factory "chain" is from the Java source code to a `<resource-ref>` element in a J2EE application component deployment descriptor. The link-reference is the location used in the JNDI lookup to obtain the JMS connection factory and must include the `java:comp/env` prefix. The link-key for a `<resource-ref>` element is the value of its `<res-ref-name>` subelement and must not include the java:comp/env prefix. Except for the `java:comp/env` prefix, the link-reference and link-key should be identical.

- **2** - The first link in a typical outbound destination chain is from the Java source code to a `<message-destination-ref>` element in a J2EE application component deployment descriptor. The link-reference is the location used in the JNDI lookup to obtain the JMS destination and must include the `java:comp/env` prefix. The link-key for a `<message-destination-ref>` element is the value of its `<message-destination-ref-name>` subelement and must not include the java:comp/env prefix. Except for the java:comp/env prefix, the link-reference and link-key should be identical.

J2EE application component deployment descriptors (application-client.xml, ejb-jar.xml, web.xml)

The logical names used by the Application Component Providers have a many-to-one relationship with physical destinations. The Application Assembler creates logical destinations which have a one-to-one relationship with physical destinations. The Application Assembler then needs to link the message destination references and MDBs created by the Application Component Providers to the message destinations created by the Application Assembler. This is done by adding `<message-destination-link>` elements that name the appropriate message destination. These links are not part of the destination chain, but instead provide information needed by the Deployer to complete the destination chain. For details, see "Application Assembler Task #2: Link to Message Destinations" in the relevant how-to-gjra-with-xxx.html document, where xxx is the name of the relevant resource provider.

Figure 3–1 depicts the two types of links fully contained within the J2EE application component deployment descriptor:

- **3** - For outbound messaging, this informational-only link is from a `<message-destination-ref>` element to a `<message-destination>` element, both in J2EE application component deployment descriptors (though not necessarily the same one). The link-reference is the value of the `<message-destination-link>` subelement. The link-key for a `<message-destination>` element is the value of its `<message-destination-name>` subelement. The link-reference may be prefixed with the name of the file containing the link-key followed by a # character. (This is only needed when different link-keys from different files happen to have the same exact value.) Except for the optional prefix, the link-reference and link-key should be identical.
- **4** - For MDB/inbound messaging, this informational-only link is from a `<message-driven>` element to a `<message-destination>` element, both in J2EE application component deployment descriptors (though not necessarily the same one). The link-reference is the value of the `<message-destination-link>` subelement. As previously mentioned, the link-key for a `<message-destination>` element is the value of its `<message-destination-name>` subelement. The link-reference may be prefixed with the name of the file containing the link-key followed by a # character. (This is only needed when different link-keys from different files happen to have the same exact value.) Except for the optional prefix, the link-reference and link-key should be identical.

OC4J-specific application component deployment descriptors (orion-application-client.xml, orion-ejb-jar.xml, orion-web.xml)

Figure 3–1 depicts the following seven types of references from the OC4J-specific application component deployment descriptor:

- Application components declare logical names for connection factories in J2EE application component deployment descriptors. The deployer then maps those logical names to JMS Connector connection factories.

  For details and examples, see "Deployer Task #1: Map Logical Connection Factories to RA ConnectionFactories" in the relevant How-To document.

  The figure depicts the two types of connection factory links provided by the `<resource-ref-mapping>` element.

  – **6** - The second link in a typical outbound connection factory chain is from a `<resource-ref-mapping>` element in an OC4J-specific application component deployment descriptor back to a `<resource-ref>` element in a J2EE application component deployment descriptor. The link-reference is the value of the `name` attribute of the `<resource-ref-mapping>` element. As mentioned previously, the link-key for a `<resource-ref>` element is the value of its `<res-ref-name>` subelement.

  – **5** - The third link in a typical outbound connection factory chain is from a `<resource-ref-mapping>` element in an OC4J-specific application component deployment descriptor to a `<connector-factory>` element in an `oc4j-ra.xml` file. The link-reference is the value of the location attribute of the `<resource-ref-mapping>` element. The link-key for a `<connector-factory>` element is the value of its location attribute (which is also the JNDI location where the resource adapter connection factory defined by the given `<connector-factory>` element will be bound).

- Application components declare logical names for destinations in J2EE application component deployment descriptors. The deployer then maps those logical names to JMS Connector destinations, making use of any information provided by the Application Assembler in the form of `<message-destination-link>`s and `<message-destination>`s.

  For each `<message-destination>`, the deployer must map all `<message-destination-ref>`s and MDBs linked to that `<message-destination>` to the same destination.

  For details and examples, see "Deployer Task #2: Map Logical Destinations to RA Destinations" in the relevant How-To document.

  The figure depicts the two types of destination links provided by the `<destination-ref-mapping>` element:

  – **8** - The second link in a typical outbound destination chain is from a `<message-destination-ref-mapping>` element in an OC4J-specific application component deployment descriptor back to a `<message-destination-ref>` element in a J2EE application component deployment descriptor. The link-reference is the value of the `name` attribute of the `<message-destination-ref-mapping>` element. As mentioned previously, the link-key for a `<message-destination-ref>` element is the value of its `<message-destination-ref-name>` subelement.

  – **7** - The third link in a typical outbound destination chain is from a `<message-destination-ref-mapping>` element in an OC4J-specific application component deployment descriptor to an `<adminobject-config>` element in an `oc4j-connectors.xml` file. The link-reference is the value of the location attribute of the `<message-destination-ref-mapping>` element. The link-key for an `<adminobject-config>` element is the value of its location attribute (which is also the JNDI location where the resource adapter destination defined by the given `<adminobject-config>` element will be bound).

- For each MDB, the deployer must indicate the JMS Connector instance, connection factory and destination that should be used to meet the MDB's inbound messaging requirements.

  For details and examples, see "Deployer Task #2: Map Logical Destinations to RA Destinations" and "Deployer Task #3: Configure the MDB" in the relevant document.

  Figure 3–1 depicts the three types of inbound messaging links from the `orion-ejb-jar.xml` file to the `oc4j-connectors.xml` and `oc4j-ra.xml` files:

  – **9** - The link that tells the container which JMS Connector instance should be used to handle a given MDB's inbound messaging needs is from the MDB's `<message-driven-deployment>` element in an `orion-ejb-jar.xml` file to the JMS Connector instance's `<connector>` element in an `oc4j-connectors.xml` file. The link-reference is the value of the resource-adapter attribute of the `<message-driven-deployment>` element. The link-key for a `<connector>` element is the value of its `name` attribute (which is also the JNDI location where the resource adapter instance defined by the given `<connector>` element will be bound).

  – **10** - For MDB/inbound messaging, a single link is used instead of the three connection factory links described to this point. This link is from a `<message-driven-deployment>` element in an `orion-ejb-jar.xml` file to a `<connector-factory>` element in an `oc4j-ra.xml file`. The link-reference is the value of the `<message-driven-deployment>`'s `ConnectionFactoryJNDIName` config property. As mentioned previously, the link-key for a `<connector-factory>` element is the value of its location attribute (which is also the JNDI location where the resource adapter connection factory defined by the given `<connector-factory>` element will be bound). Note that this links to the same place as the third link in the outbound case, and the rest of the connection factory chain is the same for both inbound and outbound messaging.

  – **11** - For MDB/inbound messaging, a single link is used instead of the three destination links described to this point. This link is from a `<message-driven-deployment>` element in an `orion-ejb-jar.xml` file to an `<adminobject-config>` element in an `oc4j-connectors.xml` file. The link-reference is the value of the `<message-driven-deployment>`'s `DestinationName` config property. As mentioned previously, the link-key for an `<adminobject-config>` element is the value of its location attribute (which is also the JNDI location where the resource adapter destination defined by the given `<adminobject-config>` element will be bound). Note that this links to the same place as the third link in the outbound case, and the rest of the destination chain is the same for both inbound and outbound messaging.

oc4j-connectors.xml

JMS Connector destinations act as wrappers for RP destinations. In order for the JMS Connector to look up an RP destination, the JMS Connector needs to know the JNDI location of the RP destination.

For details and examples, see "Resource Adapter Task #4: Create RA Destinations" in the relevant How-To document. For information on creating resource provider destinations, see "Configuring the Resource Provider" in the relevant How-To document.

Figure 3–1 depicts the two types of links which together tie JMS Connector destinations defined in `oc4j-connectors.xml` to RP destinations:

- **12** - The final segment of the destination chain is from an `<adminobject-config>` element in an `oc4j-connectors.xml` file (which defines a JMS Connector destination) to a resource provider destination, and is composed of two parallel links. The first link is from the `<adminobject-config>` element to a `<resource-provider>` element in an individual application's `orion-application.xml` file (or the default application's `application.xml` file). The link-reference is the value of the `<adminobject-config>`'s `resourceProviderName` config property. The link-key for a `<resource-provider>` element is the value of its `name` attribute (which is also the JNDI location under `java:comp/resource` where the resource provider reference defined by the `<resource-provider>` element will be bound - the *providerName*).
- **13** - The second link completes the connection from the `<adminobject-config>` element to the resource provider destination. The link-reference is the value of the `<adminobject-config>`'s `jndiName` config property. The link-key is the JNDI location of the RP destination within the resource provider's JNDI context (the *resourceName*). Together, these two links provide the JMS Connector destination with the full JNDI location

  `java:comp/resource/`*`providerName`*`/`*`resourceName`*

  of the RP destination.

oc4j-ra.xml

JMS Connector connection factories act as wrappers for RP connection factories. In order for the JMS Connector to look up an RP connection factory, the JMS Connector needs to know the JNDI location of the RP connection factory.

For details and examples, see "Resource Adapter Task #3: Create RA Connection Factories" in the relevant How-To document. For information on creating resource provider connection factories (but not for the OEMS JMS Database persistence option since all these connection factories are automatically pre-created, see "Configuring the Resource Provider" in the relevant How-To document.

Figure 3–1 depicts three types of links that help define the implementation for JMS Connector connection factories defined in `oc4j-ra.xml` and tie them to RP connection factories:

- **15** - In theory this link ties a `<connector-factory>` element in an `oc4j-ra.xml` file (which defines a JMS Connector connection factory) to a `<connector>` element in an `oc4j-connectors.xml` file (which defines an JMS Connector instance). In practice this link may not actually be used, but for future compatibility it should be set as follows: The link-reference is the value of the `connector-name` attribute of the `<connector-factory>` element. As previously mentioned, the link-key for a `<connector>` element is the value of its `name` attribute (which is also the JNDI location where the JMS Connector instance defined by the given `<connector>` element will be bound).
- **16** - The final segment of the connection factory chain is from a `<connector-factory>` element in an `oc4j-ra.xml` file (which defines a JMS Connector connection factory) to an RP connection factory, and is composed of

two parallel links. The first link gives the JNDI location under `java:comp/resource` where the resource provider reference is bound (the *providerName*). It's link-reference is located in the `ra.xml` file - see the description for arrow #18. The second link completes the connection from the `<connector-factory>` element to the resource provider connection factory. The link-reference is the value of the `<connector-factory>`'s `jndiLocation` config property. The link-key is the JNDI location of the RP connection factory within the resource provider's JNDI context (the *resourceName*). Together, these two links provide the JMS Connector connection factory with the full JNDI location

```
java:comp/resource/providerName/resourceName
```

of the RP connection factory. NOTE: This link is actually an optional over-ride. (See description for arrow #17.) By convention this over-ride is always set (even if its value is the same as the value it is overriding).

- **14** - The implementation details for a JMS Connector connection factory (a `<connector-factory>` element in an `oc4j-ra.xml` file) are defined by linking the `<connector-factory>` element to a `<connection-definition>` element in an `ra.xml` file. The link-reference is the value of the `<connector-factory>`'s `<connectionfactory-interface>` subelement. The link-key for a `<connection-definition>` element is the value of its `<connectionfactory-interface>` subelement.

ra.xml

Much of the content in the `ra.xml` file does not need to be changed when using the JMS Connector. (The reason for this is that the `ra.xml` file is based on a J2EE Connector Architecture 1.5 schema file, which is a generic schema intended to work with many types of resource adapters, including non-JMS resource adapters.)

For details and examples, see "Resource Adapter Task #1: Customize the ra.xml File" in the relevant How-To document.

Figure 3–1 depicts two types of links that define default JMS Connector connection factory settings in `ra.xml`:

- **18** - This is the first link of the final segment of the connection factory chain. (See description for arrow #16.) This link is from a `<resourceadapter>` element in an `ra.xml` file to a `<resource-provider>` element in an individual application's `orion-application.xml` file (or the default application's `application.xml` file). The link-reference is the value of the `<resourceadapter>`'s `resourceProviderName` config property. As mentioned previously, the link-key for a `<resource-provider>` element is the value of its `name` attribute (which is also the JNDI location under `java:comp/resource` where the resource provider reference defined by the `<resource-provider>` element will be bound). NOTE: The link-reference value in the `ra.xml` file is just a default, and for any given JMS Connector instance it may be overridden using the `<connector>`'s `resourceProviderName` config property in the JMS Connector instance's `oc4j-connectors.xml` file. The over-ride is generally not required and is not depicted with an arrow in the figure.
- **17** - This link acts as a default when a `<connector-factory>` linked to a `<connection-definition>` (see description for arrow #14) does not include a `jndiLocation` config property (see description for arrow #16). The link-reference is the value of the `<connection-definition>`'s `jndiLocation`

config property. The link-key is the JNDI location of the RP connection factory within the resource provider's JNDI context

Bypassing the JMS Connector for Application Clients

Most of the features provided by the JMS Connector are not applicable to application clients. In order to keep application clients as light-weight as possible, you may choose to not use the JMS Connector with application clients. An application client that is not using the JMS Connector can communicate with application components that are using the JMS Connector so long as the same underlying resource provider (RP) destination is used by both components. Bypassing the JMS Connector is accomplished by referencing resource provider resources instead of JMS Connector resources in the `orion-application-client.xml` file:

1. **Bypass JMS Connector Connection Factories -**

   For each `<resource-ref-mapping>` element, make the value of its `location` attribute be

   ```
   java:comp/resource/providerName/resourceName
   ```

   where *providerName* is equal to the link-key for arrow #18 in Figure 3–1, "JMS Infrastructure"
   and *resourceName* is equal to the link-key for arrow #16 in Figure 3-1.
   This replaces arrows #6, #18 and #16 in Figure 3-1 with a direct link to the RP connection factory, bypassing the JMS Connector's `oc4j-ra.mxl` and `ra.xml` files.

2. **Bypass JMD Connector Destinations -**

   For each `<message-destination-ref-mapping>` element, make the value of its `location` attribute be

   ```
   java:comp/resource/providerName/resourceName
   ```

   where *providerName* is equal to the link-key for arrow #12 in Figure 3–1, "JMS Infrastructure"
   and *resourceName* is equal to the link-key for arrow #13 in Figure 3-1.
   This replaces arrows #7, #12 and #13 in Figure 3-1 with a direct link to the RP destination, bypassing the JMS Connector's `oc4j-connectors.xml` file.

Some third-party tools or libraries that access JNDI directly may have rigid location limits and/or validation rules that do not allow for the `java:comp/resource` syntax and/or naming peculiarities of specific resource providers, such as the `Queues/` prefix and other prefixes used by OEMS JMS Database option. In that situation, the JMS Connector should not be bypassed. (In general this limitation does not apply to the OEMS JMS In-Memory and File-Based options. This is because, for these resources, the `resourceName` can be used in a JNDI lookup by itself. That is, the

```
java:comp/resource/providerName/resourceName
```

prefix is purely optional when using OEMS JMS In-Memory or File-Based options.)

An application must not pass JMS Connector destinations to any object derived from an RP connection factory, and must not pass RP destinations to any object derived from a JMS Connector connection factory. The JMS Connector automatically manages the conversion from one destination type to the other for the `JMSDestination` and `JMSReplyTo` header fields for all sent, received, and browsed messages. For example, if an application client that is not using the JMS Connector sets the `JMSReplyTo` header field for an RP message to an RP destination and sends the message, and another application component that is using the JMS Connector receives the message and reads the `JMSReplyTo` header field, then the receiver will get a compatible JMS Connector message and JMS Connector destination that wrap the original RP message and RP destination. There is no automatic conversion for any other case. For example, if that scenario were repeated, but instead of sending the message directly it was sent as the body of an ObjectMessage, then when the receiver extracted the body of the ObjectMessage it would get an RP message instead of an JMS Connector message, and the `JMSReplyTo` header field of that RP message would contain an RP destination rather than a JMS Connector destination.

Resource Providers

The underlying connection factories and destinations that application clients use to send and receive JMS messages are resource provider objects. In 10.1.3, OC4J uses the JMS Connector to plug in the OEMS JMS (In-Memory, File-Based, and Database), IBM MQ, TIBCO, and Sonic resource providers.

But ultimately the destinations and connection factories must be created in the resource provider.

In general, use the following steps to configure a resource provider:

- Declare Resource Provider References
- Create RP Connection Factories
- Create RP Destinations

Each JMS provider requires its own procedure for configuring the provider and creating connection factories and destination objects. For resource providers other than OEMS JMS, refer to the documentation for that provider.

- OEMS JMS In-Memory and File-Based resource provider connection factories and destinations are created and bound to JNDI in the `jms.xml` file. These OEMS JMS persistence options are discussed in "OEMS JMS In-Memory and File-Based Persistence", starting on page 3-18.
- The OEMS JMS Database persistence option connection factories and destinations are created using SQL procedures. The OEMS JMS Database option is discussed in "OEMS JMS Database Persistence", starting on page 3-41.

Declaring Resource Provider References

A client can use one or more different JMS resource providers, each with its own resource adapter, choosing according to the integration and quality-of-service (QOS) features desired.

You declare the references to the resource provider that you will use in one or more `<resource-provider>` elements of the `orion-application.xml` file and/or the `application.xml` file.

- OEMS JMS In-Memory and File-Based Persistence—These two OEMS JMS persistence options are installed with OC4J.
- OEMS JMS Database Persistence—The OEMS JMS Database persistence option is a feature of the Oracle database and is based on the Streams Advanced Queuing (AQ) messaging system.

  The benefits of the OEMS JMS Database persistence option are as follows:

  - It is backed by the Oracle database.
  - OEMS JMS Database persistence and other Oracle database transactions can be used together in one-phase commit transactions.
  - It also provides access to extra features provided by AQ including interoperability with PL/SQL and OCI.
- Using Third-Party JMS Providers—You can integrate with the following third-party JMS providers:
  - WebSphere MQ for JMS versions 6.0 and 5.3 resource provider
  - TIBCO Enterprise for JMS version 3.1.0
  - SonicMQ 6.0

Use one of the following files to declare a resource provider reference:

- To make a resource provider reference visible to all applications (global), then use the global `application.xml` file.
- To make a resource provider reference visible only to a single application (local), then use the `orion-application.xml` file specific to the application.

Add the following code to the appropriate XML file (as listed above):

```
<resource-provider class="providerClassName" name="providerName">
     <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.
  - For the OEMS JMS In-Memory and File-Based option, use: `com.evermind.server.jms.Oc4jResourceProvider`
  - For the OEMS JMS Database option, use: `oracle.jms.OjmsContext`
  - For all third party resource providers, use: `com.evermind.server.deployment.ContextScanningResourceProvider`
- `name`—A name by which to identify the resource provider. This name is used to map the resource provider's JNDI context in the application's JNDI as:

  ```
  java:comp/resource/providerName/
  ```

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

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

Before an application or a resource adapter running in OC4J can access a resource provider, a resource provider reference must be declared with the `<resource-provider>` element. The resource provider reference holds miscellaneous data (for example, the class name described below) that OC4J uses to interact with the resource provider. The resource provider reference also provides a JNDI subcontext through which resource provider resources can be accessed. The resource provider reference (and said JNDI access) can be made local to the application by placing it in `orion-application.xml`, or available to all applications by placing it in `%ORACLE_HOME%/j2ee/home/config/application.xml`.

The two pieces of information that you must provide whenever declaring a resource provider reference are the name you wish to use for the resource provider reference and the Java class that implements the resource provider interface.

The resource provider reference maps the resource provider's JNDI context, which contains resource provider connection factories and resource provider destinations, to a JNDI subcontext accessible by the application and, more importantly, the JMS Connector. The reason it is more important for the JMS Connector to be able to access resource provider resources than for the application to be able to do so is that, when using a JMS Connector, the application need not (and in general should not) directly look up or use any resource provider resources. That JNDI subcontext is `java:comp/resource/providerName` where `providerName` is the name of the resource provider reference.

The demo set at `http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html` includes examples of declaring resource provider references. For a list of the How-To documents and demo sets and their URLs, see "JMS How-To Documents and Demo Sets" on page 3-4.

Download and unzip the How-To-gjra-with-*xxx*.zip file, where *xxx* is the name of the relevant resource provider.

Drill down to the following file: `/src/META-INF/orion-application.xml`.

For more detail, search for "Configuring the Resource Provider" in the relevant How-To document.

In the demo, the data source and the resource provider are declared local to the application. If the data source definition is placed in `$J2EE_HOME/config/data-sources.xml` and the resource-provider definition is placed in `$J2EE_HOME/config/application.xml`, then they are visible to all applications. (Only the demo for the OEMS JMS Database Persistence provider requires or includes a data source.)

The following sections give more detail on declaring the resource provider reference for each resource provider supported by OC4J, including the Java class and an example resource provider reference name for each:

- OEMS JMS In-Memory and File-Based Persistence
- OEMS JMS Database Persistence
- Declaring the OEMS JMS Database Reference on page 3-44
- Declaring an IBM WebSphere MQ Resource Provider Reference on page 3-49
- Declaring a TIBCO Enterprise Message Service Resource Provider Reference on page 3-49
- Declaring a SonicMQ Resource Provider Reference on page 3-50

OEMS JMS In-Memory and File-Based Persistence

The OEMS JMS In-Memory and File-Based options provide the following features:

- Complies with the JMS 1.1 specification.
- Is compatible with the J2EE 1.4 specification.

- Offers a choice between in-memory or file-based message persistence.
- Provides an exception queue for undeliverable messages.

Note: If you see the term "OC4J JMS" or "OracleAS JMS" in the Application Server Control Console, in the MBeans, or in the sample code, this refers to the OEMS JMS In-Memory and File-Based persistence options.

This section covers the following topics:

Configuring Destination Objects and Connection Factories

Destination objects can be queues or topics. The OEMS JMS In-Memory and File-Based options are already installed with OC4J, so the only configuration necessary is for the custom destination objects and connection factories for your applications to use.

The primary tool for configuring Destination objects and connection factories is the Application Server Control Console. You can also edit the XML files directly.

Default Destination Objects and Connection Factories

The defaults are useful as follows:

- The connection factories can be copied and used as templates for connection factories that you create. The default connection factories can also be used without modification in production.
- The destinations can be copied and used as templates for destinations that you create. The default destinations should NOT be used in production.

Six default connection factories are created for the different combinations of XA (global transaction enabled), non-XA, and various JMS interfaces. Your applications can use these connection factories without your having to add them in the Application Server Control Console or the `jms.xml` file. The only reason to define a new connection factory is to specify non default values for one or more of the optional attributes of `connection-factory` elements.

The default connection factory objects are created internally by OC4J, which binds them to the default JNDI locations within the OC4J server where the JMS connection is created.

The following default connection factories are created, even though they are not explicitly defined in the `jms.xml` file. It is safer to treat these as reserved JNDI locations and to use other JNDI locations when you create custom connection factories.

	XA	non-XA
Default Queue Connection Factory	`jms/XAQueueConnectionFactory`	`jms/QueueConnectionFactory`
Default Topic Connection Factory	`jms/XATopicConnectionFactory`	`jms/TopicConnectionFactory`
Default Unified Connection Factory	`jms/XAConnectionFactory`	`jms/ConnectionFactory`

Default destinations are as follows:

Default	Destination
Default Queue	`jms/demoQueue`
Default Topic	`jms/demoTopic`

Configuring in the Application Server Control Console

The Application Server Control Console is the primary tool for configuring the OEMS JMS In-Memory and File-Based persistence option connection factories and destination objects. For each destination object, you must specify its name, location, and destination type (queue or topic).

Path to the Application Server Control Console:

OC4J:Home > Administration tab > Services > JMS Providers > Go To Task Configure OracleAS JMS > Select the appropriate tab.

Table 3–2, Configuration Elements table describes the OracleAS JMS resource provider configuration elements and their attributes.

Configuration Elements

Table 3–2 defines the configuration elements and shows where to make the settings in the Application Server Control Console, in the MBeans, and in the `jms.xml` file.

Table 3–2 Configuration Elements

Console and MBean Setting Locations	Element(s) of jms.xml	Description and Attributes
The `JMSAdministrator` MBean enables you to specify the server host name and port, and multiple related attributes and operations. **Path to the JMSAdministrator MBean**: OC4J:Home > Administration tab > Task Name: JMX. System MBean Browser. > Go To Task > Drill down: J2EEDomain:oc4j, J2EEServer:standalone, JMSAdministratorResource, `"JMSAdministrator"`	`<jms-server>`	The root element of the server configuration. The `<jms-server>` element takes the following attributes: `host` - The host name defined in a `String` (DNS or dot-notation host name) to which this server should bind. By default, the server binds to `0.0.0.0` (also known as `[ALL]` in the configuration file). Optional. `port` - The port defined as an `int` (valid TCP/IP port number) to which this server should bind. The default setting is `9127`. This setting applies only to the standalone configuration of OC4J. In the Oracle Application Server configuration, the port setting in the configuration file is overridden by command-line arguments that are used (by, for example, OPMN and others) when starting the OC4J server. Optional.
Create a resource provider destination and specify its attributes on the `Add Destinations` page. **Path to the Add Destinations page:** OC4J:Home > Administration tab > Task Name: Services.JMS Providers > Go To Task > Destinations tab > Create New	`<queue>`	This element configures queues. The queues are available when OC4J 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. The `<queue>` element takes the following attributes: `name` - This required attribute is the provider-specific name (`String`) for the queue. The name can be any valid non empty 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 two destinations to specify the same name. There is no default. `location` - This required attribute states the JNDI location (`String`) where the queue is bound. The value should follow the JNDI rules for valid names. `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/<group>` for Oracle Application Server environments and *`J2EE_HOME`*`/persistence` for standalone environments. Each queue and topic must have its own persistence file name. You must not have two objects writing to the same persistence file. The `persistence-file` attribute is discussed further at "Persistence Recovery" on page 3-33.

Table 3–2 (Cont.) Configuration Elements

Console and MBean Setting Locations	Element(s) of jms.xml	Description and Attributes
Create a topic destination and specify its attributes on the `Add Destinations` page. **Path to the Add Destinations page:** OC4J:Home > Administration tab > Task Name: Services.JMS Providers > Go To Task > Destinations tab > Create New	`<topic>`	This element configures a topic. The topics are available when OC4J 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. The `<topic>` element takes the following attributes: `name` - This required attributes is the provider-specific name (`String`) for the topic. The name can be any valid non empty 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 two destinations to specify the same name. There is no default. `location` - This required attribute states the JNDI location (`String`) where the topic is bound. The value should follow the JNDI rules for valid names. 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/<group>` for Oracle Application Server environments and *`J2EE_HOME`*`/persistence` for standalone environments. Each queue and topic must have its own persistence file name. You must not have two objects writing to the same persistence file. The `persistence-file` attribute is discussed further at "Persistence Recovery" on page 3-33.
The `Description` field is on the `Add Destination` page where you create the topic or queue to which the description applies.	`<description>`	A sub-element of `<queue>` or `<topic>`. A user-defined string to remind the user for what the queue or topic is used. Optional.

Table 3–2 (Cont.) Configuration Elements

Console and MBean Setting Locations	Element(s) of jms.xml	Description and Attributes

Create a connection factory and specify its attributes on the `Add Connection Factory` page.
 	
Path to Add or Edit a Connection Factory:
 	
OC4J:Home > Administration tab > Task Name: Services.JMS Providers > Go To Task > **Connection Factories tab** > Create New or Edit Properties
	`<connection-factory>`
	
	or
	
	`<queue-connection-factory>`
	
	or
	
	`<topic-connection-factory>`
		Connection factory configuration. A connection factory element takes the following attributes:
		
		■ `location` - Required. The JNDI location to which the connection factory is bound. The value must follow JNDI rules for valid names.
		
		■ `host` - Optional. The fixed OC4J host to which this connection factory will connect. By default, a connection factory uses the same host as configured for the `jms-server` element. Non default values can be used to force all JMS operations to be directed to a specific OC4J JVM, bypassing any locally available OC4J servers and other Oracle Application Server or clustered configurations. Optional, string, DNS or dot notation host name. Default = `ALL`
		
		■ `port` - Optional. The fixed port to which this connection factory connects. 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). Non default values can be used to force all JMS operations to be directed to a specific OC4J JVM, bypassing any locally available servers and other Oracle Application Server or clustered configurations. Optional, int, valid TCP/IP port number. Default = `9127`.
		
		■ `username` - Optional. The user name for the authentication of JMS default connections created from this connection factory. That is, if an application creates a connection and neither the application nor the `oc4j-ra.xml` file specifies a `username/password`, then the `username` and `password` attributes from this element will be used. The user name itself must be properly created and configured with other OC4J facilities. Optional, string. Default = the empty string.
		
		■ `password` - Optional. The password for the authentication of JMS default connections created from this connection factory. The password itself must be properly created and configured with other OC4J facilities. The property `password` attribute supports password indirection. For more information, refer to the ***Oracle Containers for J2EE Security Guide***. Optional, string. Default = the empty string.
		
		■ `clientID` - Optional. The administratively configured, fixed JMS `clientID` for connections created from this connection factory. If no `clientID` is specified, then the default is an empty string, which can also be programmatically overridden by client programs, according to the JMS specification. The `clientID` is used only for durable subscriptions on topics; its value does not matter for queue and nondurable topic operations. Optional, string. Default = the empty string.

Table 3–2 (Cont.) Configuration Elements

Console and MBean Setting Locations	Element(s) of jms.xml	Description and Attributes

Create an XA-enabled connection factory and specify its attributes on the `Add Connection Factory` page.
Path to Add or Edit a Connection Factory:
OC4J:Home > Administration tab > Task Name: Services.JMS Providers > Go To Task > **Connection Factories tab** > Create New or Edit Properties
	`<xa-connection-factory>`
	or
	`<xa-queue-connection-factory>`
	or
	`<xa-topic-connection-factory>`
		XA variants of connection factory configuration.
		The XA connection factory elements take the same attributes as the non-XA connection factory elements, which are described in the previous row.

	`<log>`	Enables logging of the JMS activity in either file or ODL format. See the section "Enabling OC4J Logging" in the ***Oracle Containers for J2EE Configuration and Administration Guide*** for information on logging.
Edit system properties settings in the `JMSAdministratorResource MBean`. **Path to system properties settings in JMSAdministratorResource MBean**: OC4J:Home > Administration tab > Task Name: JMX.System MBean Browser. > Go To Task > Drill down: J2EEDomain:oc4j, J2EEServer:standalone, JMSAdministratorResource, "JMSAdministrator" > Operations tab > setConfigProperty	`<config-properties>`	Sets system properties. The settings are persisted to the `jms.xml` file. `<config-property>` - Subelement of `<config-properties>`. These settings are discussed in "JMS Configuration Properties" on page 3-38.

Configuration Using jms.xml

The OEMS JMS In-Memory and File-Based configuration settings are persisted in the `jms.xml` file. The settings in `jms.xml` include:

- Connection factories
- Destinations
- JMS Router jobs
- Global configuration

Note: Remember that you must restart the OC4J instance to enable configuration changes made directly in the XML files.

The following example shows the structure of elements under `<jms-server>` within the `jms.xml` file. This example configures the following destinations and connection factories:

- The queue "MyQueue" at JNDI location `jms/MyQueue`
- The topic "MyTopic" at JNDI location `jms/MyTopic`
- A connection factory (unified) at JNDI location `jms/Cf`
- A queue connection factory at JNDI location `jms/Qcf`
- An XA topic connection factory at JNDI location `jms/xaTcf`.

```
<jms>
  <jms-server>

    <queue  name="MyQueue" location="jms/MyQueue" persistence-file="/tmp/MyQueue">
       <description>The demo queue. </description>
    </queue>

    <topic name="MyTopic" location="jms/MyTopic" persistence-file="/tmp/MyTopic">
      <description>The demo topic. </description>
    </topic>

    <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>


    <log>
      <file path="../log/jms.log" />
    </log>

    <config-properties>
      <config-property name="oc4j.jms.debug" value="true">
      </config-property>
    </config-properties>

  </jms-server>

   <jms-router>
      <!-- JMS router configuration is shown in the
      "JMS Router Configuration in jms.xml" section.
      -->
   </jms-router>

</jms>
```

A detailed example of the element structure under `<jms-router>` is available at "JMS Router Configuration in jms.xml" on page 3-84.

Configuring Ports

In a standalone OC4J instance, you can set the port range in the `JMSAdministrator` MBean. You must restart the OC4J instance for your changes to take effect. This restart requirement is a special case for port settings.

In the full Oracle Application Server environment (managed), use the Application Server Control Console to configure the port range.

Path to configure the port range in the Application Server Control Console:

OC4J:Home > Administration tab > Task Name: JVM Properties > JMS Ports

Sending and Receiving JMS Messages

The code for sending and receiving JMS messages is not dependent on the JMS Connector or the resource providers involved.

This example is from the `MyChannel.java` file in the demo sets at:
`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`

Drill down to the following file: `/src/common/MyChannel.java`

In the demo, `MyChannel.java` is the only class that sends or receives JMS messages. All other classes call `MyChannel` to do sends and receives. `MyChannel` is the same for all of the different resource providers. In fact, all of the .java source is the same for all of the resource providers except for some comments in `Player.java` that explain alternate JNDI locations (not based on logical names) that may be used for looking up connection factories and destinations

For a list of the How-To documents and demo sets and their URLs, see "JMS How-To Documents and Demo Sets" on page 3-4.

```
   public MyChannel(String connectionFactoryName, String destinationName) throws
Exception {

        Context ctx = new InitialContext();

        // Get the destination.
        Destination destination = (Destination) ctx.lookup(destinationName);

        // Get the connection factory.
        ConnectionFactory cf = (ConnectionFactory)
 ctx.lookup(connectionFactoryName);

        // Use the connection factory to create a connection.
        connection = cf.createConnection();

        // Start the connection.
        connection.start();

        // Use the connection to create a session.
        session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

        // Use the session and destination to create a message producer and a
message consumer.
```

```
        producer = session.createProducer(destination);
        consumer = session.createConsumer(destination);
    }

    /**
     * Send message.
     * prerequisite: channel is open
     * @param obj object to be sent
     */
    public void send(Serializable obj) throws JMSException {

        // Use the session to create a new message.
        ObjectMessage msg = session.createObjectMessage(obj);

        // Use the message producer to send the message.
        producer.send(msg);
        System.out.println("Sent message: " + obj);
    }

    /**
     * Receive message (wait forever).
     * prerequisite: channel is open
     * @return object which was received in message, or <code>null</code> if no
message was received
     */
    public Serializable receive() throws JMSException {

        // Use the message consumer to receive a message.
        ObjectMessage msg = (ObjectMessage) consumer.receive();

        System.out.println("Got message: " + msg.getObject());
        return msg.getObject();
    }

    /**
     * Receive message (wait a while).
     * prerequisite: channel is open
     * @param timeout maximum time (in milliseconds) to wait for a message to
arrive
     * @return object which was received in message,
     * or <code>null</code> if no message was received
     */
    public Serializable receive(long timeout) throws JMSException {

        // Use the message consumer to receive a message (if one comes in time).
        ObjectMessage msg = (ObjectMessage) consumer.receive(timeout);

        if (msg == null) return null;
        System.out.println("Got message: " + msg.getObject());
        return msg.getObject();
    }

    /**
     * Close channel.
     * prerequisite: channel is open
     * Once a MyChannel object is closed, it may no longer be used to send or
receive
     * messages.
     */
    public void close() throws JMSException {
        // Close the connection (and all of its sessions, producers and
consumers).
        connection.close();
    }

    private Connection connection;
    private Session session;
    private MessageProducer producer;
    private MessageConsumer consumer;
}
```

JMS Utility

In this release, JMS Utility functionality is available as attributes and operations on various MBeans, replacing the deprecated command line interface of previous releases.

JMS Utility functionality resides on the following MBeans:

- The `JMSAdministrator` MBean - Path:

  OC4J:Home > Administration tab > Task Name: JMX.System MBean Browser > Go To Task > Drill down: J2EEDomain:oc4j, J2EEServer:standalone, JMSAdministratorResource, "JMSAdministrator"

- The `JMS` MBean - Path:

  OC4J:Home > Administration tab > Task Name: JMX.System Bean Browser > Go To Task > Drill down: J2EEDomain:oc4j, J2EEServer:standalone, JMSResource, "JMS"

- Various JMSDestinationResource MBeans - Path:

  OC4J:Home > Administration tab > Task Name: JMX.System MBean Browser > Go To Task > Drill down: J2EEDomain:oc4j, J2EEServer:standalone, JMSResource, "JMS", JMSDestinationResource > Select the MBean that represents the desired destination.

Table 3–3 JMS Utility

MBean Implementation	Previous Command Line Command	Description
`configProperties` attribute in the `JMSAdministrator` MBean	`knobs`	Display all available system properties (shown in Table 3–4) and their current settings.

Table 3–3 (Cont.) JMS Utility

MBean Implementation	Previous Command Line Command	Description
Statistics tab in the `JMS` MBean	`stats`	The following OEMS JMS In-Memory and File-Based statistics are available through the `JMS` MBean: ■ `activeHandlers` ■ `activeConnections` ■ `pendingMessageCount` ■ `messageDequeued` ■ `messageExpired` ■ `messageCommitted` ■ `messageRolledBack` ■ `messageEnqueued` ■ `messageRecovered` ■ `messageDiscarded` ■ `messagePagedIn` ■ `messageCount`
`validateSelector` operation on the `JMSAdministrator` Mbean.	`check <selector>`	Check validity of the specified JMS message selector. The operation takes the argument `selector`.
`areSelectorsEqual` operation on the `JMSAdministrator` Mbean.	`check <sel1> <sel2>`	Check if two specified selectors are treated as equivalent. This is useful for reactivating durable subscriptions. The operation takes the arguments: `sel1` and `sel2`.
`subscribe` operation on all JMSDestinationResource MBeans whose domain = `topic`	`subscribe`	Creates a durable subscription on the destination. This replaces existing, inactive durable subscriptions. The operation takes the following arguments: ■ `name` - name of the durable subscriber ■ `noLocal` - if `true`, allows subscriber to inhibit delivery of messages published by its own connection ■ `xact` - if `true`, session will be transacted ■ `clientId` - the client id ■ `selector` - the message selector
`unsubscribe` operation on all JMSDestinationResource MBeans whose domain = `topic`	`unsubscribe`	Removes the durable subscription. The operation takes the following arguments: ■ `name` - name of the durable subscriber ■ `xact` - if `true`, session will be transacted ■ `clientId` - the client id

Table 3–3 (Cont.) JMS Utility

MBean Implementation	Previous Command Line Command	Description
`browse` operation on all JMSDestinationResource Mbeans	`browse`	Browse this destination. The operation takes the following arguments: ■ `sub` - name of the durable subscriber, only available for MBeans where domain = `topic` ■ `xact` - if `true`, session will be transacted ■ `clientId` - the client ID (optional) ■ `selector` - the message selector (optional) ■ `count` - the maximum number of messages to process (`0` for all)
`copy` operation on all JMSDestinationResource Mbeans	`copy`	Copies messages from this destination to the specified destination. The operation takes the following arguments: ■ `sub` - name of the durable subscriber, only available for MBeans where domain = `topic` ■ `toDestination` - the destination to move the messages to ■ `xact` - if `true`, session will be transacted ■ `clientID` - the client ID (optional) ■ `selector` - the message selector (optional) ■ `count` - the maximum number of messages to process (`0` for all)
`drain` operation on all JMSDestinationResource Mbeans	`drain`	Drain messages from this destination. The operation takes the following arguments: ■ `sub` - name of the durable subscriber, only available for MBeans where domain = `topic` ■ `xact` - if `true`, session will be transacted ■ `clientId` - the client ID (optional) ■ `selector` - the message selector (optional) ■ `count` - the maximum number of messages to process (`0` for all)

Table 3–3 (Cont.) JMS Utility

MBean Implementation	Previous Command Line Command	Description
`move` operation on all JMSDestinationResource Mbeans	`move`	Moves messages from this destination to the specified destination. The operation takes the following arguments: ■ `sub` - name of the durable subscriber, only available for MBeans where domain = `topic` ■ `toDestination` - the destination to move the messages to ■ `xact` - if `true`, session will be transacted ■ `clientID` - the client ID (optional) ■ `selector` - the message selector (optional) ■ `count` - the maximum number of messages to process (`0` for all)

Configuring File-Based Persistence

The following sections discuss file-based persistence:

- Enabling File-Based Persistence in the Application Server Control Console
- Enabling File-Based Persistence in the jms.xml File
- Persistence Recovery

When file-based persistence is enabled OC4J automatically performs the following:

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

Caution:

A persistence file must not be copied, deleted, or renamed when the OC4J server is active. Doing so can result in data corruption and message loss.

If OC4J is not active, then deleting 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 re initializes the file as usual.

For more information on this, see "Persistence File Management" on page 3-34.

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

- The destination object is defined to be persistent by specifying a persistence file in the Application Server Control Console or by setting the destination's `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 persisted.
- The destination is a queue, or the destination is a topic and the consumer is a durable subscriber.

Setting the `DeliveryMode` to `PERSISTENT` or `NON_PERSISTENT` is described in the JMS specification.

Setting the default `DeliveryMode` for a message producer is described at:
`http://java.sun.com/j2ee/1.4/docs/api/javax/jms/MessageProducer.html#setDeliveryMode(int)`

Setting a per-message `DeliveryMode` (over-riding the default) is described at:
`http://java.sun.com/j2ee/1.4/docs/api/javax/jms/MessageProducer.html#send(javax.jms.Destination,%20javax.jms.Message,%20int,%20int,%20long)`

and at:
`http://java.sun.com/j2ee/1.4/docs/api/javax/jms/MessageProducer.html#send(javax.jms.Message,%20int,%20int,%20long)`

Notes on Enabling File-Based Persistence

Given that the previously-listed conditions are met, the file-based persistence option features recoverable and persistent storage of messages. Each destination 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.

Enabling File-Based Persistence in the Application Server Control Console The Application Server Control Console is the primary tool for enabling file-based persistence for destination objects. Use the following path to specify the parameters of the persistence file in the Application Server Control Console.

Path to specify persistence files for destinations in the Application Server Control Console:

OC4J:Home > Administration tab > Task Name: Services, JMS Providers:, Go To Task > Destinations > Create New > "Persistence File"

You can specify a persistence file in the Application Server Control Console when creating a new destination. You cannot modify the persistence file specification for an existing destination in the Console. You can modify the persistence specification in the `jms.xml` file. See Enabling File-Based Persistence in the jms.xml File on page 3-32.

Enabling File-Based Persistence in the jms.xml File You can enable file-based persistence for destination objects, by specifying the `persistence-file` attribute in the `jms.xml` file.

The following XML configuration example demonstrates how the `persistence-file` attribute defines the name of the file as `pers`.

```
<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 OC4J server will not create any directories for persistence files. So when a persistence file is defined in `jms.xml` it must either be in an existing absolute directory, for example:

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

or simply be a filename for example:

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

In the latter case, by default the persistence file will be created in `$J2EE_HOME/persistence` (for a standalone instance) or `$J2EE_HOME/persistence/<group_name>` (in the full Oracle Application Server environment).

The `persistence-file` attribute is discussed in Table 3–2, " Configuration Elements" on page 3-21.

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

Persistence Recovery The following sections discuss the various aspects of persistence recovery:

- Scope of Recoverability
- Persistence File Management
- Reporting Errors to the JMS Client
- Recovery Steps

Scope of Recoverability

The OEMS JMS File-Based persistence option can recover from some but not all possible failures. If any of the following failures occurs, then recoverability of the persistence file is not guaranteed:

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

Persistence File Management

When the 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 the persistence files when they are being used.

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

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

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

In addition to persistence files specified by a user and lock files, the OEMS JMS In-Memory and File-Based options use a special file, `jms.state`, for internal configuration and transaction state management. The OEMS JMS server 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 depending on whether you are operating OC4J in standalone or in Oracle Application Server mode, as follows:

- Standalone: *`J2EE_HOME`*`/persistence` directory
- Oracle Application Server: *`J2EE_HOME`*`/persistence/<group_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:

1. Client makes a function call
2. Pre-persistence operations
3. Persistence occurs
4. Post-persistence operations
5. Client function call returns

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

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

Abnormal Termination

If OC4J terminates normally, then the lock files are cleaned up automatically. However, if OC4J terminates abnormally, for example, a `kill -9` command, then the lock files remain in the file system. OC4J can usually recognize leftover lock files. If not, you must manually remove lock files before restarting OC4J after abnormal termination.

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

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 persistence file in the same location, then they could overwrite each other's data and cause corruption or loss of persisted JMS messages. To protect against such sharing violations, OEMS 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 "Configuring File-Based Persistence" on page 3-31 for more information on persistence files.

OC4J must have appropriate permissions to create and delete the lock file.

On termination and restart, one of the following lock-file scenarios will apply:

- Normal termination - Lock files are automatically cleaned up. Restart proceeds normally.
- Abnormal termination - On restart, lock files are recognized. Restart proceeds normally.
- Abnormal termination - On restart, a CRITICAL message is delivered indicating a sharing violation. Restart cannot continue. Delete the lock file indicated in the error message and restart. If more than one lock file is involved, you may have to do this once for each.

JMS persistence lock files are tagged with (contain) server and persistence directory location info. If the lock file exists when the JMS server starts, and the lock file was created by the same server (having the same ip address) and using the same persistence directory location, then the JMS server will assume control of the lock file and start up successfully.

The remainder of this discussion of OEMS JMS File-Based recovery steps in this subsection assumes that all lock files in question have been removed.

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

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

The JMS server 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 configuration, and start the server in a clean-slate mode. (Oracle does not recommend doing this.) If the JMS server cannot find a `jms.state` file, then it creates a new one.

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

If messaging activity was in progress during abnormal termination, then OEMS JMS tries to recover its persistence files. Any data corruption (of the types mentioned earlier) is handled by clearing out the corrupted data; this may lead to a loss of messages and transactions.

If the headers of a persistence file are corrupted, OEMS 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 3–4, " System Properties" on page 3-38) instructs the JMS server to proceed with recovery, clearing out all invalid data at the cost of losing messages or masking user configuration errors.

Predefined Exception Queue

As an extension to the JMS specification, OEMS JMS In-Memory and File-Based options come with a predefined exception queue for handling undeliverable messages. This is a single, persistent, global exception queue to store undeliverable messages in originating in any OEMS JMS destination. The exception queue has the following fixed properties:

- A fixed name - `jms/Oc4jJmsExceptionQueue`
- A fixed JNDI location - `jms/Oc4jJmsExceptionQueue`
- A fixed persistence file - `Oc4jJmsExceptionQueue`

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

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

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

The exception queue is always available to the JMS server 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 are reserved words in their respective name spaces. Any attempt to define other entities with these names is an error.

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

Message Expiration By default, if a message sent to a persistent destination expires, then it is moved to the exception queue. The `JMSXState` of the expiring message is set to the value `3` (indicating `EXPIRED`), but the message headers, properties, and body are not otherwise modified. The message is wrapped in an `ObjectMessage` and the wrapping message is sent to the exception queue.

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

By default, messages expiring on non persistent or temporary destination objects are not moved to the exception queue. Normally, the messages sent to these destination objects are considered not worth persisting and neither are their expired versions.

You can specify that all expired messages be sent to the exception queue, regardless of whether they are sent to persistent, non persistent, or temporary destination objects, by setting the `oc4j.jms.saveAllExpired` administration property, described in Table 3–4, " System Properties" on page 3-38, to `true` when starting the OC4J server. In this case, all expired messages are moved to the exception queue. Even though this causes non persistent messages to be sent to the exception queue, it does not change their non persistent nature.

Message Paging

The OEMS JMS In-Memory and File-Based options support 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 "Configuring File-Based Persistence" on page 3-31).
- The destination is a queue, or the destination is a topic and the consumer is a durable subscriber.
- The amount of used memory in the OC4J server's JVM is above some user-defined threshold.

Only message bodies are paged. Message headers and properties are never paged. You can set the paging threshold through the system property, `oc4j.jms.pagingThreshold`, described in Table 3–4, " System Properties" on page 3-38.

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`, the JMS server tries to page out as many message bodies as possible until the memory usage fraction reduces below the threshold, or no more message bodies can be paged out.

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

If the memory usage fraction drops below the paging threshold, then the JMS server stops paging out message bodies. The bodies of messages already paged out are not automatically paged back in. The paging in of message bodies happens only on demand (that is, when a message is dequeued or browsed by a client).

By default, the paging threshold is set to `1.0`. In other words, by default, message bodies are never paged.

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

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

JMS Configuration Properties

Runtime configuration of the OEMS JMS In-Memory and File-Based options and JMS clients is accomplished through JVM system properties. None of these properties affect basic JMS functionality. They pertain to features, extensions, and performance optimizations that are specific to the JMS server. These are the properties that you see when you use the knobs command line command.

The primary tool for editing configuration properties at runtime is the `JMSAdministrator` MBean.

As a secondary method, in standalone, you can pass the configuration properties in as a command line argument as follows:

```
java -D<propertyname>=<value>
```

These property settings are persisted in the `jms.xml` file.

Path to JMS configuration properties settings in the JMSAdministratorResource MBean:

OC4J:Home > Administration tab > Task Name: JMX.System MBean Browser > Go To Task > Drill down: J2EEDomain:oc4j, J2EEServer:standalone, JMSAdministratorResource, "JMSAdministrator" > Operations tab > setConfigProperty

Table 3–4 lists and describes the system properties for the OEMS JMS resource provider In-Memory and File-Based options.

Table 3–4 System Properties

JVM System Property	Server/ Client	Description
`oc4j.jms.serverPoll`	JMS client	Interval (in milliseconds) that JMS connections ping the OC4J server and report communication exceptions to exception listeners. Type =long. Default = `15000`.
`oc4j.jms.messagePoll`	JMS client	Maximum interval (in milliseconds) that JMS consumers wait before checking for new messages. Type =long. Default = `1000`.

Table 3–4 (Cont.) System Properties

JVM System Property	Server/ Client	Description
`oc4j.jms.listenerAttempts`	JMS client	Number of listener delivery attempts before the message is declared undeliverable. This property only limits the number of delivery attempts when the message is being received by means of a MessageListener registered with JMS using one of the `setMessageListener()` methods. It does not limit the number of delivery attempts when a message is being received by means of one of the receive methods (except in the case where the message was already declared undeliverable by a previous delivery attempt to a MessageListener). Note that the JMS Connector implements inbound messaging for MDBs using the receive methods, so this property does not apply in that situation. Furthermore, due to J2EE 1.4 restrictions on the use of Message Listeners, this property is only applicable to application clients. For MDBs, the `MaxDeliveryCnt` activation spec property should be used to limit message delivery attempts. The `orion-ejb-jar.xml` demo file contains comments describing the `MaxDeliveryCnt` property. Type =int. Default = `5`.
`oc4j.jms.maxOpenFiles`	OC4J server	Maximum number of open file descriptors for persistence files. This is relevant if the server is configured with more persistent destination objects than the maximum number of open file descriptors allowed by the operating system. Type =int. Default = `64`.
`oc4j.jms.saveAllExpired`	OC4J server	Save all expired messages on all destination objects (persistent, nonpersistent, and temporary) to the exception queue. Type =boolean. Default = `false`.
`oc4j.jms.socketBufsize`	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. Type =int. Default = `64*1024`.
`oc4j.jms.debug`	JMS client	If `true`, enable tracing of `NORMAL` events in JMS clients and the 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. Type =boolean. Default = `false`.
`oc4j.jms.noDms`	JMS client	If `true`, disable instrumentation. Type =boolean. Default = `false`.

Table 3–4 (Cont.) System Properties

JVM System Property	Server/ Client	Description
`oc4j.jms.forceRecovery`	OC4J server	If `true`, forcibly recover corrupted persistence files. By default, the 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 JMS server almost always to start up correctly and make persistence files and destination objects available for use. Type =boolean. Default = `false`.
`oc4j.jms.pagingThreshold`	OC4J server	Represents the memory usage fraction above which the JMS server begins to consider message bodies for paging. This value is an estimate of the fraction of memory in use by the JVM. This value can range from `0.0` (the program uses no memory at all) to `1.0` (the program is using all available memory). Type =double. Default = `1.0`. See "Message Paging" on page 3-37 for more information.

Setting JMS Configuration Properties in the jms.xml File

The following fragment shows how a configuration property is set in the `jms.xml` file.

```
<config-properties>
  <config-property name="oc4j.jms.debug" value="true">
  </config-property>
</config-properties>
```

This fragment is shown in context in the `jms.xml` example at "Configuration Using jms.xml" on page 3-24. It is also covered in Table 3–2, " Configuration Elements" on page 3-21.

Resource Naming for OEMS JMS In-Memory and File-Based

The variable *resourceName* is used in this section to represent the JNDI location of an RP resource (connection factory or destination) within the resource provider's JNDI context.

The *resourceName* of an OEMS JMS resource is specified as a JNDI location in the console as described in Configuring Destination Objects and Connection Factories on page 3-19. Connection factory *resourceName* values are used to identify a specific OEMS JMS connection factory (as the link-key for arrow #16 in Figure 3–1). Destination *resourceName* values are used to identify a specific OEMS JMS destination (as the link-key for arrow #13 in Figure 3–1). Connection factory and destination *resourceName* values may also be used when bypassing the JMS Connector as discussed in Bypassing the JMS Connector for Application Clients on page 3-15.

Required Class Path for Application Clients Using Direct OEMS JMS In-Memory and File-Based Lookup

When using OEMS JMS In-Memory and File-Based options directly from an application client, the JAR files that must be included in the class path are listed inTable 3–5, " Client-side JAR Files Required for OEMS JMS In-Memory and File-Based Lookup".

Note: For 10g Release 3 (10.1.3), OEMS JMS cannot be used for global transactions without the JMS Connector.

Table 3–5 Client-side JAR Files Required for OEMS JMS In-Memory and File-Based Lookup

JAR	ORACLE_HOME Path
`oc4jclient.jar`	`/j2ee/<instance>`
`jta.jar`	`/j2ee/<instance>/lib`
`jms.jar`	`/j2ee/<instance>/lib`
`jndi.jar`	`/j2ee/<instance>/lib`
`javax77.jar`	`/j2ee/<instance>/lib`
`optic.jar` (Required only if the `opmn:ormi` prefix is used in Oracle Application Server environment.)	`/opmn/lib`

OEMS JMS Database Persistence

The OEMS JMS Database persistence option is the JMS interface to the Oracle Database Streams Advanced Queuing (AQ) feature in the Oracle database. This section will cover in detail the configuration and usage of OEMS JMS using AQ as the persistent store for messages.

As with the OEMS JMS In-Memory and File-Based options, Oracle recommends that your applications use the JMS Connector when accessing AQ through the OEMS JMS Database option.

For details about configuring OEMS JMS with AQ, see the *Oracle Streams Advanced Queuing User's Guide and Reference*.

This section describes the following topics:

- Using the OEMS JMS Database Option
- Using OEMS JMS Database with the Oracle Application Server and the Oracle Database

---

Note:

In past Oracle Application Server documentation and collateral, as well as in the Oracle Streams Advanced Queuing User's Guide and Reference, you will see the OEMS JMS Database persistence option described as "OJMS". When you encounter the acronym, "OJMS", it is describing the OEMS JMS Database persistence option.

Using the OEMS JMS Database Option

To create and access OEMS JMS Database resource provider destination objects (queues and topics), do the following:

- Install and configure OEMS JMS on the database. See "Install and Configure OEMS JMS Database" on page 3-42.
- On the database, create an RDBMS user. The JMS application will connect the RDBMS user to the back-end database and assign privileges. See "Create User and Assign Privileges" on page 3-42.
- Create the JMS destination objects. See "Creating OEMS JMS Database Destination Objects" on page 3-43.
- Create data sources or LDAP directory entries, if needed. See "Declaring the OEMS JMS Database Reference" on page 3-44.
- In the OC4J XML configuration, define the OEMS JMS Database option in the `<resource-provider>` element of the `orion-application.xml` file with information about the back-end database.
- Access the resource in your implementation through a JNDI lookup. See "Sending and Receiving JMS Messages" on page 3-26.

Install and Configure OEMS JMS Database You or your DBA must install OEMS JMS according to the *Oracle Streams Advanced Queuing User's Guide and Reference* and generic database manuals. After you have installed and configured OEMS JMS, you must apply additional configuration. This includes the following:

1. You or your DBA must create an RDBMS user through which the JMS client connects to the database. Grant this user appropriate access privileges to perform OEMS JMS operations. OEMS JMS 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 3-42.
2. You or your DBA must create the tables and queues to support the JMS destination objects. See "Creating OEMS JMS Database Destination Objects" on page 3-43.

Note: The following sections use SQL for creating queues, topics, and their tables, and for assigning privileges.

For examples, see `http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`.

Create User and Assign Privileges Create an RDBMS user through which the JMS client connects to the database. Grant access privileges to this user to perform OEMS JMS operations. The privileges that you need depend on what functionality you are requesting. Refer to the *Oracle Streams Advanced Queuing User's Guide and Reference* 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 OEMS JMS operations. You must be a `SYS DBA` to execute these statements.

```
DROP USER jmsuser CASCADE ;

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

```
connect jmsuser/jmsuser;
```

You may need to grant other privileges, such as two-phase commit or system administration privileges, based on what the user needs. See Chapter 5, "OC4J Transaction Support", for information on two-phase commit privileges.

Creating OEMS JMS Database Destination Objects Refer to the *Oracle Streams Advanced Queuing User's Guide and Reference* for information on the `DBMS_AQADM` packages and OEMS JMS Database messages types.

> **Note:** For examples, see `http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`.

The following examples demonstrate creating a queue and a topic in OEMS JMS Database.

1. Create the table that handles the OEMS JMS destination (queue or topic).

   Both topics and queues use a queue table. The following SQL example creates a single table, `demoTestQTab,` for a queue.

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

   The `multiple_consumers` parameter specifies whether there are multiple consumers. Set `multiple_consumers` to `false` for a queue. Set `multiple_consumers` to `true` for a topic.

2. Create the JMS destination. This SQL example creates a queue called `demoQueue` within the queue table `demoTestQTab` and then starts the queue.

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

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

Example:

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.

> **Note:** OEMS JMS Database 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.START_QUEUE('demoTopic');
```

Note:

OEMS JMS Database incorporates the (`Queue_name`) names passed into the `DBMS_AQADM.CREATE_QUEUE` method into the JNDI names for the destinations. For example, OEMS JMS Database makes the queue created with `Queue_name` "`demoQueue`" available with the JNDI name "`Queues/demoQueue`".

In order to wrap an OEMS JMS destination with a JMS Connector destination, the OEMS JMS-provided JNDI name for the destination and the "`jndiName`" `<config-property>` for the JMS Connector destination defined in the `oc4j-connectors.xml` file (either the global one in `$J2EE_HOME/config` or a local one optionally contained in an application's .ear file) must match. See arrow #13 in Figure 3–1, "JMS Infrastructure"

Declaring the OEMS JMS Database Reference

For an overview of declaring the resource provider reference, see "Declaring Resource Provider References" on page 3-16.

The two pieces of information you must provide whenever declaring a resource provider reference are the name you wish to use for the resource provider reference and the Java class that implements the resource provider interface.

For OEMS JMS Database, the class is `oracle.jms.OjmsContext`. To declare a resource provider reference named `OJMSReference`, use:

```
<resource-provider class="oracle.jms.OjmsContext" name="OJMSReference">
...
</resource-provider>
```

For example, if the OEMS JMS reference is named `OJMSReference`, and the JNDI location (within the resource provider's JNDI context of a resource provider queue is `Queues/MY_QUEUE`, then that resource provider queue is accessible to the application and resource adapter at JNDI location `java:comp/resource/OJMSReference/Queues/MY_QUEUE`.

In general, use the following steps to declare an OEMS JMS reference:

1. First create a local data-source in the `data-sources.xml` file. The demo set has an example of this at: `/ojms/src/META-INF/data-sources.xml`.
2. Then tell OC4J where the `data-sources.xml` file has been placed by adding a `<data-sources>` element to `orion-application.xml`:

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

   Note that the data-sources path is relative to the `orion-application.xml` file.
3. Finally, set the datasource for the resource provider reference by adding a `<property>` subelement to the previously created `<resource-provider>` element:

   ```
   <resource-provider class="oracle.jms.OjmsContext" name="OJMSReference">
   ```

```
        <property name="datasource"
value="jdbc/xa/MyChannelDemoDataSource"></property>
    </resource-provider>
```

Configuring data sources is discussed in the "Data Sources" chapter of this guide. When selecting which driver to use (OCI or thin), it is best to measure actual application performance. The OCI driver may be faster for non-XA operations, but can be significantly slower than the thin driver for XA operations.

For a list of the How-To documents and demo sets and their URLs, see "JMS How-To Documents and Demo Sets" on page 3-4.

Resource Naming for OEMS JMS Database The *resourceName* of an OEMS JMS Database resource is:

typeName/instanceName

The values for *typeName* and *instanceName* depend on the type of resource (connection factory or destination), and are described next.

For OEMS JMS Database connection factories:

typeName corresponds to the connection factory type and is one of the following:

- `ConnectionFactories`
- `QueueConnectionFactories`
- `TopicConnectionFactories`
- `XAConnectionFactories`
- `XAQueueConnectionFactories`
- `XATopicConnectionFactories`

instanceName can be anything (because it is ignored - OEMS JMS Database connection factories are not customizable so there is no need for multiple instances of the same connection factory type).

For example, the *resourceName* for an OEMS JMS Database non-XA queue connection factory (where the ignored *instanceName* is arbitrarily set to `myQCF`) is:

```
QueueConnectionFactories/myQCF
```

Connection factory *resourceName* values created as per the above instructions are used to identify a specific OEMS JMS Database connection factory (that is, as the link-key for arrow #16 in Figure 3–1, "JMS Infrastructure"). They may also be used when bypassing the resource adapter as discussed in Bypassing the JMS Connector for Application Clients on page 3-15.

The following table shows the `javax.jms.*` interfaces implemented.

Table 3–6 javax.jms.* Interfaces Implemented

typeName	CF	QCF	TCF	XACF	XAQCF	XATCF
ConnectionFactories	X					
QueueConnectionFactories	X	X				
TopicConnectionFactories	X		X			

Table 3–6 (Cont.) javax.jms.* Interfaces Implemented

typeName	CF	QCF	TCF	XACF	XAQCF	XATCF
XAConnectionFactories	x			x		
XAQueueConnectionFactories	x	x		x	x	
XATopicConnectionFactories	x		x	x		x

If the application requires a `javax.jms.TopicConnectionFactory` (as specified by the `<res-type>` element), the only type names that will return a suitable connection factory are `"TopicConnectionFactories"` and `"XATopicConnectionFactories"`.

For OEMS JMS Database destinations:

typeName corresponds to the destination type and is one of the following:

- `Queues`
- `Topics`

Note: A destination is a queue (*typeName* = `Queues`) if the `multiple_consumers` parameter passed to `DBMS_AQADM.CREATE_QUEUE_TABLE` was `false`. Otherwise the destination is a topic (*typeName* = `Topics`).

instanceName is the name of the destination (the Queue_Name parameter provided to `DBMS_AQADM.CREATE_QUEUE`). If the destination is not owned by the user specified in the `username` property in the `<resource-provider>` element, then *instanceName* must be prefixed with "*owner*." where *owner* is the owner of the destination. (Even when not required, the "*owner*." prefix is still allowed.)

For example, the *resourceName* for an OEMS JMS Database queue given the name `demoQueue` in the call to `DBMS_AQADM.CREATE_QUEUE` is:

```
Queues/demoQueue
```

If the *owner* (for example, `someUser`) needs to be specified, then the *resourceName* would be:

```
 Queues/someUser.demoQueue
```

Destination *resourceName* values created according to the above instructions are used to identify a specific OEMS JMS Database destination (that is, as the link-key for arrow #13 in Figure 3–1). They may also be used when bypassing the resource adapter as discussed in Bypassing the JMS Connector for Application Clients on page 3-15.

Sending and Receiving Messages Using OEMS JMS Database Persistence Oracle recommends that the application use the JMS Connector for sending and receiving messages. In this way the sending and receiving code can be independent of the resource provider used. The examples at "Sending and Receiving JMS Messages" on page 3-26 can be used for all resource providers, including OEMS JMS Database, with only the passed-in destination and connection factory locations being different.

Required Class Path for Application Clients Using Direct OEMS JMS Database Lookup

When using OEMS JMS Database options directly from an application client, the JAR files that must be included in the class path are listed inTable 3–7, " Client-side JAR Files Required for OEMS JMS Database Lookup".

Table 3–7 Client-side JAR Files Required for OEMS JMS Database Lookup

JAR	ORACLE_HOME Path
`oc4jclient.jar`	`/j2ee/<instance>`
`ejb.jar`	`/j2ee/<instance>/lib`
`jta.jar`	`/j2ee/<instance>/lib`
`jms.jar`	`/j2ee/<instance>/lib`
`jndi.jar`	`/j2ee/<instance>/lib`
`javax77.jar`	`/j2ee/<instance>/lib`
`adminclient.jar`	`/j2ee/<instance>/lib`
`ojdbc14dms.jar`	`/j2ee/<instance>/../../oracle/jdbc/lib`
`dms.jar`	`/j2ee/<instance>/../../oracle/lib`
`bcel.jar`	`/j2ee/<instance>/lib`
`optic.jar` (Required only if the `opmn:ormi` prefix is used in Oracle Application Server environment.)	`/opmn/lib`

Using OEMS JMS Database with the Oracle Application Server and the Oracle Database

This section discusses common issues encountered by users of OEMS JMS Database with Oracle Application Server.

- Error When Copying aqapi.jar
- OEMS JMS Database Certification Matrix

Error When Copying aqapi.jar A common error seen when using the OEMS JMS Database option with the Oracle Application Server is:

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

If you receive this message, then the `aqapi.jar` file being used in Oracle Application Server is not compatible with the version of the Oracle database being used for AQ. A common mistake is to copy the `aqapi.jar` file from the Oracle database installation into the Oracle Application Server installation, or from the Oracle Application Server installation into the Oracle database installation, under the false assumption that they are interchangeable. The confusion is because the Oracle Application Server and the Oracle database both ship the OEMS JMS client JAR file. Do not copy this file. Use the matrix in Table 3–8 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 OEMS JMS Database client JAR file can be found at *`ORACLE_HOME`*`/rdbms/jlib/aqapi.jar` and should be included in the `CLASSPATH`.

OEMS JMS Database Certification Matrix Table 3–8 indicates which versions of the Oracle database work with the Oracle Application Server when the OEMS JMS 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 cell has no **X**, then the corresponding version of the Oracle Database and Oracle Application Server should not be used together.

> **Note:** This is not a certification matrix for the Oracle Application Server and the Oracle Database in general. It is only for the OEMS JMS Database persistence option when used in the Oracle Application Server.

Table 3–8 OEMS JMS Database Certification Matrix

OracleAS / Oracle Database	**v9.0.1**	**v9.0.1.3**	**v9.0.1.4**	**v9.2.0.1**	**v9.2.0.2+**	**v10.1.0+**
9.0.2	X	X		X		
9.0.3			X			X
9.04			X		X	
9.0.4.1						X
10.1.2			X		X	X
10.1.3			X			

Using Third-Party JMS Providers

This section briefly discusses declaring references to the supported third-party JMS resource providers.

OC4J supports two-phase commit (2pc) for all supported resource providers as long as the resource provider has an XA interface and the JMS Connector and application are configured to use it. All of the supported third party providers have an XA interface.

The versions of each third-party JMS provider that OC4J supports are listed at Using Third-Party JMS Providers on page 3-48.

This section provides information on declaring references for the following third-party JMS providers:

- Declaring an IBM WebSphere MQ Resource Provider Reference
- Declaring a TIBCO Enterprise Message Service Resource Provider Reference
- Declaring a SonicMQ Resource Provider Reference

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

> **Note:** To declare a resource provider reference, use one or the other of the following files:
>
> - To make the resource provider reference visible to all applications (global), then use the global `application.xml` file.
> - To make the resource provider reference visible to a single application (local), then use the `orion-application.xml` file specific to the application.

Declaring an IBM WebSphere MQ Resource Provider Reference

WebSphere MQ is an IBM messaging provider. WebSphere MQ resources are available under

`java:comp/resource/`*`providerName`*

where *providerName* is the name used in the `<resource-provider>` element.

To declare a WebSphere MQ resource provider reference, perform the following steps:

1. Install and configure WebSphere MQ on your system, then verify the installation by running any examples or tools supplied by the vendor. See the documentation supplied with the WebSphere MQ software for instructions.
2. Add WebSphere MQ as a custom resource provider.

   The following example demonstrates using the `<resource-provider>` element in the `orion-application.xml` file to declare a WebSphere MQ resource provider reference.

   ```
   <resource-provider
       class="com.evermind.server.deployment.ContextScanningResourceProvider"
       name="MQSeries">
     <description> MQSeries resource provider </description>
     <property
         name="java.naming.factory.initial"
         value="com.sun.jndi.fscontext.RefFSContextFactory">
     </property>
     <property
         name="java.naming.provider.url"
         value="file:/home/developer/services_guide_demo/mqseries/src/bindings">
     </property>
   </resource-provider>
   ```

   This example shows how this configuration was accomplished in the demo set. It applies to the demo author's environment only and must be edited to work in any other environment.
3. Add to `J2EE_HOME/applib` any JAR files required by a Websphere MQ JMS client as described in the IBM documentation.

   The section "Resource Provider Task #3: Declare a Resource Provider Reference" in the `how-to-gjra-with-mqseries.html` document in the demo set has more detail.

For a list of the How-To documents and demo sets and their URLs, see "JMS How-To Documents and Demo Sets" on page 3-4.

Declaring a TIBCO Enterprise Message Service Resource Provider Reference

TIBCO Enterprise Message Service is a message provider from TIBCO Software. TIBCO resources are available under

`java:comp/resource/`*`providerName`*

where *providerName* is the name used in the `<resource-provider>` element.

To declare a TIBCO resource provider reference, perform the following steps:

1. Install and configure TIBCO Enterprise Message Service on your system, then verify the installation by running any examples or tools supplied by the vendor. See the documentation supplied with the TIBCO software for instructions.

2. Add TIBCO as a custom resource provider. The following example demonstrates using the `<resource-provider>` element in the `orion-application.xml` file to declare a TIBCO resource provider reference.

   ```
   <resource-provider
      class="com.evermind.server.deployment.ContextScanningResourceProvider"
      name="TibcoJMSReference">
       <property
           name="java.naming.factory.initial"
           value="com.tibco.tibjms.naming.TibjmsInitialContextFactory">
       </property>
       <property
           name="java.naming.provider.url"
           value="tibjmsnaming://jleinawe-sun:7222">
       </property>
   </resource-provider>
   ```

   This example shows how this configuration was accomplished in the demo set. It applies to the demo author's environment only and must be edited to work in any other environment.

3. Add to `J2EE_HOME/applib` any JAR files required by a TIBCO JMS client as described in the TIBCO documentation.

   The section "Resource Provider Task #3: Declare a Resource Provider Reference" in the `how-to-gjra-with-tibco.html` document in the demo set has more detail.

For a list of the How-To documents and demo sets and their URLs, see "JMS How-To Documents and Demo Sets" on page 3-4.

Declaring a SonicMQ Resource Provider Reference

SonicMQ is a messaging provider from Sonic Software Corporation. Sonic resources are available under

`java:comp/resource/`*`providerName`*

where *providerName* is the name used in the `<resource-provider>` element.

To declare a SonicMQ resource provider reference, 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 the Sonic software for instructions.

2. Add SonicMQ as a custom resource provider. The following example demonstrates using the `<resource-provider>` element in the `orion-application.xml` file to declare a SonicMQ resource provider reference.

   ```
   <resource-provider
      class="com.evermind.server.deployment.ContextScanningResourceProvider"
      name="SonicJMSReference">
       <property
           name="java.naming.factory.initial"
           value="com.sonicsw.jndi.mfcontext.MFContextFactory">
       </property>
       <property
           name="java.naming.provider.url"
           value="tcp://stadd41:2506">
       </property>
       <property
   ```

```
                name="com.sonicsw.jndi.mfcontext.domain"
                value="Domain1">
            </property>
        </resource-provider>
```

This example shows how this configuration was accomplished in the demo set. It applies to the demo author's environment only and must be edited to work in any other environment.

3. Add to `J2EE_HOME/applib` any JAR files required by a SonicMQ JMS client as described in the Sonic documentation.

   The section "Resource Provider Task #3: Declare a Resource Provider Reference" in the `how-to-gjra-with-sonic.html` document in the demo set has more detail.

For a list of the How-To documents and demo sets and their URLs, see "JMS How-To Documents and Demo Sets" on page 3-4.

JMS Connector

Oracle provides a J2CA 1.5-compliant resource adapter called the JMS Connector that allows OC4J-managed applications to have a unified mechanism to access any JMS provider, regardless of whether their level of J2CA support is at version 1.5. The JMS Connector does not use any Oracle proprietary APIs. Supported JMS implementations include OEMS JMS and third-party products such as IBM Websphere MQ JMS, TIBCO Enterprise for JMS, and SonicMQ JMS.

The JMS Connector is the recommended path for JMS usage in the OC4J 10.1.3 implementation. It is based on the J2CA 1.5 and JMS 1.1 and 1.02b standards and includes minimal customization for OC4J, and none for individual JMS providers. It is intended to seamlessly integrate any standard JMS implementation.

(Note that the JMS Connector does not typically provide optimal access to a particular JMS provider, given that many JMS providers support custom extensions.)

The JMS Connector includes features in the following areas:

- JNDI mapping
- MDB integration (including dynamic adjustment to changing message load)
- Global transaction support (including standards-based support for transaction recovery). Transaction support is discussed in Chapter 5, "OC4J Transaction Support".
- True generic JMS connection pooling
- Deployment convenience (including order independence)
- Lazy resolution of JMS operations (including start order independence, tolerance of dynamic management such as starts and stops of JMS providers, and connection retries in case of provider failure)
- Performance
- JSR-77 statistics

Typically, the JMS Connector is used in situations where the EIS being connected is a JMS resource provider. However, it can also be used in situations where the EIS uses JMS messaging as a means of notifying J2EE application components. In this case, the resource adapter (along with a JMS resource provider) can be used instead of the inbound communication features (if any) of the EIS-specific resource adapter. This two-adapter solution, where the EIS-specific adapter is used for outbound communication and the JMS Connector is used for inbound communication, enables bidirectional communication between the EIS and J2EE applications where it may otherwise not be possible.

For more information on resource adaptors see the *Oracle Containers for J2EE Resource Adapter Administrator's Guide*.
For information on using the Application Server Control Console for JMS Connector settings, see Chapter 2.
For information on configuring JMS Connector connection factories, see Chapter 3.

Modifying the JMS Connector

The JMS Connector provided with OC4J is configured out-of-the-box to support the OEMS JMS In-Memory and File-Based persistence options. If you need to integrate with the OEMS JMS Database option or one of the supported non-Oracle JMS providers then you must create another configuration for the JMS Connector. Another reason you may want to create another JMS Connector is to support an application-local adapter to connect to the OEMS JMS In-Memory and File-Based options.

For a list of the How-To documents and demo sets and their URLs, see "JMS How-To Documents and Demo Sets" on page 3-4.

Create and configure a new JMS Connector module as follows:

1. Go to the demo set for the resource provider to be connected.
2. Read the How-to document.
3. Copy the following files to be used as templates for your new JMS Connector module:
   - `ra.xml`
   - `oc4j-ra.xml`
   - `oc4j-connectors.xml`
4. Modify the `ra.xml`, the `oc4j-ra.xml`, and the `oc4j-connectors.xml` files according to the instructions at "Configuring the Resource Adapter" in the How-to document.
5. Create a new `.rar` file (You can name it whatever you want.) with the following structure:

   ```
   META-INF/ra.xml
   ```

   ```
   META-INF/oc4j-ra.xml
   ```
6. Place the new `oc4j-connectors.xml` in one of the following locations according to the desired level of visibility:
   - For application-local visibility, place the new `oc4j-connectors.xml` file in the top-level `META-INF/` directory of your application's `.ear` file

     For global visibility, copy the new `<connector>` element(s) into the `$J2EE_HOME/config/oc4j-connectors.xml` file. Make sure the name of your new connector (the `<connector>`'s `name` attribute) does not conflict with the name of other connectors or any other JNDI objects.
7. Prepare deployment as follows:
   - For local visibility, do the following:

- Place the new `.rar` file in the `.ear` file.
- Insert a `<connectors>` element that looks like the following into the `META-INF/orion-application.xml` file in the `.ear` file.

```
<connectors path="oc4j-connectors.xml"/>
```

- Insert a `<module>` element that looks like the following into the `META-INF/application.xml` file in the `.ear` file.

```
<module>
    <connector>rarFileName</connector>
</module>
```

- For global visibility, follow the instructions for deploying a connector in the *Oracle Containers for J2EE Deployment Guide*.

Configuring the JMS Connector

The Application Server Control Console is the primary tool for configuring the JMS Connector.

JMS Connector Connection Factories and Destinations

The default application defines the following destination objects for the default JMS Connector:

- A queue bound to JNDI location OracleASjms/MyQueue1.
- A topic bound to JNDI location OracleASjms/MyTopic1.
- An automatic destination wrapping JNDI subcontext for queues bound to JNDI location OracleASjms/Queues.
- An automatic destination wrapping JNDI subcontext for topics bound to JNDI location OracleASjms/Topics.

Your applications can use these destinations and automatic destination wrapping JNDI subcontexts without your having to add them in the Console or the `JMSAdministrator` MBean.

The JMS Connector default connection factories are as follows:

	XA	**non-XA**
Default Queue Connection Factory	`OracleASjms/MyXAQCF`	`OracleASjms/MyQCF`
Default Topic Connection Factory	`OracleASjms/MyXATCF`	`OracleASjms/MyTCF`
Default Unified Connection Factory	`OracleASjms/MyXACF`	`OracleASjms/MyCF`

The default connection factories in the table are explicitly declared in the default `oc4j-ra.xml` file that ships with OC4J).

Creating JMS Connector connection factories and destinations

For information on configuring the JMS Connector connection factories and destinations, including examples of the `oc4j-connectors.xml`, `oc4j-ra.xml`, and `ra.xml` files, go to:
`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`

Download and unzip the relevant how-to-gjra-with-*xxx*.zip file, where *xxx* is the name of the resource provider. Search for "Configuring the Resource Adapter" in the relevant How-To document.

For detailed reference information on the JMS Connector XML files, go to "Appendix A, OC4J Resource Adapter Configuration Files" of the *Oracle Containers for J2EE Resource Adapter Administrator's Guide*.

JMS Connector Settings

Table 3–9 lists and describes the JMS Connector configuration settings.

Path to JMS Connector Settings in the Application Server Control Console:

OC4J:Home > Applications tab > View: Standalone Resource Adapters > OracleASjms

Table 3–9 JMS Connector Configuration Settings

Console Settings	XML Persistence File(s)	Description
OC4J:Home > Applications tab > View: Standalone Resource Adapters > OracleASjms > **Connection Factories tab** > Create button **Connection Factory Interface** This is the link-reference for arrow #14 in Figure 3–1.	This setting is persisted in the `<connectionfactory-interface>` element of the `oc4j-ra.xml` file.	The Connection Factory Interface setting defines the type of connection factory to be created.
JNDI Location This is the link-key for arrow #5 in Figure 3–1.	This setting is persisted in the `location` attribute of a `<connector-factory>` element of the `oc4j-ra.xml` file.	The JNDI Location setting specifies the JNDI location where the new resource adapter connection factory is to be bound. Enter a valid JNDI location so the application component can locate the connection factory when it needs to connect to the EIS. Note that "JNDI Location" is not the same as "jndiLocation".

Table 3–9 (Cont.) JMS Connector Configuration Settings

Console Settings	XML Persistence File(s)	Description
Connection Pooling	This setting is persisted in the `<connection-pooling>` element of the `oc4j-ra.xml` file.	Connection pooling allows a set of connections to the EIS to be reused within an application. An application can choose to either create its own exclusive connection pool or use one of the shared connection pools available for this resource adapter. `No Connection Pool` - Select this option to disable connection pooling. `Use Private Connection Pool` - Select this option to create a new connection pool for exclusive use by the selected connection factory. `Use Shared Connection Pool` - Select this option to use a shared connection pool that can be used by multiple connection factories. The `oc4j-ra.xml` file contains OC4J-specific configuration for the JMS Connector. Subelements of `<connector-factory>` include `<connection-pooling>`, to set up connection pooling for the factory, and `<security-config>`, to set up container-managed sign-on. Each connector factory can have configuration for a private connection pool, or can use a shared connection pool that is set up through a `<connection-pool>` subelement of `<oc4j-connector-factories>`.
jndiLocation This is the link-reference for arrow #16 in Figure 3–1.	This setting is persisted in a `<config-property>` subelement of a `<connector-factory>` element of the `oc4j-ra.xml` file.	The `jndiLocation` setting specifies the resource provider connection factory to be wrapped by the resource adapter connection factory that you are creating. Note that "jndiLocation" is not the same as "JNDI Location".
`<none>`	This setting is persisted in a `<config-property>` subelement of a `<connector-factory>` element of the `oc4j-ra.xml` file.	The `clientId` setting specifies what client id should be applied to any newly created resource provider connection created by the JMS Connector connection.
OC4J:Home > Applications tab > View: Standalone Resource Adapters > OracleASjms > **Administered Objects tab** > Create button **Object Class**	This setting is persisted in an `<adminobject-class>` subelement of an `<adminobject-config>` element of the `oc4j-connectors.xml` file.	The `Object Class` setting defines the type of administered object (destination) to be created. Select from the drop-down list.

Table 3–9 (Cont.) JMS Connector Configuration Settings

Console Settings	XML Persistence File(s)	Description
JNDI Location This is the link-key for arrows #7 and #11 in Figure 3–1.	This setting is persisted in the `location` attribute of an `<adminobject-config>` element of the `oc4j-connectors.xml` file.	The `JNDI Location` setting specifies the JNDI location where the new resource adapter administered object (destination) is to be bound.
jndiName This is the link-reference for arrow #13 in Figure 3–1.	This setting is persisted in the `value` attribute of a `<config-property>` subelement of a `<adminobject-config>` element of the `oc4j-connectors.xml` file.	The `jndiName` setting is the JNDI name of the resource provider destination to be wrapped by the resource adapter administered object (destination) that you are creating. Note: This description of the `jndiName` setting applies to individually-bound resource provider queues and topics. See the comments in the `oc4j-connectors.xml` files in the demo set for more information.
resourceProviderName This is the link-reference for arrow #12 in Figure 3–1.	This setting is persisted in the `value` attribute of a `<config-property>` subelement of a `<adminobject-config>` element of the `oc4j-connectors.xml` file.	The `resourceProviderName` setting identifies the resource provider that owns the destination to be wrapped by the resource adapter administered object (destination) that you are creating.

For a list of the How-To documents and demo sets and their URLs, see "JMS How-To Documents and Demo Sets" on page 3-4.

Configuring the JMS Connector in the XML Files

The JMS Connector configuration settings are persisted in the following files:

- `oc4j-connectors.xml` - The `oc4j-connectors.xml` file is used to create JMS Connector instances and JMS Connector destinations.
- `oc4j-ra.xml` - The `oc4j-ra.xml` file contains OC4J-specific configuration for a JMS Connector. When you use Application Server Control Console to create or edit a JMS Connector connection factory, OC4J updates the `oc4j-ra.xml` file.
- `ra.xml` - The standard JMS Connector module configuration file provided by Oracle. When you configure a JMS Connector, entries in `ra.xml` typically serve as defaults.

For detailed examples of the `oc4j-connectors.xml`, the `oc4j-ra.xml`, and the `ra.xml` files, go to:
`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`

For detailed reference information on the JMS Connector XML files, go to Appendix A, OC4J Resource Adapter Configuration Files of the *Oracle Containers for J2EE Resource Adapter Administrator's Guide*.

Note: Remember that you must restart the OC4J instance to enable configuration changes made directly in the XML files.

Using Message-Driven Beans

OC4J supports Message-Driven Beans (MDBs), using the JMS Connector in OC4J to plug in message providers, including OEMS JMS, as well as the supported third-party message providers. This offers significant advantages such as JMS connection pooling, and MDB listener thread sets that size themselves according to changing load (dynamic load adjustment).

A full example of configuring an MDB is included in the demo at: `http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`

Download and unzip one of the how-to-gjra-with-*xxx*.zip files, where *xxx* is the name of the relevant resource provider. MDB example code can be viewed in the following files: `/src/ejb/META-INF/ejb-jar.xml` and `/src/ejb/META-INF/orion-ejb-jar.xml`

Connection Pooling

The JMS Connector uses J2CA connection pooling. For information on connection pooling, see the ***Oracle Containers for J2EE Resource Adapter Administrator's Guide***.

Dynamic Load Adjustment

Dynamic load adjustment for listener threads is based on the following `<activation-config>` properties:

- `ListenerThreadMaxIdleDuration`
- `ListenerThreadMinBusyDuration`
- `ReceiverThreads`

The `orion-ejb-jar.xml` demo file contains comments describing the `<activation-config>` properties. For information on MDB configuration, including dynamic load adjustment for MDB instances, see the ***Oracle Containers for J2EE Enterprise JavaBeans Developer's Guide***.

Using Logical Names to Reference Resources

This section describes how you can use logical names in your client application, thereby reducing the number of dependencies on installation-specific JMS configuration within the non-OC4J-specific deployment descriptors. With this indirection, you can make your client implementation generic for any JMS configuration (and therefore independent of any specific JMS resource provider).

Using logical names enables you to make your client application code resource provider-independent. In general, configure and use logical names as follows:

1. In your client application code, use logical names for JMS destinations and connection factories.
2. Declare the logical names in the J2EE application component deployment descriptors, such as `application-client.xml` and `ejb-jar.xml`.

3. Map the logical names to explicit JNDI locations in the OC4J-specific application component deployment descriptors, such as `orion-application-client.xml` and `orion-ejb-jar.xml`.

Configuring and using logical names is discussed and demonstrated in the How-To documents and the .java and .xml files available at:
`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`

Download and unzip one of the how-to-gjra-with-*xxx*.zip files, where *xxx* is the name of the resource provider.

This section describes the following topics:

- How to Declare Logical Names
- Mapping Logical Names to Explicit JNDI Locations
- JNDI Naming Property Setup for Java Application Clients
- Client Sends JMS Message Using Logical Names

The client uses JMS destinations and connection factories to send and receive messages. The recommended method for a client to retrieve a JMS destination object or connection factory is by using a logical name (an environment entry). Using direct JNDI locations should generally not be used unless logical names are not suitable, or some other mechanism is used to maintain the application's portability.

How to Declare Logical Names

To use a logical name in your application code, you must declare the logical name in one of the following XML deployment files before the application is deployed:

- A standalone Java client—in the `application-client.xml` file
- An EJB —the `ejb-jar.xml` file
- For a JSP or servlet —the `web.xml` file

This is covered in the How-To documents available at:
`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`. Under "Developing the Application Components" look at task #2 "Declare Logical Names for JMS Resources".

Declare logical names for connection factories and destinations as follows:

- Declare a logical name for a connection factory using a `<resource-ref>` element.
  - Specify the logical name for the connection factory in the `<res-ref-name>` element. This is the link-key for arrows #1 and #6 in Figure 3–1.
  - Specify the connection factory class type in the `<res-type>` element as one of the following:
    - `javax.jms.ConnectionFactory`
    - `javax.jms.QueueConnectionFactory`
    - `javax.jms.TopicConnectionFactory`
  - Specify the authentication responsibility (`Container` or `Application`) in the `<res-auth>` element.
  - Specify the sharing scope (`Shareable` or `Unshareable`) in the `<res-sharing-scope>` element.

- Declare a logical name for a JMS destination using a `<message-destination-ref` element.
  - Specify the logical name for the destination in the `<message-destination-ref-name>` element. This is the link-key for arrows #2 and #8 in Figure 3–1.
  - Specify the destination type in the `<message-destination-ref-type>` element as either `javax.jms.Queue` or `javax.jms.Topic`.
  - Specify whether the client will produce messages to this destination, consume messages from this destination, or both by setting the `<message-destination-usage>` element to `Produces`, `Consumes`, or `ConsumesProduces`.

Example

The following example illustrates how to specify logical names for a queue. In the example, `jms/PlayerConnectionFactory` is the logical name for the connection factory and `jms/PlayerCommandDestination` is the logical name for the destination queue. This example is from the `application-client.xml` file in the demo set.

```
<resource-ref>
  <res-ref-name>jms/PlayerConnectionFactory</res-ref-name>
  <res-type>javax.jms.ConnectionFactory</res-type>
  <res-auth>Application</res-auth>
</resource-ref>
<message-destination-ref>

<message-destination-ref-name>jms/PlayerCommandDest</message-destination-ref-name>
  <message-destination-type>javax.jms.Queue</message-destination-type>
  <message-destination-usage>Produces</message-destination-usage>
</message-destination-ref>
```

A full example of specifying logical names, with descriptive comments, is included in the demo at:
`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`

Drill down to the following files:
`/src/client/META-INF/application-client.xml`

and

`/src/ejb/META-INF/ejb-jar.xml`

For a list of the How-To documents and demo sets and their URLs, see "JMS How-To Documents and Demo Sets" on page 3-4.

Mapping Logical Names to Explicit JNDI Locations

After the logical names are created, the logical names must be mapped to the JNDI locations of resources. Normally, the deployer sets these up. This mapping is defined in one of the following OC4J deployment descriptor files:

- For a standalone Java client, the mapping is defined in the client's `orion-application-client.xml` file.
- For an EJB, the mapping is defined in the EJB's `orion-ejb-jar.xml` file.
- For a JSP or a servlet, the mapping is defined in the `orion-web.xml` file of the JSP or the servlet.

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

- For connection factories, the logical name (declared in a `<resource-ref>` element) is mapped to a JNDI location using a `<resource-ref-mapping>` element. This is represented by arrows #6 and #5 in Figure 3–1.
- For destinations, the logical name, declared via a `<message-destination-ref>` element, is mapped to a JNDI location using a `<message-destination-ref-mapping>` element. This is represented by arrows #8 and #7 in Figure 3–1.

See the following sections for how the mapping occurs for the three OEMS JMS quality of service options and how application components use this naming convention:

- Resource Naming for OEMS JMS In-Memory and File-Based
- Resource Naming for OEMS JMS Database
- JNDI Naming Property Setup for Java Application Clients
- Client Sends JMS Message Using Logical Names

JNDI Naming Property Setup for Java Application Clients

In an OC4J standalone environment, a Java application client accesses an OEMS JMS destination object by defining the following properties in the `jndi.properties` file:

```
java.naming.factory.initial=
   com.evermind.server.ApplicationClientInitialContextFactory
java.naming.provider.url=ormi://myhost/myApplicationDeploymentName
java.naming.security.principal=oc4jadmin
java.naming.security.credentials=welcome
```

If you wish to specify the port, use the optional `:port` element as follows:

```
java.naming.provider.url=ormi://myhost:port/myApplicationDeploymentName
```

The default RMI port is `23791`.

You must:

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

In the Oracle Application Server, a Java application client accesses an OEMS JMS destination object by defining the following properties in the `jndi.properties` file:

```
java.naming.factory.initial=
   com.evermind.server.ApplicationClientInitialContextFactory
java.naming.provider.url=opmn:ormi://$HOST:$OPMN_REQUEST_PORT:$OC4J_INSTANCE/myApp
licationDeploymentName
java.naming.security.principal=oc4jadmin
java.naming.security.credentials=welcome
```

- Use the `ApplicationClientInitialContextFactory` as your initial context factory object.

- Supply the OPMN host and port, the OC4J instance, and the application deployment name in the provider URL.

You can see a full example of a `jndi.properties` file at:
`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`

Drill down to the following file:

```
/src/client/jndi.properties
```

Client Sends JMS Message Using Logical Names

After the resources have been defined and the JNDI properties configured, the client must perform the following steps to send a JMS message. These steps summarize the procedure shown in "Sending and Receiving JMS Messages" on page 3-26.

1. Retrieve both the configured JMS destination and its connection factory using a JNDI lookup.
2. Create a connection from the connection factory.
3. Create a session over the connection.
4. Providing the retrieved JMS destination, create a message producer.
5. Create the message.
6. Send the message using the message producer.
7. Close the connection.

Required Class Path for Application Clients Using JMS Connector Lookup

When using the JMS Connector from an application client, the JAR files that must be included in the class path are listed inTable 3–10, " Client-side JAR Files Required for JMS Connector Lookup".

Table 3–10 Client-side JAR Files Required for JMS Connector Lookup

JAR	ORACLE_HOME Path
`oc4jclient.jar`	`/j2ee/<instance>`
`jta.jar`	`/j2ee/<instance>/lib`
`jms.jar`	`/j2ee/<instance>/lib`
`jndi.jar`	`/j2ee/<instance>/lib`
`javax77.jar`	`/j2ee/<instance>/lib`
`adminclient.jar`	`/j2ee/<instance>/lib`
`oc4j-internal.jar`	`/j2ee/<instance>/lib`
`connector.jar`	`/j2ee/<instance>/lib`
`jmxri.jar`	`/j2ee/<instance>/lib`
`jazncore.jar`	`/j2ee/<instance>`
`oc4j.jar`	`/j2ee/<instance>`

Using High Availability and Clustering for OEMS JMS

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

For information about high availability, see the *Oracle Application Server High Availability Guide*.

Table 3–11, " High Availability Summary" summarizes the support for high availability in OEMS JMS.

Table 3–11 High Availability Summary

Feature	Database	In-Memory and File-Based
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 or different machine within Oracle Application Server Cold Failover Cluster (Midtier)

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

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

The following sections discuss how to achieve high availability and clustering with OEMS JMS:

- Configuring OEMS JMS In-Memory and File-Based High Availability
- Configuring OEMS JMS Database High Availability
- Failover Scenarios When Using a RAC Database
- Sample Code for Connection Recovery
- Clustering Best Practices

Configuring OEMS JMS In-Memory and File-Based High Availability

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

This section describes the following OEMS JMS clustering topics:

- Terminology
- Distributed Destinations

In this configuration, all factories and destinations are defined on all OC4J instances. Each OC4J instance has a separate copy of each destination.

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

- Cold Failover Cluster

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

- 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 JMS applications, especially those where message ordering is a requirement.

- Custom Topologies

  This section discusses reasons for using custom topologies, lists requirements and impacts of various topology choices, and explains how custom topologies can be created.

  Custom topologies by their very nature are more complex to understand and use correctly, and involve more configuration effort. They should only be used when none of the standard configurations is adequate.

Terminology

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

- *OHS*—Oracle HTTP Server
- *OracleAS Cluster*—A grouping of similarly configured Oracle Application Server instances
- *Oracle Application Server Instance*—Represents an installation of Oracle Application Server (that is, an *`ORACLE_HOME`*)
- *OC4J Instance*—Within an Oracle Application Server instance there can be multiple OC4J instances.
- *Factory*—Denotes a JMS connection factory
- *Destination* —Denotes a JMS destination

Distributed Destinations

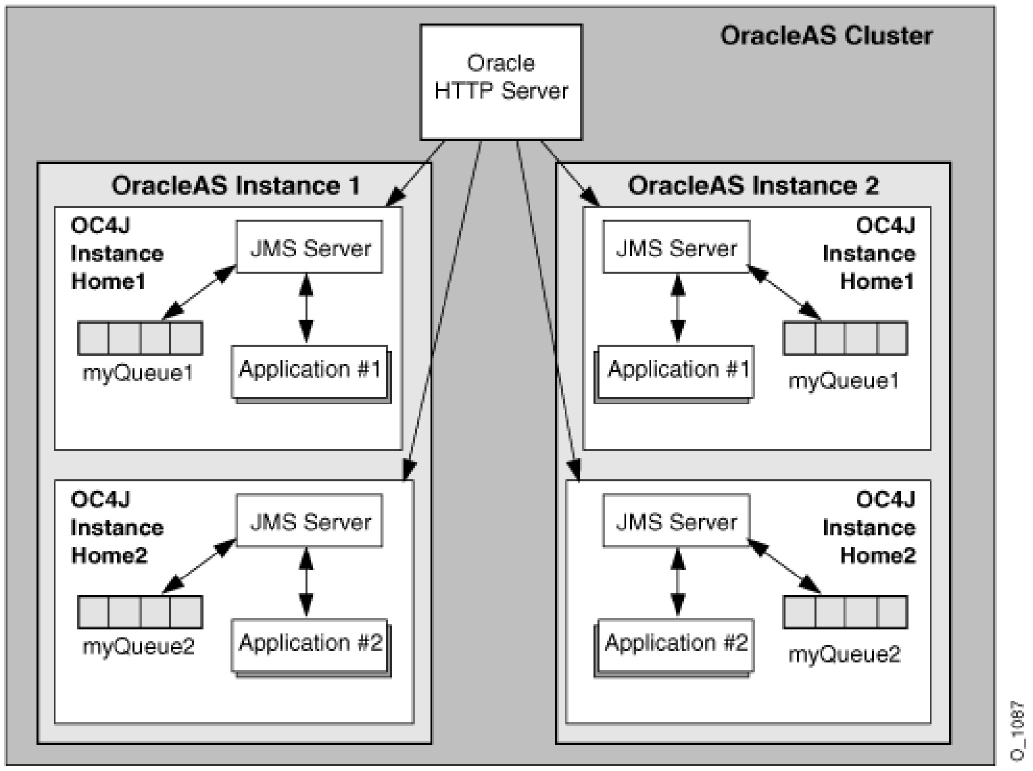
In this configuration, OHS services HTTP requests and load balances them across two or more Oracle Application Server instances in an Oracle Application Server cluster.

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 type of deployment has several advantages:

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

Figure 3–2 Distributed Destinations



Configuring a Distributed Destinations Cluster

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

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

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

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

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

Cold Failover Cluster

This configuration is a two-node cluster. Only one node is active at any time. The second node is made active if the first node fails. For larger clusters, Cold Failover Clustering can be used in combination with the dedicated JMS server configuration described in the next section by having two nodes configured to be the dedicated JMS server, but having only one of them active at any given time. For Cold Failover documentation, see the *Oracle Application Server High Availability Guide*.

Configuring a Cold Failover Cluster

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

```
<jms-server host=vmt.my.company.com port="9127">
….
….
</jms-server>
```

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

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

Stop and Start

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

```
$ORACLE_HOME/opmn/bin/opmnctl stopall
$ORACLE_HOME/opmn/bin/opmnctl startall
```

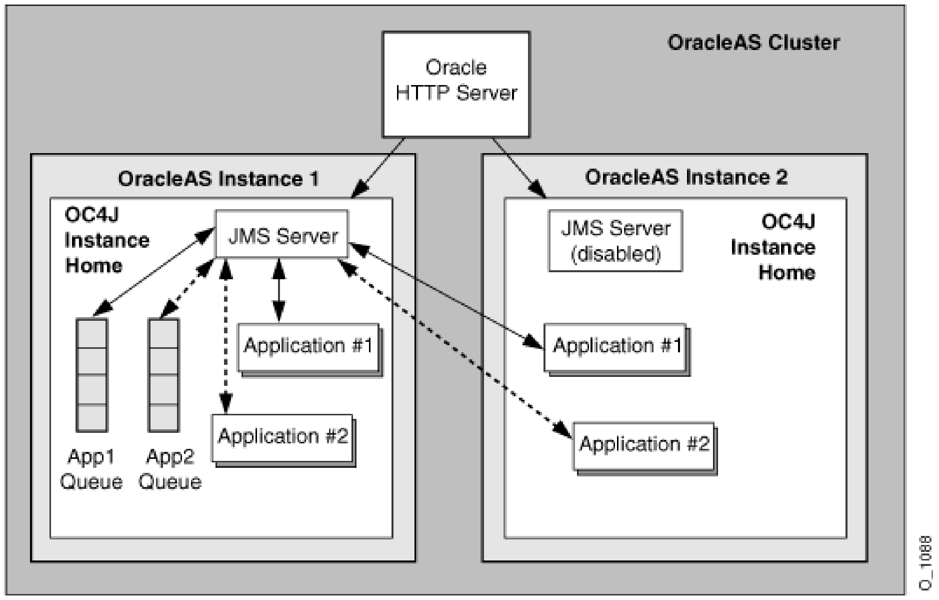
Dedicated JMS Server

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

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

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

Figure 3–3 Dedicated JMS Server



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

1. Make sure that only one JVM is started for this OC4J instance (`JMSserver`).

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

2. Specify only one value for the JMS port range for this instance.

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

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

Modifying the OPMN Configuration

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

- Assuming an OC4J instance has been created through Application Server Control Console called `JMSserver`, then the line denoted by (1) demonstrates where to locate the start of the `JMSserver` definition.
- The line denoted by (2) is the JMS port range that OPMN uses when assigning JMS ports to OC4J JVMs. For the desired dedicated OC4J instance that acts as your JMS provider, narrow this range down to one value. In this example, the original range was from `3701-3800`. In our connection factory definitions, we know the port to use by configuring this value as `3701-3701`.
- The line denoted by (3) defines the number of JVMs that will be in the `JMSserver` default group. 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_group" numprocs="1"/>
  </process-type>
</ias-component>
```

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

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

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

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

Queue Connection Factory Definition Example

Here is an example for defining a queue connection factory in the `jms.xml` file of the dedicated server:

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

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

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

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

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

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

Custom Topologies

In addition to the previously discussed configurations, OEMS JMS can be configured into arbitrary user-defined topologies.

Mechanisms

This section discusses the mechanisms that allow custom topologies to be created and to function. These mechanisms are already defined in other places in this document, however this section also covers some of the finer points that need to be understood when using a custom topology.

Using a JMS server other than the one in the local OC4J JVM requires a connection factory that references the desired JMS server. This is done by way of a host and port mapping.

First, the JMS server must be associated with a host and port. Usually, the host is the single host/IP address of the machine on which the JMS server is run. In the case of a machine with multiple host/IP addresses, all addresses will be associated with the JMS server unless a specific address is specified via the "host" attribute of the `<jms-server>` element in the `jms.xml` file. The `port` is `9127` unless a different value is specified via the "`port`" attribute. See the `<jms-server>` entry in Table 3–2, " Configuration Elements" on page 3-21, and also see "Configuring Ports" on page 3-26. When OPMN is used, the "`port`" attribute is ignored and port assignment is handled automatically from a range of port values set in the `opmn.xml` file. See "Modifying the OPMN Configuration" on page 3-67.

Once the host and port have been designated for each JMS server, then connection factories can be made to reference specific JMS servers. The first step in making a connection factory reference a specific JMS server is to set the `host` attribute of the connection factory element to be the same as the host/IP address of the given JMS server. See `<connection-factory>`, `<xa-connection-factory>`, and the other connection factory elements in Table 3–2, " Configuration Elements" on page 3-21. When making a connection factory reference a JMS server on the same machine where that machine has a single host/IP address, this step may be skipped (the "`host`" attribute may be left out). The next step in making a connection factory reference a specific JMS server is to set the "`port`" attribute of the connection factory element to be the same as the port of the given JMS server. If the port is `9127`, this step may be skipped (the "`port`" attribute may be left out).

If connection factory CF1 has been configured (as described in the previous paragraph) to reference JMS server JMS1, then a connection created from CF1 is a connection to JMS1. All operations performed via that connection will go to JMS1. It is not possible to use that connection to interact with any other JMS server. (It is not even possible to use that connection to interact with the JMS server running in the local JVM, unless it happens to be JMS1.) Likewise, if that connection is used to create a session, and that session to create a message producer, then that session and message producer are both connected to JMS1. All messages sent via that message producer will go to JMS1, will be stored (in memory or file) by JMS1, and can only be received or browsed from JMS1 (via a message consumer or queue browser that is connected to JMS1). Note that this JMS server association does not extend to `javax.jms.Message` objects. Message objects created or received from a session or message producer associated with one JMS server may be sent using any message producer. For example, an application could receive a message from a physical destination maintained by JMS server "JMS1" using a message consumer associated with "JMS1", and then send that message to a physical destination maintained by JMS server "JMS2" using a message producer associated with "JMS2".

When dealing with multiple JMS servers, it is necessary to draw a distinction between physical destinations (physical locations where messages are stored), destination configuration (elements/attributes in `jms.xml`) and destination admin objects (Java objects looked up in JNDI).

From the perspective of a JMS server:

- A name (whether from a destination configuration or from a destination admin object) uniquely identifies a physical destination - one controlled solely by the given JMS server.
- The `persistence-file` attribute (or its absence) tells the JMS server where it should persist the messages for a given physical destination.
- The location attribute tells the JMS server where to bind a destination admin object (containing the value of the `name` attribute) in JNDI in the local JVM.

From the perspective of a JMS client:

- Destination admin objects do not contain JNDI locations. Instead, they are looked-up at a JNDI location.
- Destination admin objects do not have any persistence file settings. Instead, the persistence file (if any) used for a given physical destination is determined by the JMS server that controls that physical destination.
- Destination admin objects contain only a name, and names do not uniquely identify a physical destination. When interacting with JMS server JMS1, a destination admin object will be considered to be a reference to a physical destination controlled by JMS1. When interacting with JMS server JMS2, that same destination admin object will be considered to be a reference to a different physical destination - one controlled by JMS2. (If, when interacting with a JMS server, a destination admin object is used whose name does not match the name of any physical destination on the given JMS server, the operation will fail.)

For example, given an excerpt from JMS server JMS1's `jms.xml` file:

```
<queue name="queue1"
       location="jms/alpha"
       persistence-file="foo">
</queue>
```

And given an excerpt from JMS server JMS2's `jms.xml` file:

```
<queue name="queue1"
       location="jms/omega"
       persistence-file="bar">
</queue>

<queue name="queue2"
       location="jms/alpha">
</queue>
```

In Table 3–12, " Results for Various Message Scenarios" on page 3-70, if a destination is looked-up from the server in column 1 using the JNDI location from column 2, and then a message producer connected to the server in column 3 is used to send a PERSISTENT message, the result is as described in the final column.

Table 3–12 Results for Various Message Scenarios

Destination looked-up in the same JVM as	JNDI location used to look up destination	Message producer connected to	Result (physical destination that receives message, or error)
JMS1	jms/alpha	JMS1	JMS1's "queue1", persisted to file "foo"
JMS1	jms/alpha	JMS2	JMS2's "queue1", persisted to file "bar"
JMS2	jms/alpha	JMS2	JMS2's "queue2", stored in memory on JMS2
JMS2	jms/alpha	JMS1	Error (JMS1 does not have a "queue2")
JMS1	jms/omega	don't care	Error looking up "jms/omega" in JNDI
JMS2	jms/omega	JMS1	JMS1's "queue1", persisted to file "foo"
JMS2	jms/omega	JMS2	JMS2's "queue1", persisted to file "bar"

Note that the above example (and the connection factories in the "Distributed Destinations" configuration) bind/require different values for the same JNDI location in different JVMs, and therefore require that JNDI values not be automatically propagated from one JVM to another.

When using the following combination:

- OPMN (automatic JMS port assignment)
- One or more dedicated JMS servers
- Multiple OC4J instances per machine

Care should be taken to ensure that:

- The range of available JMS ports is just enough for the number of JMS servers on the given machine. This guarantees that the dedicated JMS server port number actually gets assigned to one of the available JMS servers.
- The persistence files (if any) for the destination(s) controlled by the dedicated JMS serve are specified in the `jms.xml` file(s) using absolute paths. If a path is relative to the OC4J instance, then previously persisted messages would be lost after a server restart when OPMN assigns the dedicated JMS server port number to a JMS server in a different OC4J instance than it did after the previous restart.

Considerations

The "Dedicated JMS Server" and "Distributed Destinations" configurations are the only configurations where each instance of an arbitrary JMS application is guaranteed to only ever communicate with a single JMS server. The considerations discussed in this section do not apply to those scenarios. They also do not apply to other scenarios where it can be guaranteed that a JMS application instance only ever communicates with a single JMS server.

For example, if all instances of application A use JMS server #1 and all instances of application B use JMS server #2, then the following considerations are not applicable.

In other scenarios, where a single JMS application instance interacts with two or more JMS servers, there are several consequences:

- There is an application-complexity impact:

  Since a JMS server is selected based on the connection factory (and its derived objects), using multiple JMS servers from a single application instance requires the use of multiple connection factories (and derived objects such as sessions, consumers and producers). For example, if an application uses producer A to enqueue a message to JMS server #1, it is not possible to also use producer A to enqueue a message to JMS server #2. Instead, a different producer, derived from a different connection factory than the one from which producer A was derived, must be used. (To be specific, a producer derived from a connection factory associated with JMS server #2 must be used.) This also means that using multiple JMS servers may not be possible with a pre-existing JMS application unless the application already accommodates (or can be modified to accommodate) this prerequisite.

- There is a performance impact:

  Since two different JMS servers represent two different resource managers, global transactions involving two JMS servers will always require two-phase commit, even if no other resource types (such as JDBC) are used during the transaction. Even if two-phase commit was already required for a given transaction, when more resource managers participate in a transaction, the cost (in performance) of the transaction goes up. However, using multiple JMS servers is still a performance win in many scenarios since it can be used to offload work from potential bottlenecks (such as a single, dedicated JMS server) and increase both parallelism and locality.

These same considerations also apply if a single JMS application instance is using two different JMS resource providers. For example, an application might use OEMS JMS In-Memory for memory-based or file-based persistence and also use OEMS JMS Database for database-backed persistence.

Cases

This section shows two example situations driven by application-specific messaging requirements, along with custom topologies that could be used to improve throughput in each case. These examples should not be considered exhaustive, either in terms of situations you may encounter or in topologies that can be created.

Case #1: Only some destinations need global consistency:
It may be that some destinations must provide global consistency, as provided by the "Dedicated JMS Server" configuration, and other destinations have no such requirement. In that case, it is not necessary to pay the expense of global consistency (routing all messages through a single JMS server) for the destinations that do not require it and could instead be locally partitioned, such as in the "Distributed Destinations" configuration.

Say that destinations A and B need global consistency, and that destinations C and D do not. In that case, all jms.xml files define all four destinations. One machine is selected to host the dedicated JMS server for destinations A and B. The dedicated JMS server must have fixed host and port values in order to allow connection factories to reference it. At least two connection factories are defined in the jms.xml files, one that references the dedicated JMS server (that is, with a fixed host and port value) and one that references the JMS server in the local JVM (that is, with the default host and port values). In order to access destinations A and B (and maintain global consistency), the Java code must use a connection factory that references the dedicated JMS server. In order to access destinations C and D, the Java code must use a connection factory that references the JMS server in the local JVM.

Case #2: Using multiple dedicated JMS servers for load-balancing:
In the previous example, two destinations (A and B) needed global consistency. This requires that destination A has a dedicated JMS server and that destination B has a dedicated JMS server, but does not require that those two JMS servers be the same. When multiple destinations requiring global consistency are heavily used, it may be worth while to divide the load of those destinations across multiple dedicated JMS servers. This is especially true for destinations that tend to not be involved in a common transaction and/or when a multi-destination dedicated JMS server is acting as a system bottleneck.

For this case, all jms.xml files define all four destinations. One machine is selected to host the dedicated JMS server for destination A. Another machine (or another JVM on the same machine) is selected to host the dedicated JMS server for destination B. Both dedicated JMS servers must have fixed host and port values in order to allow connection factories to reference them. At least three connection factories are defined in the jms.xml files, one that references the dedicated JMS server for destination A, one that references the dedicated JMS server for destination B, and one that references the JMS server in the local JVM. In order to access destination A, the Java code must use a connection factory that references the dedicated JMS server for destination A. In order to access destination B, the Java code must use a connection factory that references the dedicated JMS server for destination B. In order to access destinations C and D, the Java code must use a connection factory that references the JMS server in the local JVM.

Configuring OEMS JMS Database High Availability

To enable high availability the OEMS JMS Database option, run the following:

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

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

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

If network failover of a back-end database occurs, where the database is a non-RAC database, then state information in the server is lost (that is, state of transactions not yet committed). Additionally, the JMS objects (connection factories, destination objects, connections, sessions, and so on) inside the application may also become invalid.

After the failure of the database occurs, those JMS objects will throw exceptions whenever the application code attempts to use them. In order to recover, the application must recreate any and all JMS objects which are throwing such exceptions. This includes relooking up connection factories using JNDI.

Failover Scenarios When Using a RAC Database

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

- RAC Network Failover
- Transparent Application Failover (TAF)

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

RAC Network Failover

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

The following example outlines the logic:

```
getMessage(QueueSesssion session) {
    try {
        Message msgRec = null;
        QueueReceiver rcvr = session.createReceiver(rcvrQueue);
        msgRec = rcvr.receive();
    } catch(Exception exc) {
        while (exc instanceof JMSException) {
           exc = ((JMSException) exc).getLinkedException();
        }
        if (exc instanceof SQLException) {
            sql_ex = (SQLException) exc;
            db_conn = (oracle.jms.AQjmsSession) session.getDBConnection();
            if ((DbUtil.oracleFatalError(sql_ex, db_conn)) {
                // failover logic
            }
        }
    }
}
```

Transparent Application Failover (TAF)

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

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

- As described in "RAC Network Failover" on page 3-73, you can use the `DbUtil.oracleFatalError` method to determine if the error is a fatal error. If it is not a fatal error, then the client recovers by sleeping for a short time and then retrying the current operation.
- You can recover from failback and transient errors caused by incomplete failover by trying to use the JMS connection after a short time. Waiting allows the database failover to recover from the failure and reinstate itself.
- In the case of transaction exceptions (such as "Transaction must roll back" (ORA-25402) or "Transaction status unknown" (ORA-25405)) you must roll back the current operation and retry all operations past the last commit. The connection is not usable until the cause of the exception is dealt with. If this retry fails, then close and re-create all connections and retry all non committed operations.

Sample Code for Connection Recovery

The following example shows OEMS JMS application code that is can be used with any of the persistence options, for a queue that is tolerant to intermittent connection failures, such as may happen during network outages, server reboots, JMS server failover, and other situations.

```
while (!shutdown) {
       Context ctx = new InitialContext();

       // get the queue connection factory
       QueueConnectionFactory qcf =
                      (QueueConnectionFactory) ctx.lookup(QCF_NAME);

       // get the queue
       Queue q = (Queue) ctx.lookup(Q_NAME);

       ctx.close();

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

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

           // receive requests using the queue receiver and send
           // replies using the queue sender
           while (!done) {
               Message request = rcv.receive();
               Message reply = qs.createMessage();

               // put code here to process request and construct reply

               snd.send(reply);
               qs.commit();
           }
       } catch (JMSException ex) {
           if (transientServerFailure) {
               // retry
           } else {
               shutdown = true;
           }
       } finally {
          // close the queue connection
          if (qc != null) qc.close();
       }
   }
```

J2CA Configuration for Connection Recovery

Note that connection pooling means that when a connection is closed by the application, the physical connection is not actually closed by the connector but is instead returned to the connection pool. If a connection fails, this can result in an invalid connection being returned to the pool. Subsequent attempts to create a new connection may then obtain an invalid connection, negating the value of the above style of code. In order to make failover reliable, connection pooling should be disabled for the connection factory used to create the connection. This is done by modifying the the connection factory's `<connector-factory>` element in the `oc4j-ra.xml` file to include the following:

```
<connection-pooling use="none">
</connection-pooling>
```

Clustering Best Practices

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

JMS Router

This section describes the JMS Router.

Functionality

In the present complex and disparate enterprise-computing environment, a messaging infrastructure must integrate and interoperate with various messaging systems to achieve maximum connectivity.

The JMS Router provides a reliable asynchronous JMS message routing service that can bridge:

- Destinations of the same JMS provider.
- Destinations of different JMS providers.

The JMS Router guarantees the following behavior:

- For persistent JMS messages, the JMS Router guarantees exactly-once message delivery.
- For non-persistent JMS messages, the JMS Router guarantees at-most-once message delivery.
- The JMS Router guarantees the delivery of messages in the order they are received by the JMS Router from the source destination.

Users can specify a message selector to selectively route messages when creating a router job. Only the messages that satisfy the selector are routed from the source destination to the target destination by the JMS Router. The message selectors must be valid JMS message selectors. See the JMS 1.1 specification for the message selector syntax and semantics for JMS queues and topics.

The JMS Router requires that the participating JMS provider support JMS 1.1 if either the source or target of a JMS Router job is a JMS topic.

You can access the JMS 1.1 specification at:
`http://java.sun.com/products/jms/docs.html`.

By bridging JMS destinations, the JMS Router provides users the following benefits:

- The application that sends messages in one messaging system and the application that receives messages in another messaging system are decoupled. It is not necessary for the two applications to be aware of messages traveling across different messaging systems.
- An application that needs to send and receive messages to or from multiple messaging systems need only connect to one messaging system and rely on the JMS Router to distribute messages across multiple messaging systems. This simplifies application code by eliminating the need to connect to multiple messaging systems, to manage global transactions with 2-PC, and to deal with message translation.
- The automatic recovery and guaranteed message delivery by the JMS Router make applications and application integration more reliable because the applications do not need to require that all involved messaging systems be alive at the same time.

JMS Providers

The JMS Router uses the JMS Connector to access the following JMS providers:

- OEMS JMS In-Memory, File-Based, and Database persistence options
- IBM WebSphere MQ V6.0 JMS, MQ V5.3 JMS with Fix Pack 8 (CSD08)
- TIBCO Enterprise for JMS version 3.1.0
- SonicMQ 6.0 JMS

The JMS Router relies on the OC4J pre-packaged standalone JMS Connector instance OracleASjms to access the OEMS JMS In-Memory and File-Based options that run in the same container as the JMS Router. Therefore, there is no need to deploy any additional adapters for routing messages to and from OEMS JMS In-Memory or File-Based Destinations.

For routing messages to and from any JMS provider other than OracleAS JMS, there must be a standalone JMS Connector deployed for the JMS provider.

The JMS Router uses the J2CA adapter's declarative container-managed sign-on mechanism for JMS provider authentication and authorization. The JMS Router runs on behalf of the role `jmsRouter`. Therefore, all the resource adapters used by the JMS Router must have a valid principal mapping with either the `<default-mapping>` element or a `<principal-mapping-entry>` element that has `jmsRouter` as `<initiating-user>` in the `oc4j-ra.xml`.

Additional information on how to use the JMS Router for OEMS JMS is available at `http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`.

Configuration

This section discusses configuration with respect to the following JMS Router objects:

- Router Jobs
- Global Router Parameter(s)
- Subscription
- Log Queues and Exception Queues

The primary administration interface of the JMS Router is the JMS Router MBean. The MBean enables you to manage configuration, to start and stop individual JMS Router jobs, and to monitor the status of the jobs.

The JMS Router may also be configured statically through the `jms.xml` file.

Router Jobs

The JMS Router job is defined as a message routing task to move messages from the source destination to a target destination. When processing a router job, the JMS Router dequeues messages from the source destination and enqueues the messages to the target destination.

The source and the target destinations can be either JMS queues or topics. If the source is a JMS queue, then the JMS Router moves all messages from the queue. If the source is a topic, then the JMS Router moves all messages that are published to the topic since the JMS durable subscriber was created.

Note: JMS Router jobs must not share a source, whether it is a queue or a durable subscriber.

A JMS Router job uses the objects described in Table 3–14, " JMS Router Settings" on page 3-81.

The JMS Router uses the JMS Connector to access JMS destinations and connection factories. The JMS Router requires that source and target destinations for all JMS Router jobs exist in the accessed JMS provider, and are configured in the appropriately configured JMS Connector. For information on configuring Queues, Topics, and ConnectionFactories, see "Configuring Destination Objects and Connection Factories" on page 3-19. For additional resource adapter configuration information, see "JMS Connector" on page 3-51 and the *Oracle Containers for J2EE Resource Adapter Administrator's Guide*.

Global Router Parameter(s)

The number of concurrent jobs is configurable as a JMS Router attribute `maxLocalConcurrency`. This attribute prevents a JMS Router with many active jobs from dominating the OC4J J2EE container. This also sets the upper limit on the number of JMS sessions created by JMS Connectors that are used by the JMS Router.

Subscription

If the source is a JMS topic, the user may provide a durable subscriber name on the topic. The actual durable subscriber can be created before the associated router job is created. If the name of the durable subscriber is not specified, then the JMS Router creates a subscriber name based on the name of the job. If the durable subscriber associated with the subscriber name does not exist when the JMS Router starts to process the job, then the router creates the durable subscriber. The JMS Router moves all messages that are published to the topic since the durable subscriber was created.

By default, when a JMS Router job is deleted, the JMS Router attempts to remove the durable subscriber. If the JMS Router fails to remove the durable subscriber, then the user must remove it through the administration interface of the messaging system to avoid unnecessary message accumulation. Users can also optionally ask the JMS Router not to remove the durable subscriber when deleting a JMS Router job.

Log Queues and Exception Queues

This section discusses log queues and exception queues.

Log Queues

The JMS Router uses queues for internal logging. Each router job must have a source log queue and a target log queue. These queues are used to ensure quality of service when processing router jobs. For the OEMS JMS In-Memory and File-Based options, the JMS Router creates log queues. For the OEMS JMS Database option and other supported non-Oracle JMS providers, a JMS queue must be created on that system, and the name of the queue must be specified when creating the router job. Log queues may be shared by multiple jobs.

Log queues for the JMS Router must exist in the accessed messaging system and must be configured in the appropriate resource adapter.

Exception Queues

When the JMS Router fails to send a message to the target destination, it blocks processing of the associated router job in order to maintain message order.

In order for the JMS Router to keep processing a job with problematic messages, users can configure the job with an exception queue into which the JMS Router can move the problematic messages in order to process the remaining messages in the source destination.

Each JMS Router job can have one exception queue, which must be a JMS queue in the same messaging system as the source destination and accessible through the same connection factory as that of the source destination. The physical JMS destination for the exception queue must exist before it is used for any JMS Router jobs. In the OEMS JMS In-Memory and File-Based options, an exception queue is already defined by default -- `Oc4jJmsExceptionQueue`.

In order for the JMS Router to move problematic messages into the exception queue, the `useExceptionQueue` flag must be set to `true` for the associated router job.

Optionally, exception queues, if configured, for the JMS Router must exist in the accessed messaging system and must be configured in the appropriate resource adapter.

Configuring the JMS Router and Its Objects

The primary administration interface of the JMS Router is the JMS Router MBean. Use the MBean to manage configuration, to start and stop individual JMS Router jobs, and to monitor the status of the jobs.

This section describes the operations and settings available in JMS Router MBean.

Path to the JMS Router MBean:

OC4J:Home > Administration tab > Services > JMS Providers > Click the icon. > Select the appropriate tab. > Related Links: OracleASJMSRouter

Note: The JMS Router exports a single JMS Router MBean per OC4J instance: "`jmsrouter:j2eeType=OracleASJMSRouter`".

The following table lists and describes the operations available on the JMS Router MBean. All operations update the JMS Router dynamically and persist the changes to `jms.xml`.

Table 3–13 JMS Router MBean Operations

Operation	Description
`addRouterJob`	Configures a JMS Router job.
`alterRouterJob`	Alters the attributes of a JMS Router job. The default value for all parameters to alterRouterJob is to leave them unchanged.
`configureRouter`	Configures certain global parameters of the JMS Router, such as `maxLocalConcurrency`.
`pauseRouterJob`	Suspend the execution of a JMS Router job.
`removeRouterJob`	Removes a JMS Router job.
`resetRouterJob`	Resets the number of failures in the JMS Router job to zero.
`resumeRouterJob`	Resume the execution of a JMS Router job.

Table 3–14, " JMS Router Settings" lists and describes the router and router job settings in the JMS Router MBean. The first column of the table lists the name of the setting in the JMS Router MBean. The settings that you make in the JMS Router MBean are persisted to the `jms.xml` file. The second column of the table lists the name of the corresponding elements in the `jms.xml` file.

Table 3–14 JMS Router Settings

MBean Setting	XML Entity	Description
`jobName` Editable through the `addRouterJob` operation.	`<job-name>` Router job element	The name given to this router job. It must be unique for the OC4J instance for which it is configured.

`messageSource`
Editable through the `addRouterJob` operation.
	`<message-source>`
	Router job element
		The JNDI location of the destination (topic or queue) to be used as the source of messages.
		If automatic destination wrapping is used, then the name may be of the form:
		`<JNDIsubcontext>/providerName`
		where `<JNDIsubcontext>` is the automatic destination wrapping JNDI subcontext as specified in `oc4j-connectors.xml`.
		For example, if automatic destination wrapping is used with the JMS Connector, a name would be of the form:
		`OracleASjms/Queues/jms/queue_name`
		or
		`OracleASjms/Topics/jms/topic_name`

`sourceConnectionFactory` Editable through the `addRouterJob` operation.	`<source-connection-factory>` Router job element	The JNDI location of the connection factory used to access the message source. For example, using the JMS Connector, a name might be `OracleASjms/MyCF` If the message source is a JMS Topic, this name must refer to a connection factory that supports both JMS Queues and Topics.

`messageTarget`
Editable through the `addRouterJob` operation.
	`<message-target>`
	Router job element
		The JNDI location of the destination (topic or queue) to be used as the target for message propagation.
		If automatic destination wrapping is used, then the name may be of the form:
		`<JNDIsubcontext>/providerName`
		where `<JNDIsubcontext>` is the automatic destination wrapping JNDI subcontext as specified in `oc4j-connectors.xml`.
		For example, if automatic destination wrapping is used with the JMS Connector, a name would be of the form:
		`OracleASjms/Queues/jms/queue_name`
		or
		`OracleASjms/Topics/jms/topic_name`

Table 3–14 (Cont.) JMS Router Settings

MBean Setting	XML Entity	Description
`targetConnectionFactory` Editable through the `addRouterJob` operation.	`<target-connection-factory>` Router job element	The JNDI location of the connection factory used to access `messageTarget`. For example, using the JMS Connector, a name might be `OracleASjms/MyCF` If the message source is a JMS Topic, this name must refer to a connection factory that supports both JMS Queues and Topics.
`sourceLogQueue` Editable through the `addRouterJob` operation.	`<source-log-queue>` Router job element	The JNDI location of the queue to be used for JMS Router internal logging for the source messaging system. The log queue must be accessible through the connection factory indicated by `sourceConnectionFactory`. This parameter is optional when using OEMS JMS In-Memory or File-Based Destinations. In this case, if not specified, then the JMS Router uses the queue `OracleASRouter_LOGQ` If this queue does not exist, then the JMS Router will create it.
`targetLogQueue` Editable through the `addRouterJob` operation.	`<target-log-queue>` Router job element	The JNDI location of the queue to be used for JMS Router internal logging for the target messaging system. The log queue must be accessible through the connection factory indicated by `targetConnectionFactory`. This parameter is optional when using OEMS JMS In-Memory or File-Based Destinations. In this case, if not specified, then the JMS Router uses the queue `OracleASRouter_LOGQ` If this queue does not exist, then the JMS Router will create it.
`messageSelector` Editable through the `addRouterJob` operation.	`<message-selector>` Router job element	Optional. A message selector for selectively receiving messages from the `messageSource`. The default is `none`.
`subscriberName` Editable through the `addRouterJob` operation.	`<subscriber-name>` Router job element	Optional. The name for a durable subscriber to use if the `messageSource` is a topic. If the specified durable subscriber does not exist and the `messageSource` is a topic, then the JMS Router attempts to create it The default value is `OracleASRouter_`*`jobName`* where *jobName* is the name of the Router job.

Table 3–14 (Cont.) JMS Router Settings

MBean Setting	XML Entity	Description
`exceptionQueue` Editable through the `addRouterJob` and `alterRouterJob` operations.	`<exception-queue>` Router job element	The JNDI location of a queue into which undeliverable messages are placed. The exception queue must be accessible from `sourceConnectionFactory`. This parameter need only be specified if useExceptionQueue is true. When useExceptionQueue is `true`, this parameter is optional when the message provider is OEMS JMS In-Memory or File-Based option. If not specified, then the JMS Router will use the OEMS JMS exception queue, `Oc4jJmsExceptionQueue`. This queue already exists, so does not need to be created separately. The default is `null` when using `alterRouterJob`.
`maxRetries` Editable through the `addRouterJob` and `alterRouterJob` operations.	`max-retries` Router job attribute	Optional. The number of times the JMS Router will attempt to deliver a message for this job before suspending execution of the job. The value must be an integer-valued string. If the string does not represent an integer, then it is ignored and the default is used. The default is `16` when using `addRouterJob`. The default is `null` when using `alterRouterJob`.
`pollingInterval` Editable through the `addRouterJob` and `alterRouterJob` operations.	`polling-interval` Router job attribute	Optional. If no message is present in the message source, then this parameter represents the minimum time in seconds to wait before checking the message source again. The value must be a string representing an integer. If the string does not represent an integer, then it is ignored and the default is used. The default is `5` when using `addRouterJob`. The default is `null` when using `alterRouterJob`.
`useExceptionQueue` Editable through the `addRouterJob` and `alterRouterJob` operations.	`use-exception-queue` Router job attribute	Optional. If the value is set to `true` and an exception queue is available, then undeliverable messages will be placed in it. Otherwise, no exception queue will be used. The default is `false` when using `addRouterJob`. The default is `null` when using `alterRouterJob`.
`pauseJob` Editable through the `addRouterJob` operation.	`pause-job` Router job attribute	Optional. If `true`, then the job is added in deactivated mode. To start the job, invoke `resumeJob`. If not `true`, then the job is created in activated mode. The default is `false`.
`batchSize` Editable through the `addRouterJob` and `alterRouterJob` operations.	`batch-size` Router job attribute	Optional. The number of messages to dequeue and enqueue in a single transaction. The default is `30` when using `addRouterJob`. The default is `null` when using `alterRouterJob`.

Table 3–14 (Cont.) JMS Router Settings

MBean Setting	XML Entity	Description
`removeSubscriber` Editable through the `removeRouterJob` operation.	`remove-subscriber` Router job attribute	If `true` and the router job used a durable subscriber, then the JMS Router attempts to remove that durable subscriber. If `false`, then no attempt is made to remove the durable subscriber. If the durable subscriber is not removed successfully by the JMS Router, then the user is responsible for its removal. The default is `true`.
`maxLocalConcurrency` Editable through the `configureRouter` operation.	`maxlocalconcurrency` Global JMS Router attribute	Optional. The maximum concurrency of dequeue operations possible. This argument places a limit on the number of threads that the JMS Router can use to dequeue messages at one time. A negative number indicates the JMS Router will enforce no limits on concurrency. The default behavior is to enforce no limits.

JMS Router Configuration in jms.xml

The file `J2EE_HOME/config/jms.xml` is used to persist JMS Router jobs and global configuration.

In the `jms.xml` file, the JMS Router is configured in the `<jms-router>` element. A `<jms-router>` element consists of zero or more `<router-job>` elements. A JMS Router job is defined in the element `<router-job>`.

Table 3–14, " JMS Router Settings" lists and describes the JMS Router elements of the `jms.xml` file.

Minimal Example: JMS Router in jms.xml

This example illustrates a minimal configuration for a single JMS Router job that uses an OEMS JMS In-Memory queue as a source and an OEMS JMS In-Memory queue as target, utilizing the globally deployed JMS Connector instance `OracleASjms` for JMS objects.

```
<?xml version="1.0"?>
<jms xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchema
Location="http://www.oracle.com/technology/oracleas/schema/jms-server-10_1.xsd"
schema-major-version="10" schema-minor-version="1">
  <!-- OracleAS JMS server configuration -- omitted for brevity  -->
  <jms-server>
     ...
  </jms-server>

  <!-- JMS Router configuration -->
  <jms-router max-local-concurrency="-1" >

     <!-- Minimal configuration for a JMS Router job.-->
     <router-job job-name="routerjob1">

        <!-- The name of a JMS Router destination -->
        <message-source>OracleASjms/Topics/jms/mySource</message-source>

        <!-- Connection factory for the message source. -->
        <source-connection-factory>OracleASjms/MyCF</source-connection-factory>

        <!-- The name of a JMS Router destination -->
        <message-target>OracleASjms/Queues/jms/myTarget</message-target>

        <!--Connection factory for the message target. -->
        <target-connection-factory>OracleASjms/MyCF</target-connection-factory>

     </router-job>
  </jms-router>

</jms>
```

Syntax Example: JMS Router in jms.xml

This example illustrates the syntax of the JMS Router portion of `jms.xml` by defining a router job that provides values for all available attributes. It defines a configuration for a single JMS Router job that uses OEMS JMS In-Memory queue as source and OEMS JMS Database queue as target, utilizing the JMS Connector instance `OracleASjms` for the source JMS objects, and the JMS Connector instance `ojmsaq` for the target objects.

```
<?xml version="1.0"?>
<jms xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchema
Location="http://www.oracle.com/technology/oracleas/schema/jms-server-10_1.xsd"
schema-major-version="10" schema-minor-version="1">
  <!-- OracleAS JMS server configuration -- omitted for brevity  -->
  <jms-server>
     ...
  </jms-server>

  <!-- JMS Router configuration -->
  <jms-router max-local-concurrency="-1" >

      <router-job
          job-name="routerjob1"
          max-retries="16"
          polling-interval="5"
          pause-job="false"
          use-exception-queue="true"
          batch-size="30"
      >
          <!-- The name of an JMS Router source destination -->
          <message-source>OracleASjms/Topics/jms/mySource</message-source>

          <!-- Connection factory for the message source. -->
          <source-connection-factory>OracleASjms/MyCF</source-connection-factory>

          <!-- A message selector used at the message source -->
          <message-selector>color='blue'</message-selector>

          <!--This is the default subscriber name for this job as written to this
file by the JMS Router -->
          <subscriber-name>OracleASRouter_routerjob1</subscriber-name>

          <!--There is no need to specify the log queue for OracleAS JMS, but the
default value will be written back to this file by the JMS Router -->

<source-log-queue>OracleASjms/Queues/OracleASRouter_LOGQ</source-log-queue>
```

```
          <!-- An exception queue -->
          <exception-queue>OracleASjms/Queues/jms/myExQ</exception-queue>

          <!-- The name of an JMS Router target destination -->
          <message-target>ojmsaq/Queues/Queues/MyDBTarget</message-target>

          <!-Connection factory for the message target. -->
          <target-connection-factory>ojmsaq/CF</target-connection-factory>

          <!-- A log queue must be specified for all providers but OracleAS JMS.
This queue must already exist  ->
          <target-log-queue>ojmsaq/Queues/Queues/MyJMSRouterLog</target-log-queue>

      </router-job>
  </jms-router>

</jms>
```

Managing the Router

This section describes the JMS Router MBean operations for managing the JMS Router.

The JMS Router MBean enables you to:

- Start and stop individual JMS Router jobs
- Monitor router job status
- Monitor and query the JMS Router

Table 3–13, " JMS Router MBean Operations" on page 3-80 lists and describes the JMS Router MBean operations.

Path to the JMS Router MBean:

OC4J:Home > Administration tab > Services > JMS Providers > Click the icon. > Select the appropriate tab. > Related Links: OracleASJMSRouter

Router Logging

The JMS Router logs all significant events and error messages to the standard OC4J log file. The logger name is `oracle.j2ee.jms.router`.

JMS Router Status Information

You can access JMS Router run-time status information using the `RouterJobsStatus` and `RouterGlobalStatus` attributes of the JMS Router MBean.

Table 3–15 JMS Router and Router Job Status

Statistic	Description
`NumberJobs`	The number of configured jobs
`RouterState`	A string representing the JMS Router state
`Retries`	Number of times the JMS Router has failed to deliver a message for this job

Table 3–15 (Cont.) JMS Router and Router Job Status

Statistic	Description
`ExceptionQMessages`	Number of messages moved to exception queue by this job
`LastErrorTime`	If this job is in an error state, the time last error occurred
`TargetQMessages`	Number of messages propagated to target queue by this job
`JobState`	A string representing the state of this job

Error Handling

While processing a JMS Router job, the JMS Router may encounter various failures, such as an unreachable source or target destination or messaging operation failures. Based on the nature of the failures and the router job configuration, the JMS Router handles failures as follows:

If the JMS Router fails to enqueue a message to the target destination due to the content of the message and

- if no exception queue exists or the flag `useExceptionQueue` is set to `false` for the associated router job, then the JMS Router stops processing the router job.
- if an exception queue is provided for the job and the flag `useExceptionQueue` is set to `true` for the associated router job, then the JMS Router moves the message into the exception queue and the job continues processing subsequent messages.

When moving a message from the source destination to the exception queue, the JMS Router adds certain message properties to the original message to preserve error information. These messages are listed and described in Table 3–16, " Properties Added to Messages in Exception Queue".

Table 3–16 Properties Added to Messages in Exception Queue

JMS Property	Description
`oraMsgRouter_origMsgid`	ID of the source message
`oraMsgRouter_jobName`	Name of the router job
`oraMsgRouter_srcCF`	Name of the connection factory used for dequeue/enqueue
`oraMsgRouter_srcQName`	Name of the source queue from which the message was obtained
`oraMsgRouter_moveReason`	Reason the message was moved to exception queue
`oraMsgRouter_moveTime`	Time the message was moved to exception queue

If the failure is not due to the content of a message, the JMS Router will automatically retry the failing operation in the increasing intervals

```
(2^n) * (pollingInterval),
```

with a maximum of 15 minutes, until it either succeeds or stops processing the router job when it reaches the configurable maximum number of retries specified in the `maxRetries` attribute, as described in Table 3–14, " JMS Router Settings" on page 3-81.

If the JMS Router stops processing a router job because the number of retries specified in the `maxRetries` setting has been reached, then users can call `resetJob` to reset the failure count to `0` to resume processing of the job.

The failure count of each job is stored in memory. Therefore, when the JMS Router restarts, the failure count of every job is reset to `0`.

Pausing and Resuming a Job

A JMS Router job is in either `activated` or `deactivated` mode, as specified in the `pauseJob` setting, described in Table 3–14, " JMS Router Settings" on page 3-81.

The JMS Router does not process jobs that are in `deactivated` mode.

You can use the `pauseJob` operation in the JMS Router MBean to put a job in `deactivated` mode.

You can use the `resumeJob` operation in the JMS Router MBean to put a job in `activated` mode.

By default, jobs are created in `activated` mode.

Running In a Clustered OC4J Environment

In a clustered OC4J environment, a JMS Router instance may run on each OC4J instance. These JMS Router instances are configured independently and run without knowledge of each other. A JMS Router job that is configured on an OC4J instance is processed by only the JMS Router instance running in that OC4J instance.

- Creating or managing a JMS Router job must be done using the JMS Router MBean or jms.xml file of a particular OC4J instance.
- All JMS connection factories and JMS destinations that are referenced by JMS Router jobs defined on an OC4J instance must be accessible from that OC4J instance
- If an OC4J instance is terminated, no jobs defined for the JMS Router instance running on the OC4J instance will be processed.

As a rule, different JMS Router jobs must not share a common source, i.e., a queue or a durable subscriber. Otherwise the JMS Router jobs may run into unrecoverable failures. In a clustered OC4J environment, this rule applies across all OC4J instances in the cluster.

- A JMS queue that is not an OEMS JMS In-Memory or File-Based distributed destination, can only be the source of one JMS Router job across the entire OC4J cluster.
- A JMS topic that is not an OEMS JMS In-Memory or File-Based distributed destination can be the source of multiple JMS Router jobs across the entire OC4J cluster. However, the associated durable subscriber names must be unique in the

cluster. Specifically, if subscriber names of the JMS Router jobs are specified, the subscriber names must be different. If the subscriber names are not specified, the JMS Router job name must be unique across the cluster for jobs sharing the same topic as a source, since the JMS Router generates the durable subscriber names based on the JMS Router job names.

- Each copy of an OEMS JMS In-Memory or File-Based distributed destination on an OC4J instance in a cluster is treated by the JMS Router as a separate JMS destination. In order to route messages from an OEMS JMS In-Memory or File-Based distributed destination, a JMS Router job using the destination as the source must be defined on each OC4J instance.

Routing with Remote Destinations

The JMS Router can route messages from one OC4J instance to another via OEMS JMS by using OEMS JMS remote connection factories. For more information, see Custom Topologies on page 3-68.

If a JMS Router job is defined using a connection factory for a remote OEMS JMS instance, any JMS destinations accessed using that connection factory must be defined on both the remote and local OC4J instances. If the source connection factory is remote, these JMS destinations include the message source, the JMS Router source log queue, and, optionally, an exception queue. If the target connection factory is remote, applicable JMS destinations include the message target, and the JMS Router target log queue.

The JMS Router log queue for OEMS JMS is named `OracleASRouter_LOGQ`. This queue is normally created automatically when a JMS Router job using OEMS JMS is first configured. However, when the message source or message target is remote, it may need to be created manually on the remote instance. Alternatively, a different persistent log queue may be created on the remote instance and specified when creating the JMS Router job.

4 Data Sources

This chapter discusses Data Sources in Oracle Containers for J2EE (OC4J). It contains the following sections:

- Data Source Types, including:
  - Managed Data Sources
  - Native Data Sources
- Defining Data Sources, including:
  - Defining a Connection Pool
  - Defining a Managed Data Source
  - Defining a Native Data Source
  - Using Password Indirection
- Getting a Connection From a DataSource
- Connections
- Statements
- Transactions
- Configuring Data Source Objects
- Configuration Examples
- Using High Availability and Fast Connection Failover
- Using JDBC Drivers

Tasks

The following OC4J Data Source tasks are described in this chapter:

- Defining a Connection Pool
- Defining a Managed Data Source
- Defining a Native Data Source
- Defining Fatal Error Codes
- Using Password Indirection
- Establishing a Connection

- Using Connection Pools for Managed Data Sources
- Using Connection Proxies with Managed Data Sources
- Getting a Connection From a DataSource
- Setting the JDBC Statement Cache Size in Data Sources
- Configuring Data Source Objects
- Using High Availability and Fast Connection Failover
- Using JDBC Drivers

What's New in 10.1.3

The following OC4J Data Source features and behaviors are new for this release:

- Data source configuration can be performed entirely in the Oracle Enterprise Manager 10g Application Server Control Console.
- The OC4J Data Sources types are managed data sources and native data sources, replacing emulated, non-emulated, and native.
- OC4J tracks local transactions by default. This ensures that OC4J maintains consistency across different database drivers and vendors. More importantly it allows OC4J to avoid transaction corruption due to the interleaving of transactions.
- New connection caching mechanism that is uniform across Oracle data sources and offers integrated Real Application Clusters (RAC) failover support. For more information, see "Connection Pools" on page 4-23, Table 4–3, " Connection Pool Attributes" on page 4-23, and "Implicit Connection Cache" on page 4-28.

Deprecated

The following item(s) are deprecated in this release:

- The class `OracleConnectionCacheImpl` is deprecated in OC4J 10.1.3. This class does not support multiple schemas, only one user can be cached.
- The `stmt-cache-size` attribute is deprecated. In 10.1.3, to configure JDBC statement caching for a data source, use the `num-cached-statements` attribute to set the size of the cache instead of the `stmt-cache-size` attribute.

Additional Documentation

The following documents give additional data source information:

- *Oracle Database JDBC Developer's Guide and Reference*
- How-To overview documents with code examples available at:
  `http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`
- Additional articles available at:
  `http://www.oracle.com/technology/tech/java/oc4j/index.html`

Data Source Types

A data source is a Java object that implements the `javax.sql.DataSource` interface. Data sources offer a portable, vendor-independent method for creating connections to databases. A data source object's properties are set so that it represents a particular database. An application can use a different database by changing the data source's properties; no change in the application code is required.

OC4J provides two types of data sources: managed and native.

Managed Data Sources

Managed data sources are managed by OC4J. This means that OC4J provides critical system infrastructure such as global transaction management, connection pooling, and error handling. A managed data source is an OC4J-provided implementation of the `javax.sql.DataSource` interface that acts as a wrapper to a JDBC driver or data source. J2EE components access managed data sources via JNDI with no knowledge that the data source implementation is a wrapper.

Managed data sources differ from native data sources as follows:

- The connections retrieved from a managed data source can participate in global transactions.
- A managed data source uses OC4J's connection pool and statement cache.
- A connection returned from a managed data source is wrapped with an OC4J Connection proxy.

Native Data Sources

Native data sources implement the `javax.sql.DataSource` interface and are provided by JDBC driver vendors (such as Oracle and DataDirect.) Native data sources differ from managed data sources as follows:

- The connections retrieved from a native data source cannot participate in global transactions.
- A native data source does not use OC4J's connection pool or statement cache.
- A connection returned from a native data source is not wrapped with an OC4J Connection proxy.

For information on configuring native data sources, see Defining a Native Data Source on page 4-6.

Defining Data Sources

The Application Server Control Console is your primary tool for managing data sources including operations to create data sources and connection pools, remove data sources and connection pools, and modify existing data sources and connection pools.

The online help in the Application Server Control Console provides useful information on data source settings.

When the data sources are modified in the Application Server Control Console, the data source settings are immediately persisted to the `data-sources.xml` file for that application.

The default application's data sources configuration file is located in `$J2EE_HOME/config/data-sources.xml`.

Each `<data-source>` tag in this file represents one data source that is bound into JNDI and therefore accessible from client components (servlets, EJBs, etc.)

This section shows examples of data source definitions in the `data-sources.xml` configuration file.

For more information on defining data sources, see "Configuring Data Source Objects" on page 4-17.

For examples of the `data-sources.xml` file, see the Configuration Examples section on page 4-28.

Configuration Notes

When a data source is added, edited, or deleted directly in the `data-sources.xml` file, you must restart OC4J to implement the changes.

When a data source is added, edited, or deleted in the Application Server Control Console, it is not necessary to restart OC4J to implement the changes.

Each time you save your data source settings in the Application Server Control Console, a new `data-sources.xml` file is generated and comments are lost.

The `jndi-name` of a data source must be unique for an application. You cannot have duplicate jndi names in the same `data-sources.xml` file. For 10.1.3, OC4J throws an exception if jndi location is repeated within a `data-sources.xml` file. You may specify a `jndi-name` in the `data-sources.xml` file for an application that has already been specified in the global `data-sources.xml` file. In this case, jndi locations specified in the application `data-sources.xml` serve as overrides for those locations in the application context.

Data sources are defined using the `<data-source>` tag. The `class` attribute can be set to any full-path class name of the object implementing the `javax.sql.DataSource` interface.

Certain well-defined properties are specified in this tag as well such as `user`, `password`, `url`, and so on.

Properties that OC4J does not know about can be defined using a `<property>` tag.

OC4J-specific implementations DO NOT have any extra properties that can be set. That is, all of their properties are set be means of the values specified in the `<data-source>` tag.

Oracle-specific implementations such as `oracle.jdbc.pool.OracleDataSource` DO have properties that can be set by means of the `<property>` sub tag of the `<data-source>` tag.

For information about Oracle JDBC drivers, see the *Oracle Database JDBC Developer's Guide and Reference*.

Non-Oracle vendor-specific implementations, such as DB2, Sybase, SQLServer, etc. most likely also have properties outside of the ones defined in the `<data-source>` tag that can be defined using the `<property>` tag. It is up to the user to determine the properties of non-Oracle data source implementations.

Defining a Connection Pool

A managed data source uses a connection pool to efficiently manage connections. If you will create managed data sources, you must define at least one connection pool and its connection factory.

OC4J provides the connection pool feature to increase efficiency for managed data sources by maintaining a cache of physical connections that can be reused. When a client closes a connection, the connection gets placed back into the pool so that another client can use it. A connection pool improves performance and scalability by allowing multiple clients to share a small number of physical connections.

Note: The terms "connection pool" and "connection cache" are synonymous.

For more information on the nature and purpose of connection pools, see Using Connection Pools for Managed Data Sources on page 4-8.

Path to Connection Factory and Connection Pool Settings in the Application Server Control Console

OC4J:Home > Administration tab > Task Name: Services > JDBC Resources: Go To Task > Drill down to desired settings.

Here is an example of defining a connection pool in the `data-sources.xml` file instead of the Application Server Control Console.

```
<connection-pool name="myConnectionPool">
    <connection-factory
        factory-class="oracle.jdbc.pool.OracleDataSource"
        user="scott"
        password="tiger"

url="jdbc:oracle:thin:@//localhost:1521/oracle.regress.rdbms.dev.us.oracle.com"/>
         <property name="foo" value="bar"/>
    </connection-factory>
</connection-pool>
```

The `<connection-pool>` element contains a `name` attribute that uniquely identifies the connection pool. Other attributes define parameters for the connection pool such as the maximum number of connections that the connection pool will hold. A connection pool uses a connection factory (defined by the `<connection-factory>` element) to get physical connections from the database.

The `<connection-factory>` element contains the URL that the JDBC driver uses to connect to the database plus an optional default user and password that can be used to get connections from the database. The `factory-class` attribute defines the implementation class provided by the JDBC driver that is used to get the connections. The implementation class must be an implementation of one of the following:

- `java.sql.Driver`
- `javax.sql.DataSource`
- `javax.sql.XADataSource`
- `javax.sql.ConnectionPoolDataSource`.

For details on connection pool and connection factory settings, see Table 4–3, " Connection Pool Attributes" on page 4-23.

Defining a Managed Data Source

After you have defined at least one connection pool you can define a managed data source.

Path to Managed Data Source Settings

OC4J:Home > Administration tab > Task Name: Services > JDBC Resources: Go To Task > Data Sources > Create > Select Application > Select Data Source Type > Managed

Here's an example of a managed data source definition in the `data-sources.xml` file using the connection pool defined above:

```
<managed-data-source
    jndi-name="jdbc/ManagedDS"
    name="Managed DataSource Name"
    connection-pool-name="myConnectionPool"/>
```

The `name` attribute uniquely identifies the managed data source. The `jndi-name` attribute defines the location with which this data source will be placed into JNDI. The `connection-pool-name` attribute identifies the connection pool with which this managed data source will interact to get connections. This connection pool name corresponds to the value specified for the `name` attribute in the `<connection-pool>` element in the example in the example in the previous section "Defining a Connection Pool".

Defining a Native Data Source

A native data source has no dependencies on a connection pool. As such, a native data source definition includes data required to communicate with the underlying database.

Path to Native Data Source Settings

OC4J:Home > Administration tab > Task Name: Services > JDBC Resources: Go To Task > Data Sources > Create > Select Application > Select Data Source Type > Native

Here's an example of native data source definition in the `data-sources.xml` file:

```
<native-data-source
    name="nativeDataSource"
    jndi-name="jdbc/nativeDS"
    data-source-class="com.acme.DataSourceImpl"
    user="frank"
    password="frankpw"
    url="jdbc:acme:@localhost:5500:acme">
</native-data-source>
```

Additional data source configuration examples are shown in the article *Data Source Configuration in Oracle Application Server 10g* available at `http://www.oracle.com/technology/tech/java/newsletter/articles/oc4j_datasource_config.html`

The `name` attribute uniquely identifies the native data source. The `jndi-name` attribute defines the location with which this data source will be placed into JNDI. The `data-source-class` attribute defines the implementation class of the native data source and must be an implementation of `javax.sql.DataSource`. The `user` and `password` attributes define the default user and password. The `url` attribute defines the url that the data source will use to communicate with the database.

Defining Fatal Error Codes

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

Use the following procedure to define fatal error codes for non-Oracle databases or to add additional fatal error codes for Oracle databases.

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

```
<connection-pool name="myConnectionPool">
 <connection-factory factory-class="oracle.jdbc.pool.OracleDataSource"
   user="scott"
   password="tiger"

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

  <fatal-error-codes>
   <error-code code='10'/>
   <error-code code='20'/>
   <error-code code='30'/>
  </fatal-error-codes>
 </connection-factory>
</connection-pool>
```

Using Password Indirection

The `data-sources.xml` file requires passwords for authentication. Embedding these passwords without some kind of obfuscation poses a security risk. 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 retrieves 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 Containers for J2EE Security Guide*.

For example, if the native data source entry looks like:

```
<native-data-source
    name="nativeDataSource"
    jndi-name="jdbc/nativeDS"
    data-source-class="com.acme.DataSourceImpl"
    user="frank"
    password="frankpw"
    url="jdbc:acme:@localhost:5500:acme">
    </native-data-source>
```

You can replace the password, "`frankpw`", with the indirection symbol (`->`) and a user name (`frank`) as follows: `password="->frank"`. This assumes that a user named `frank` with the password `frankpw` has been created in a user manager.

You can configure password indirection in the Application Server Control Console.

To configure an indirect password for a data source directly in the `data-sources.xml` file and change the value of the `password` attribute so that its value is "->", followed either by the username or by the realm and user separated by a slash ("/"). Here is an example:

```
<native-data-source
    name="nativeDataSource"
    jndi-name="jdbc/nativeDS"
    data-source-class="com.acme.DataSourceImpl"
    user="frank"
    password="->frank"
    url="jdbc:acme:@localhost:5500:acme">
</native-data-source>
```

Note that there are `password` attributes for `<native-data-source>`, `<managed-data-source>`, and `<connection-factory>` elements.

Connections

OC4J data sources return two types of connections:

- **Managed Connections** - Connections returned from managed data sources come from a connection pool and are wrapped by a proxy class thus allowing OC4J to process SQL errors and to enlist these connections in global transactions. See "Using Connection Proxies with Managed Data Sources" on page 4-9.
- **Native Connections** - Connections returned from native data sources are not manipulated in any way by OC4J. That is, the connections are not wrapped, do not have proxies associated with them, and cannot participate in global transactions.

Establishing a Connection

A data source produces connections by communicating with a database. Typically a data source uses a URL to identify the machine, port, and database name, etc. that it uses to communicate with that database.

For a managed data source, the URL is defined in that managed data source's connection pool's connection factory. The connection factory's `url` attribute defines the URL that is used to communicate with the database. The JDBC driver defines the format of the URL. Several examples of the URL are provided throughout this document.

For a native data source, the URL is defined by the `url` attribute of the `<native-data-source>` element.

Using Connection Pools for Managed Data Sources

Basic implementations of data sources have a one-to-one correspondence between a client's connection object and a physical connection. When the client closes the connection, the physical connection is typically dropped. This incurs a lot of overhead each time a client retrieves a connection from the data source.

OC4J's managed data sources make frequent use of connection pools.

A connection pool can be used by more than one managed data source; that is, multiple managed data sources can share the same connection pool.

When a data source is defined to use the Oracle 10g JDBC driver, OC4J uses the sophisticated connection pool that is provided by the Implicit Connection Cache (ICC) that comes with that driver. OC4J data sources automatically use the ICC. All of the connection pool attributes described in Table 4–3, " Connection Pool Attributes" on page 4-23 apply to the ICC, unless otherwise specified. Some of the attributes apply only to Implicit-Connection-Cache-enabled data sources (`OracleDataSource` and `OracleXADataSource`). There is no additional configuration necessary to use the ICC.

For information on connection pool configuration settings, see Table 4–3, " Connection Pool Attributes" on page 4-23.

Connection pools are also provided for non-Oracle JDBC drivers and previous versions of Oracle JDBC drivers. An example of configuring a connection pool are described in the Configuration Examples section on page 4-28.

Using Connection Proxies with Managed Data Sources

When using managed data sources, OC4J wraps each connection retrieved from the connection pool with a proxy object. This proxy enables OC4J to provide transaction enlistment, exception handling, and logging.

Vendor-Specific Extensions

Clients can also use extensions to the `java.sql` interfaces that are provided by the vendor implementations of these interfaces. For example, the Oracle extension of the `java.sql.Connection` interface is the `oracle.jdbc.OracleConnection`. This interface provides Oracle-specific APIs that are not part of the `java.sql.Connection` interface. OC4J provides a configuration element that allows you to limit the interfaces that the proxy should implement so that the client has access to only those APIs. This configuration element can be used to specify additional interfaces for any of the `java.sql.*` interfaces. By default the proxies implement any public interface that is implemented by the underlying object.

For information on setting proxies, see Table 4–3, " Connection Pool Attributes" on page 4-23.

Here is an example of defining a connection proxy and a statement proxy for a connection pool in a `data-sources.xml` file instead of the Application Server Control Console.

```
<connection-pool name="myConnectionPool">
    <connection-factory
        factory-class="com.acme.AcmeDataSource"
        user="scott"
        password="tiger"
        url="jdbc:acme:@localhost:1234:acme"/>
        <property name="foo" value="bar"/>
        <proxy-interface sql-object="Connection"
        interface="com.acme.AcmeConnection"/>
        <proxy-interface sql-object="CallableStatement"
        interface="com.acme.AcmeCallableStatement"/>
    </connection-factory>
</connection-pool>
```

In this example, proxies generated for Connection objects would only expose the `com.acme.AcmeConnection` interface, regardless of what other interfaces are implemented by the underlying connection object. Likewise, proxies generated for Statement objects would only expose the `com.acme.AcmeStatement` interface. This gives the data source deployer a way to limit the interfaces exposed by the proxy objects.

Getting a Connection From a DataSource

This section provides sample code for getting a connection from a data source and executing a statement.

Notes:

Take care to always close a connection that is retrieved from a data source even when exceptions are thrown.

The string used to perform the lookup must match the value of a JNDI Location `jndi-name` setting in a managed data source or a native data source.

```
Connection connection = null;
try {
   InitialContext context = new InitialContext();
   DataSource ds = (DataSource) context.lookup( "jdbc/ManagedDS" );
   connection = ds.getConnection();
   Statement statement = connection.createStatement();
   statement.execute( "select * from dual" );
   statement.close();
}
catch( Exception exception ) {
   // process exception
}
finally {
   if ( connection != null )
   {
      try {
         connection.close();
      }
      catch( SQLException sqlException ){}
   }
}
```

Notes:

When using `getConnection()`, if no user/password is passed in, then the user and password specified in the definition of the connection factory is automatically used and the connection is successfully created. If different user/passwords are specified in connection factory, one pair in attributes and a different pair in properties, then `getConnection()`, uses the user/password specified in the properties.

Specifying the connection factory user/password is described in Table 4–4, " Connection Factory Attributes" on page 4-27.

Connection factory properties is discussed in Connection Factory Properties on page 4-21.

Retry

Under certain circumstances a data source may not be able to return a connection. The most common cause for this is when all of the connections in the connection pool are in use. You may want the data source to wait for a period of time and then check the connection pool to see if it has any available connections before returning.

There are two connection pool configuration settings that you can use to control the amount of time to wait if a connection is not available in the pool and the number of times to retry asking the connection pool for a connection.

The `max-connect-attempts` setting defines the number of times that a managed data source will retry getting a connection from the connection pool (when all of the connection pool's connections are in use.) The `connection-retry-interval` setting specifies the interval to wait (in seconds) before attempting to get a connection from the connection pool after the last failed attempt. For more on these settings, see Connection Pool Attributes on page 4-23.

Here is an example of configuring the retry in the `data-sources.xml` instead of the Application Server Control Console. The example sets the `max-connect-attempts` to `5` seconds and the `connection-retry-interval` to `3` seconds.

```
<connection-pool name="myConnectionPool"
max-connect-attempts="5"
connection-retry-interval="3">
  <connection-factory
    factory-class="oracle.jdbc.pool.OracleDataSource"
    user="scott"
    password="tiger"

url="jdbc:oracle:thin:@//localhost:1521/oracle.regress.rdbms.dev.us.oracle.com"/>
  </connection-factory>
</connection-pool>
```

Statements

Managed data sources provide caching and proxies to make working with statements more efficient.

Statement Caching with Managed Data Sources

Statement caching improves performance by caching executable statements that are used repeatedly and makes it unnecessary for programmers to explicitly reuse prepared statements. Statement caching eliminates overhead due to repeated cursor creation, repeated statement parsing and creation and reduces overhead of communication between the application server and the database server. The following is true about statement caching:

- Statement caching and reuse is transparent to an application.
- Each statement cache is associated with a physical connection. That is, each physical connection will have its own statement cache.
- The statement match criteria are the following:
  - The SQL string in the statement must be identical (case-sensitive) to one in the cache.
  - The statement type must be the same (`prepared` or `callable`).
  - The scrollable type of result sets produced by the statement must be the same (`forward-only` or `scrollable`).
  - Maximum Number of Statements Cached

Setting the JDBC Statement Cache Size in Data Sources

To lower the overhead of repeated cursor creation and repeated statement parsing and creation, you can use statement caching with database statements. To enable JDBC statement caching, which caches executable statements that are used repeatedly, configure a datasource to use statement caching. A JDBC statement cache is associated with a particular physical connection maintained by a datasource. A statement cache is not per data source so it is not shared across all physical connections. The JDBC statement cache is maintained in the middle-tier (not in the database server).

You can dynamically enable and disable statement caching programmatically using the `setStmtCacheSize()` method on the connection object.

To configure JDBC statement caching for a data source, use the `num-cached-statements` attribute to set the size of the cache. This attribute sets the maximum number of statements to be placed in the cache. If you do not specify the `num-cached-statements` attribute or set it to `0`, the statement cache is disabled.

The following XML sets the statement cache size to 200 statements:

```
<data-source>
 ...
num-cached-statements="200"
</data-source>
```

To set the `num-cached-statements` attribute, first determine how many distinct statements the application issues to the database. Then, set the size of the cache to this number. If you do not know the number of statements that your application issues to the database, you can use the JDBC performance metrics to assist you with determining the statement cache size. To use the statement metrics you need to set the Java property `oracle.jdbc.DMSStatementMetrics` to `true` for the OC4J.

Note: In 10.1.3, to configure JDBC statement caching for a data source, use the `num-cached-statements` attribute to set the size of the cache. The `stmt-cache-size` attribute is deprecated.

Statement Cache Size Resource Issues

Even though the `num-cached-statements` is specified for a data source, statements are cached per connection, not per data source or connection pool. In other words, each managed connection acquired from a given data source will maintain its own statement cache if `num-cached-statements` is greater than `0` for that data source.

You should be aware that statements held in a connection's statement cache may hold on to database resources. It is possible that the number of opened connections combined with the number of cached statements per connection could exceed the limit of open cursors allowed for the database. You may be able to avoid this problem by reducing the `num-cached-statements` value or by increasing the limit of open cursors allowed for the database.

Statement Proxies with Managed Data Sources

All implementations of the `java.sql.*` interfaces (managed data sources) are wrapped by OC4J with a proxy object. This includes the statement objects as well (`java.sql.Statement`, `java.sql.PreparedStatement`, and `java.sql.CallableStatement`).

For information on setting proxies, see the Connection Factory Proxy Interfaces tab description in Table 4–3, " Connection Pool Attributes" on page 4-23.

Under certain conditions, a connection proxy may rebind to a new physical connection. This can happen, for example, when a connection proxy is used across a transaction. When this occurs, any statement objects obtained through the connection proxy are no longer valid since they were created using the old physical connection. For this reason, a proxy fronts statement objects acquired from the physical connection as well. These statement proxies are associated with the connection proxy from which they were obtained so that they can monitor the association with the underlying physical connection. If the statement proxy determines that the physical connection associated with its connection proxy has changed, then it will acquire a new physical statement from the connection proxy.

Vendor-specific extensions to the `java.sql.Statement`, `java.sql.PreparedStatement`, and `java.sql.CallableStatement` interfaces can be made available to clients in the same manner as connections.

Transactions

J2EE supports two kinds of transactions:

- **Local Transactions** - A local transaction is internal to a single resource.
- **Global Transactions** - A global transaction is created by an external transaction manager and is used to scope work on multiple resources.

Transaction support, including local transactions and global transactions, is discussed in the Chapter 4, "Data Sources",

Local Transactions

When a managed data source is configured for local transactions it returns connections that can participate in local transactions but cannot participate in global transactions. This means that the connections will not be enlisted in global transactions. The data source will set the auto commit to true for retrieved connections. However, it is up to the client to determine how the connections will be used in local transactions. That is, the client can change the auto-commit mode by using `setAutoCommit()` on a connection.

To set a managed data source for local transactions, see the Transaction Level setting in Table 4–1, " Managed Data Source Settings" on page 4-18.

To configure a data source for local transactions in the `data-sources.xml` file instead of the Application Server Control Console, set the `tx-level` attribute to `"local"` (The default value is `"global"`.) Here's an example:

```
<managed-data-source
jndi-name="jdbc/ManagedDS"
name="Managed DataSource Name"
connection-pool-name="myConnectionPool"
tx-level="local"/>
```

Native data sources can only participate in local transactions so there is no setting for a native data source for transaction support.

It is possible that a connection marked as `local` will be used inside a global transaction. Although this case is not specifically addressed in the JDBC specification, the specification implies that if a connection is not participating in a distributed transaction then the connection behaves like a local connection. If a connection is not enlisted in a global transaction then it is not participating in the transaction. Therefore a connection produced by a data source that is configured with `local transaction` will be treated as if it is in a local transaction even if work is performed on it inside a global transaction. That is, if auto commit is set to `false`, then the work performed on that connection cannot be committed or rolled back until commit or rollback is called on the connection even if the commit or rollback is executed on the distributed transaction. In this case, the connection only appears syntactically with the transaction boundaries, but does not actually participate in that transaction semantically, that is, it is not enlisted.

For example:

- Get a connection, `lc`, from a data source configured for local transaction.
- Begin a global transaction.
- Get a connection, `gc`, from a data source that can be used in global transactions.
- Perform work on both connections.

  The work done on `lc` may be committed when the work is performed (if auto commit is set to `true`) or commit/rollback may be called on `lc` (if auto commit is set to `false`.)
- Commit or rollback the global transaction.

The work done on `gc` is now committed or rolled back. No work will be committed on `lc`.

- The work done on `lc` may be committed or rolled back by calling `commit` or `rollback` on the connection (assuming that auto commit is set to `false`.)

Here is a code example of the above steps:

```
Connection lc = localTxDataSource.getConnection();
userTransaction.begin();
Connection gc = globalTxDataSource.getConnection();
lc.doWork();
gc.doWork();
userTransaction.commit();
// At this point work done on gc is now committed.
//The work done on lc is NOT yet committed.
lc.commit();
// At this point work done on lc is now committed.
```

Local Transaction Management

Typically a local transaction begins when a client sets `autoCommit` to `false` on a connection and the local transaction ends when the client calls `commit()` or `rollback()` on that connection. If no transactional work has been performed on the connection when `autoCommit` is `false` then explicitly calling `commit()` or `rollback()` on the connection may be deemed as unnecessary (the driver may be smart enough to know that no transactional work was done so committing or rolling back is not necessary.)

OC4J determines that there is an active local transaction on a connection when `autoCommit` is `false` and any method, other than `commit()`, `rollback()`, `setAutoCommit(true)`, or `close()` has been called on the connection (note that OC4J cannot determine if the work done on the connection is actually transactional or not.) Calling `commit()`, `rollback()`, or changing the value of `autoCommit` ends the current local transaction. If `autoCommit` is `false` and a method (other than `commit()`, `rollback()`, `setAutoCommit(true)`, or `close`) is subsequently called on the connection then OC4J considers this the beginning of a new local transaction.

What happens when the client does not explicitly end the local transaction (by calling `commit()`, `rollback()`, or `setAutoCommit(true)`) for the connection? There are two cases to consider:

- In the first case the client there is an active local transaction and the user closes the connection.
- In the second case there is an active local transaction and the connection is used in a global transaction.

OC4J can handle these cases in two ways:

- OC4J can manage local transactions and intercede when the connection is closed or when the connection is used in the global transaction. More specifically, OC4J can implicitly end the local transaction by calling `commit()` or `rollback()`. OC4J can also throw an exception when the connection is closed or when the connection is used in a global transaction.
- OC4J cannot manage local transactions and will not intercede in these cases. More specifically, when a connection is closed or when a connection is used in a global

transaction, then the resource must determine how to end the local transaction (or not end it.) Note that when OC4J is configured to not manage local transactions it is possible that when a connection is placed back into the connection pool it will have an uncommitted local transaction active.

Global Transactions (XA)

When a managed data source is configured for global transactions, it returns connections that can participate in global transactions. A global transaction (also called a distributed transaction) enlists more than one resource in the transaction.

For information on how the transaction manager deals with global transactions when there are outstanding JCA local transactions, see "Local Transaction Management" on page 4-15.

For more on transactions, see Chapter 5, "OC4J Transaction Support".

To set a managed data source for global transactions, see the Transaction Level setting in Table 4–1, " Managed Data Source Settings" on page 4-18.

To configure a data source for global transactions in the `data-sources.xml` file instead of the Application Server Control Console, either do not include the `tx-level` attribute (The default is `"global"`.) or set the `tx-level` attribute to `global`. Here's an example:

```
<managed-data-source
    jndi-name="jdbc/ManagedDS"
    name="Managed DataSource Name"
    connection-pool-name="myConnectionPool"
    tx-level="global"/>
```

XA Recovery

When a global transaction fails, the transaction manager must perform XA recovery. To do this, it must have some information defined for it for each resource to be recovered. For data sources this means defining a recovery `username` and `password` for each connection factory that uses a `javax.sql.XADataSource` as its factory-class.

See the User and Password settings in Table 4–3, " Connection Pool Attributes" on page 4-23

Here's an example of configuring XA recovery in the `data-sources.xml` file instead of the Application Server Control Console. Note the `xa-recovery-config` node.

```
<connection-pool name="myConnectionPool">
    <connection-factory
        factory-class="oracle.jdbc.xa.client.OracleXADataSource"
        user="scott"
        password="tiger"

url="jdbc:oracle:thin:@//localhost:1521/oracle.regress.rdbms.dev.us.oracle.com"/>
            <xa-recovery-config>
                <password-credential>
                    <username>system</username>
                    <password>manager</password>
                </password-credential>
            </xa-recovery-config>
        </connection-factory>
</connection-pool>
```

Notes:

- If the `factory-class` is an instance of `java.sql.Driver`, `javax.sql.DataSource`, or `javax.sql.ConnectionPoolDataSource` then the XA recovery configuration is not necessary. for more information, see "Emulating XA" on page 4-17.
- `OracleXADataSource` is also an instance of `javax.sql.XADataSource`.

Emulating XA

An emulated XAResource is an implementation of `javax.sql.XAResource` that emulates the semantics of the XA protocol. This means that during a global transaction, those connections that are associated with an emulated XAResource follow the semantics of XA by using local transactions rather than transaction branches controlled explicitly as a subset of the global unit of work.

Emulating an XAResource is needed to support JDBC drivers that do not supply implementations of `javax.sql.XADataSource`. Also, an emulated XAResource will perform faster than a true XAResource since the performance of an emulated XAResource will not be affected by the overhead associated with true two-phase commit.

Note that using emulated XAResources can lead to inconsistent or non-recoverable outcomes when more than one XAResource is enlisted and at least one of them is emulated. The reason for this is that during the prepare phase an emulated XAResource does not perform a true prepare because it is using a local transaction. One way that this can be a problem is that when commit is called on the emulated XAResource its local transaction may have timed out which causes the local transaction's commit to fail which in turn causes the entire transaction to be in an inconsistent state.

OC4J automatically determines when to emulate XA behavior. It does this by introspecting the connection factory's factory-class object (the `factory-class` attribute specifies the object that is used by the connection factory to create connections for the connection pool.) If this object is an instance of `javax.sql.XADataSource` then OC4J does NOT emulate XA. If this object is an instance of `java.sql.Driver`, `javax.sql.DataSource`, or `javax.sql.ConnectionPoolDataSource` then OC4J emulates XA behavior for this data source.

Configuring Data Source Objects

This section lists and describes the configuration settings for the various data-source-related objects, whether you make the settings in the Application Server Control Console or directly in the `data-sources.xml` file.

The settings are discussed in the following tables:

- Table 4–3, " Connection Pool Attributes"
- Table 4–1, " Managed Data Source Settings"

- Table 4–2, " Native Data Source Settings"

For more information on configuring data source objects, see "Defining Data Sources" on page 4-3.

Managed Data Sources

You must define at least one connection pool before you can define a managed data source.

Path to Managed Data Source Settings

OC4J:Home > Administration tab > Task Name: Services > JDBC Resources: Go To Task > Data Sources > Create > Select Application > Select Data Source Type > Managed

- Each `<managed-data-source>` tag defines one managed data source.
- Attributes are described in Table 4–1, " Managed Data Source Settings" on page 4-18.

Here's an example of a managed data source definition in `data-sources.xml`:

```
<managed-data-source
  jndi-name="jdbc/ManagedDS"
  name="Managed DataSource Name"
  connection-pool-name="myConnectionPool"/>
```

Table 4–1 Managed Data Source Settings

Application Server Control Console Property	<managed-data-source> Attribute	Description
Name	`name`	**Required**. The name of the data source. This must be a unique value. This name is used as the "name" key property of the `JDBCDataSource` managed object (`j2eeType=JDBCDataSource,name=data source name`).
JNDI Location	`jndi-name`	**Required**. The JNDI logical name for the data source object. OC4J binds an instance of the data source into the application JNDI namespace with this value.
Connection Pool	`connection-pool-name`	**Required**. The name of the connection pool that this managed data source uses to get connections.
Schema	`schema`	The path to the database schema for this data source when using the Orion CMP implementation for EJBs. This is provided for backward compatibility.
Transaction Level	`tx-level`	The transaction level supported by this managed data source. A value of `local` indicates that this data source and the connections it produces may participate in local transactions only. A value of `global` indicates that this data source and the connections it produces may participate in local and global transactions. Optional. Default = `global`.

Table 4–1 (Cont.) Managed Data Source Settings

Application Server Control Console Property	<managed-data-source> Attribute	Description
Local Transaction Management	`manage-local-transactions`	Specifies whether or not OC4J should manage local transactions. ■ If `true`, then the following happens when `autoCommit` is `false` for a connection: When `close()` is called on a connection, OC4J calls `commit()` on the connection before calling `close()`. If the connection is used in a global transaction, then OC4J calls `rollback()` on the connection and throws an exception. ■ If `false`, then the following happens when `autoCommit` is `false` for a connection: When `close()` is called on a connection, OC4J will not call `commit()` on the connection before calling `close()`. The JDBC driver determines how to handle the local transaction. If the connection is used in a global transaction, then OC4J will not call `rollback()` on the connection nor will it throw an exception. The JDBC driver determines how to handle the local transaction. Optional. Default = `true`.
SQL Object Management	`manage-sql-objects`	Determines how OC4J manages the `java.sql.*` objects. Managing one of these objects means that OC4J will wrap the object in a proxy and will intercept the methods that are invoked on the objects. A value of `all` means that OC4J will manage all `java.sql.*` objects. A value of `basic` means that OC4J will manage only `java.sql.Connection`, `java.sql.Statement`, `java.sql.PreparedStatement`, and `java.sql.CallableStatement`. Optional. Default = `basic`.
Login Timeout	`login-timeout`	The maximum time, in seconds, that this data source will wait while attempting to connect to a database. A value of `0` (zero) specifies that the timeout is the default system timeout if there is one; otherwise, it specifies that there is no timeout. Optional. Default = `0`
User	`user`	The default user to use to connect to the database. Optional. No default.
Password	`password`	The default password to use to connect to the database. Optional. No default.

Native Data Source

Path to Native Data Source Settings

OC4J:Home > Administration tab > Task Name: Services > JDBC Resources: Go To Task > Data Sources > Create > Select Application > Select Data Source Type > Native

- A native data source has no dependencies on a connection pool. As such, a native data source definition includes data required to communicate with the underlying database.
- Each `<native-data-source>` tag defines one native data source.
- Native data source settings are described in Table 4–2, " Native Data Source Settings" on page 4-20.
- Each "`native-data-source`" tag may have 0 or more "`property`" tags. Each "`property`" tag defines a property on the native data source instance. Reflection will be used on the native data source object to set the property's value. The property name must match (case sensitive) the name of the setter method used to set the property. For example, if there exists on the connection factory object a property named "`MyProp`" then a method named "`setMyProp`" will be called to set the property. Therefore the "`property" tag's` name must be "`MyProp`" in order to set the property correctly.

```
<native-data-source
   name='My Native DataSource'
   jndi-name='jdbc/nativeDs'
   data-source-class='com.acme.DataSourceImpl'
   user='frank'
   password='frankpw'
   url='jdbc:acme:@localhost:5500:acme'>
   <property name="foo" value="bar"/>
</native-data-source>
```

Additional data source configuration examples are shown in the article *Data Source Configuration in Oracle Application Server 10g* available at
`http://www.oracle.com/technology/tech/java/newsletter/articles/oc4j_datasource_config.html`

Table 4–2 Native Data Source Settings

Application Server Control Console Property	<managed-data-source> Attribute	Description
Name	`name`	**Required**. The name of the data source. This must be a unique value.
JNDI Location	`jndi-name`	**Required**. The JNDI logical name for the data source object. OC4J binds an instance of the data source into the application JNDI namespace with this value.
Data Source Class	`data-source-class`	**Required**. The name and path of the data source class implementation. This must be an implementation of `javax.sql.DataSource`.
URL	`url`	**Required**. The URL that will be used by the JDBC driver to connect to the database. The URL typically identifies the database host machine, port, and database name. For example: `jdbc:acme:@localhost:1234:acme`

Table 4–2 (Cont.) Native Data Source Settings

Application Server Control Console Property	<managed-data-source> Attribute	Description
Login Timeout	`login-timeout`	The maximum time, in seconds, that this data source will wait while attempting to connect to a database. A value of `0` (zero) specifies that the timeout is the default system timeout if there is one; otherwise, it specifies that there is no timeout. Optional. Default = `0`.
User	`user`	The default user to use to connect to the data source. Optional. No default.
Password	`password`	The default password to use to connect to the data source. Optional. No default.

Connection Pools and Connection Factories

Path to Connection Factory and Connection Pool Settings

OC4J:Home > Administration tab > Task Name: Services > JDBC Resources: Go To Task > Connection Pools > Drill down to desired settings.

Connection Factories

The `connection-factory` tag defines the connection factory that will be used to create connections for the data source.

If the `factory-class` is an implementation of `javax.sql.XADataSource`, then the connections retrieved from this connection factory will be able to participate in global transactions and will NOT have their XA capabilities emulated. If the `factory-class` is not an implementation of `javax.sql.XADatatSource`, then the connections retrieved from this connection factory will emulate the XA behavior when participating in global transactions.

Connection Factory Properties Each `<connection-factory>` tag can have zero or more `<property>` tags. Each `<property>` tag defines a property on the connection factory instance.

If the connection factory is an implementation of `java.sql.Driver` then each of these driver properties is placed in a `java.util.Properties` object that is used by the driver when it retrieves connections from the database.

If the connection factory is an implementation of `javax.sql.DataSource`, `javax.sql.ConnectionPoolDataSource`, or `javax.sql.XADataSource`, then reflection is used on the connection factory object to set the property's value.

The property name must match (case sensitive) the name of the setter method used to set the property. For example, if there exists on the connection factory object a property named `MyProp`, then a method named `setMyProp()` will be called to set the property. Therefore the `property` tag's name must be `MyProp` in order to set the property correctly.

Note:

It is possible to specify two different user/passwords in connection factory, one user/password in attributes and a different user/password in properties, as in the following example. In this case `getConnection()`, uses the user/password specified in the properties, in this example, scott2/tiger2, not the one specified in the attributes.

```
<connection-factory user="scott1" password="tiger1" ...>
    <property name="user" value="scott2" />
    <property name="password" value="tiger2" />
</connection-factory>
```

Connection Factory Proxy Interface Each `<connection-factory>` tag may have zero or more `<proxy-interface>` tags.

Each proxy interface is implemented by a proxy that wraps the connection objects returned by the connection factory and the `java.sql.*` objects created by these connection objects.

The SQL Object setting defines the `java.sql.*` object for which the proxy interface is defined. This must be one of the following:

- `"Array"`
- `"Blob"`
- `"CallableStatement"`
- `"Connection"`
- `"DatabaseMetaData"`
- `"ParameterMetaData"`
- `"PreparedStatement"`
- `"Ref"`
- `"resultSet"`
- `"ResultSetMetaData"`
- `"Savepoint"`
- `"SQLData"`
- `"SQLInput"`
- `"SQLOutput"`
- `"Struct"`
- `"Statement"`

The interface attribute defines the fully qualified path of the interface that the proxy to this object will implement.

There may be more than one proxy interface defined for each SQL object.

The `interface` attribute defines the fully qualified path of the interface that the proxy to this object will implement.

There may be more than one `proxy-interface` tag defined for each SQL object.

The `<xa-recover-config>` tag defines the information needed for the transaction manager to perform recovery when a global transaction fails. The `<username>` sub tag defines the username used to perform the recovery. The `<password>` sub tag defines the password used to perform recovery.

Connection Properties

The `<connection-properties>` tag defines the connection properties that will be set on the connection factory when the connection factory is an instance of `oracle.jdbc.pool.OracleDataSource` (including instances that are derived from `oracle.jdbc.pool.OracleDataSource`). Each connection property is defined by the `<property>` sub-tag. There may be 0 - N `<property>` sub-tags defined for the `<connection-properties>` tag.

Connection Pools

A managed data source uses a connection pool to efficiently manage connections. If you will create managed data sources, you must define at least one connection pool and its connection factory.

The `<connection-pool>` tag defines one connection pool.

Each `<connection-pool>` tag must have one `<connection-factory>` tag.

Table 4–3 Connection Pool Attributes

Application Server Control Console Setting	<connection-pool> Attribute	Description
Name	`name`	**Required**. The name of the connection pool. This must be a unique value.
Minimum Number of Connections	`min-connections`	The minimum number of connections that the connection pool will maintain. Optional. Default = `0`. The `min-connections` setting specifies the minimum number of connections that will be kept in the pool at any given time assuming the following activity: ■ There are no connections in use. If there are connections in use, then they are not in the pool, so there may be < min-connections> in the pool. ■ There have been sufficient connections created and were simultaneously in use such that the connection pool was required to create those connections. For example, if `min-connections` is `10` and only 2 connections were ever used, then the number of connections available in the pool would be 2. OC4J will not create connections unnecessarily.

Table 4–3 (Cont.) Connection Pool Attributes

Application Server Control Console Setting	<connection-pool> Attribute	Description
Maximum Number of Connections	`max-connections`	The maximum number of connections that the connection pool can contain. A value of `0` indicates: ■ The data source connection pool is off. Connections are not pooled. ■ All other connection pool settings are ignored. A negative value indicates that the connection pool is on and there is no maximum limit. Optional. Default = No limit. When the session starts, if `max-connections` is set lower than `min-connections`, then `min-connections` is reset to the value of `max-connections`.
Initial Size of Connection Cache	`initial-limit`	The size of the connection cache when the cache is initially created or reinitialized. When this property is set to a value greater than `0`, that many connections are pre-created and are ready for use. This parameter is typically used to reduce the ramp-up time in priming the cache to its optimal size. Optional. Default = `0`. When the `initial-limit` of a connection pool is greater than `1`, but the `user/password` is not provided in the connection-factory, OC4J fails to start and throws an error. See the note after Table 4–4, " Connection Factory Attributes" on page 4-27. When the session starts, if `initial-limit` is set greater than `max-connections` (for example, `initial-limit=10` and `max-connections=5`), then only the `max-connections` number (5) of connections will be initialized. When the session starts, if `initial-limit` is set less than `min-connections`, (for example, `initial-limit=10` and `min-connections=15`), then only the `initial-limit` number (10) of connections will be initialized. Later on, when more connections are called, the `min-connections` number of connections will be maintained for the pool.
Wait for Used Connection Timeout	`used-connection-wait-timeout`	The amount of time to wait, in seconds, for a used connection to be released by a client. This parameter only applies when the maximum number of connections has been retrieved from the data source and are in use. In this case when a client tries to borrow a connection from the pool and all connections are in use, the connection pool will wait for a connection to be released back to the pool. Optional. Default = `0`. For a timeout setting to be enforced, the `property-check-interval` must be set lower than that timeout setting.

Table 4–3 (Cont.) Connection Pool Attributes

Application Server Control Console Setting	<connection-pool> Attribute	Description
Inactivity Timeout	`inactivity-timeout`	The amount of time to wait, in seconds, that an unused connection may be inactive before it is removed from the pool. Optional. Default = `60`. For a timeout setting to be enforced, the `property-check-interval` must be set lower than that timeout setting.
Login Timeout	`login-timeout`	The maximum amount of time, in seconds, that this data source will wait while attempting to connect to a database. A value of `0` specifies that the timeout is the default system timeout if there is one; otherwise, it specifies that there is no timeout. Optional. Default = `0`. For a timeout setting to be enforced, the `property-check-interval` must be set lower than that timeout setting.
Connection Retry Interval	`connection-retry-interval`	The interval to wait, in seconds, the before retrying a failed connection attempt. This parameter is used in conjunction with `max-connect-attempts`. Optional. Default = `1`.
Maximum Connection Attempts	`max-connect-attempts`	The number of times to retry making a connection. This parameter is used in conjunction with `connection-retry-interval`. Optional. Default = `3`.
Validate Connection	`validate-connection`	Oracle Implicit Connection Cache only. Indicates whether or not a connection, when borrowed from the pool, will be validated against the database. Validation is performed by the SQL statement specified as the value of the `validate-connection-statement` parameter. A value of `true` indicates that when a connection is borrowed from the connection pool, the SQL statement is executed to verify that the connection is valid. Optional. Default = `false`, meaning that no validation is performed.
SQL Statement for Validation	`validate-connection-statement`	Oracle Implicit Connection Cache only. If `validate-connection` is `true`, the SQL statement executed when a connection is borrowed from the pool. Optional. No default.
Maximum Number of Statements Cached	`num-cached-statements`	The maximum number of SQL statements that should be cached for each connection. Any value greater than `0` automatically enables statement caching for the data source. Optional. Default = `0`. For more detail, see "Setting the JDBC Statement Cache Size in Data Sources" on page 4-12.

Table 4–3 (Cont.) Connection Pool Attributes

Application Server Control Console Setting	<connection-pool> Attribute	Description
Max Active Time for a Used Connection	`time-to-live-timeout`	Oracle Implicit Connection Cache only. The maximum time, in seconds, a used connection may be active. When this timeout expires, the used connection is unconditionally closed, the relevant statement handles canceled, and the connection is returned to the connection pool. Optional. Default = `-1` means that the feature is not enabled. For a timeout setting to be enforced, the `property-check-interval` must be set lower than that timeout setting.
Abandoned Connection Timeout	`abandoned-connection-timeout`	Oracle databases only. The amount of time to wait, in seconds, that an unused logical connection may be inactive before it is removed from the pool. This parameter is similar to `inactivity-timeout`, but on a logical connection borrowed from the cache by the user. When set, JDBC monitors SQL database activity on this logical connection. For example, when a `stmt.execute()` is invoked on this connection, a heart beat is registered to convey that this connection is active. The heart beats are monitored only at places (to lower the cost of monitoring), that result in database execute calls. If a connection has been inactive for the specified amount of time, the underlying PooledConnection is reclaimed and returned to the cache for reuse. Optional. Default = `-1`, indicating that the feature is disabled. For a timeout setting to be enforced, the `property-check-interval` must be set lower than that timeout setting.
Disable Server Connection Pooling (checkbox)	`disable-server-connection-pooling`	Whether or not to disable the application server's connection pool. This parameter is provided because some JDBC drivers provide connection pooling inside the driver. f the JDBC driver is Oracle and the driver is using the Implicit Connection Cache, then this parameter is ignored. Optional. Default = `false`, indicating that the pool is enabled.

Table 4–3 (Cont.) Connection Pool Attributes

Application Server Control Console Setting	<connection-pool> Attribute	Description
Enforce Timeout Limits Interval	`property-check-interval`	Oracle databases only. Used with Oracle databases only. The time interval, in seconds, for the cache daemon thread to enforce the time out limits. Optional. Default = `900`. For a timeout setting to be enforced, the `property-check-interval` must be set lower than that timeout setting.
Lower Threshold Limit On Pool	`lower-threshold-limit`	Oracle databases only. Used with Oracle databases only. The lower threshold limit on the connection pool as a percentage of the value indicated in `max-connections`. Optional. Default = `20` percent.

Table 4–4 lists and describes the connection factory attributes.

Table 4–4 Connection Factory Attributes

Application Server Control Console Setting	<connection-factory> Attribute	Description
Connection Factory Class	`factory-class`	**Required**. The name and path of the connection factory class that will be used to create connections for the data source. This class is provided by the JDBC driver. For example: `com.acme.AcmeDataSource` This class must be an implementation of `java.sql.Driver`, `javax.sql.DataSource`, `javax.sql.ConnectionPoolDataSource`, or `javax.sql.XADataSource`. If the `factory-class` is an implementation of `javax.sql.XADataSource` then the connections retrieved from this connection factory will be able to participate in global transactions and will NOT have their XA capabilities emulated. If the `factory-class` is not an implementation of `javax.sql.XADatatSource` then the connections retrieved from this connection factory will emulate the XA behavior when participating in global transactions.
URL	`url`	**Required**. The URL that will be used by the JDBC driver to connect to the database. The URL typically identifies the database host machine, port, and database name. For example: `jdbc:acme:@localhost:1234:acme`

Table 4–4 (Cont.) Connection Factory Attributes

Application Server Control Console Setting	<connection-factory> Attribute	Description
User	`user`	The default user to use to connect to the database. Optional. No default. When the initial-limit of a connection-pool is greater than `1`, but the `user/password` is not provided in the connection-factory, OC4J fails to start and throws an error. See the note after this table.
Password	`password`	The default password to use to connect to the database. Optional. No default.
Login Timeout	`login-timeout`	The maximum amount of time, in seconds, that this data source will wait while attempting to connect to a database. A value of `0` specifies that the timeout is the default system timeout if there is one; otherwise, it specifies that there is no timeout. Optional. Default = `0`.

Note: When the initial-limit of a connection-pool is greater than `1`, but the `user/password` is not provided in the connection-factory, OC4J fails to start and throws an error.

SEVERE: Error occurred initializing connectors. Exception is: Error creating data source connection pool. Exception: oracle.oc4j.sql.DataSourceException: Could not get/create instance of ConnectionCacheManager. Exception: User credentials doesn't match the existing ones com.evermind.server.ApplicationStateRunning initConnector

This error occurs because there is no user/password to initialize the connections in the pool.

Implicit Connection Cache The Oracle Implicit Connection Cache (ICC) is a sophisticated connection pool that comes with the Oracle 10*g* JDBC driver. OC4J data sources automatically use the ICC. There is no additional configuration necessary to use the ICC.

Table 4–3, " Connection Pool Attributes" lists and describes the Connection Pool attributes. All of the connection pool attributes described in Table 4–3 apply to the Implicit Connection Cache (ICC), unless otherwise specified. Some of the attributes (identified) apply only to Implicit-Connection-Cache-enabled data sources (`OracleDataSource` and `OracleXADataSource`).

Configuration Examples

This section shows examples of data source definitions in the `data-sources.xml` configuration file.

The default application's data sources configuration file is located in `$J2EE_HOME/config/data-sources.xml`.

Each application can have its own `data-sources.xml` file. If an application has its own, then it is located in the root directory of the application.

This section contains the following information:

- Syntax of the data-sources.xml File
- Examples: Configuring Data Sources, including:
  - Example: Native Data Source
  - Example: Managed Data Source Using an XADataSource Connection Factory
  - Example: Managed Data Source Using a DataSource Connection Factory
  - Example: Managed Data Source Using a Driver Connection Factory
  - Example: Defining Proxy Interfaces
  - Example: Defining XA Recovery
- Examples: Configuring Transaction Level, including:
  - Global
  - Local
- Examples: Configuring Fast Connection Failover, including:
  - Thin
  - OCI

Syntax of the data-sources.xml File

Data source settings are persisted in an Enterprises Application's `data-sources.xml` file. Each `<data-source>` tag in this file represents one data source that is bound into JNDI and therefore accessible from client components (servlets, EJBs, etc.)

The following example describes the syntax of the `data-sources.xml` file. See the schema for details.

```
<managed-data-source
  attr1="val1"
  attr2="val2"
  …
/>

<native-data-source
  attr1="val1"
  attr2="val2"
  …>
      <property name="propertyName" value="propertyValue"/>
      …

</native-data-source>
<connection-pool
  attr1="val1"
  attr2="val2"
  …
>

      <connection-factory
            attr1="val1"
            attr2="val2"
            …>

      <proxy-interface sql-object="javaSQLObject" interface=""/>
      …
      <property name="propertyName" value="propertyValue"/>
      …

      <xa-recover-config>
            <password-credential>
                  <username></username>
                  <password></password>
            </password-credential>
      </xa-recovery-config>

      <fatal-error-codes>
          <error-code code="integerCode"/>
          …
       </fatal-error-codes>
       <connection-properties>
          <property name="propertyName value="propertyValue"/>
         …
       </connection-properties>

   </connection-factory>
</connection-pool
```

Populated examples

The following example shows populated examples of the `data-sources.xml` definitions:

```
<?xml version="1.0" standalone="yes"?>
<data-sources>
   <connection-pool name="myConnectionPool" max-connections="30">
     <connection-factory
       factory-class="oracle.jdbc.pool.OracleDataSource"
       user="scott"
       password="tiger"

url="jdbc:oracle:thin:@//localhost:1521/oracle.regress.rdbms.dev.us.oracle.com"/>
     </connection-factory>
   </connection-pool>

   <managed-data-source
     jndi-name="jdbc/ManagedDS"
     name="Managed DataSource Name"
     connection-pool-name="myConnectionPool"/>

   <native-data-source
     name="nativeDataSource"
     jndi-name="jdbc/nativeDS"
     data-source-class="com.acme.DataSourceImpl"
     user="frank"
     password="frankpw"
     url="jdbc:acme:@localhost:5500:acme">
   </native-data-source>

</data-sources>
```

Examples: Configuring Data Sources

This section provides examples of data source configuration file definitions.

Additional data source configuration examples are shown in the article *Data Source Configuration in Oracle Application Server 10g* available at
`http://www.oracle.com/technology/tech/java/newsletter/articles/oc4j_datasource_config.html`

Example: Native Data Source

```
<native-data-source
    name='My Native DataSource'
    jndi-name='jdbc/nativeDs'
    data-source-class='com.acme.DataSourceImpl'
    user='frank'
    password='frankpw'
    url='jdbc:acme:@localhost:5500:acme'>
        <property name="foo" value="bar"/>
</native-data-source>
```

Example: Managed Data Source Using an XADataSource Connection Factory

This data source does NOT emulate XA behavior. See "Emulating XA" on page 4-17 for more information about emulating XA behavior.

```
<managed-data-source
    name='My Managed DataSource'
    jndi-name='jdbc/managedDs_1'
    connection-pool-name='myConnectionPool'/>

<connection-pool
    name='myConnectionPool'
    min-connections='5'
    max-connections='25'>
    <connection-factory
        factory-class='oracle.jdbc.xa.client.OracleXADataSource'
        user='scott'
        password='tiger'

url='jdbc:oracle:thin:@//localhost:1521/oracle.regress.rdbms.dev.us.oracle.com'>
    </connection-factory>
</connection-pool>
```

Example: Managed Data Source Using a DataSource Connection Factory

This data source emulates XA behavior. See "Emulating XA" on page 4-17 for more information about emulating XA behavior.

```
<managed-data-source
    name='My Managed DataSource'
    jndi-name='jdbc/managedDs_1'
    connection-pool-name='myConnectionPool'/>

<connection-pool
    name='myConnectionPool'
    min-connections='5'
    max-connections='25'>
    <connection-factory
        factory-class='oracle.jdbc.pool.OracleDataSource'
        user='scott'
        password='tiger'

url='jdbc:oracle:thin:@//localhost:1521/oracle.regress.rdbms.dev.us.oracle.com'>
    </connection-factory>
</connection-pool>
```

Example: Managed Data Source Using a Driver Connection Factory

This data source emulates XA behavior. See "Emulating XA" on page 4-17 for more information about emulating XA behavior.

```
<managed-data-source
    name='My Managed DataSource'
    jndi-name='jdbc/managedDs_1'
    connection-pool-name='myConnectionPool'/>

<connection-pool
    name='myConnectionPool'
    min-connections='5'
    max-connections='25'>
    <connection-factory
        factory-class='oracle.jdbc.OracleDriver'
        user='scott'
        password='tiger'

url='jdbc:oracle:thin:@//localhost:1521/oracle.regress.rdbms.dev.us.oracle.com'>
    </connection-factory>
</connection-pool>
```

Example: Defining Proxy Interfaces

```
<connection-pool
    name='myConnectionPool'
    min-connections='5'
    max-connections='25'>
    <connection-factory
        factory-class='oracle.jdbc.pool.OracleDataSource'
        user='scott'
        password='tiger'

url='jdbc:oracle:thin:@//localhost:1521/oracle.regress.rdbms.dev.us.oracle.com'>
        <proxy-interface sql-object="Connection"
                       interface="oracle.jdbc.internal.OracleConnection"/>
        <proxy-interface sql-object="Statement"
                        interface="oracle.jdbc.OracleStatement"/>
        <proxy-interface sql-object="CallableStatement"
                        interface="oracle.jdbc.OracleCallableStatement"/>
        <proxy-interface sql-object="ResultSet"
                        interface="oracle.jdbc.OracleResultSet"/>
        <proxy-interface sql-object="PreparedStatement"
                        interface="oracle.jdbc.OraclePreparedStatement"/>
    </connection-factory>
```

```
</connection-pool>
```

Example: Defining XA Recovery

```
<connection-pool
    name='myConnectionPool'
    min-connections='5'
    max-connections='25'>
    <connection-factory
        factory-class='oracle.jdbc.xa.client.OracleXADataSource'
        user='scott'
        password='tiger'

url='jdbc:oracle:thin:@//localhost:1521/oracle.regress.rdbms.dev.us.oracle.com'>
        <xa-recovery-config>
            <password-credential>
                <username>system</username>
                <password>manager</password>
            </password-credential>
        </xa-recovery-config>
    </connection-factory>
</connection-pool>
```

Example: Connection Properties

```
<managed-data-source
   jndi-name="jdbc/managedDs_1"
   name="Managed DataSource"
   connection-pool-name="myConnectionPool"/>

<connection-pool
   name="myConnectionPool">
      <connection-factory
            factory-class="oracle.jdbc.pool.OracleDataSource"
            user="scott"
            password="tiger"

url='jdbc:oracle:thin:@//localhost:1521/oracle.regress.rdbms.dev.us.oracle.com'>
            <connection-properties>
               <property name="oracle.jdbc.RetainV9LongBindBehavior"
                  value="true"/>
            </connection-properties>
      </connection-factory>
</connection-pool>
```

For information on the connection properties, see "Connection Properties" on page 4-23.

Examples: Configuring Transaction Level

Global

Here's an example of configuring a managed data source for global transactions by setting the `tx-level` attribute to `global` in the `data-sources.xml` file.

```
<managed-data-source
   jndi-name="jdbc/ManagedDS"
   name="Managed DataSource Name"
   connection-pool-name="myConnectionPool"
   tx-level="global"/>
```

Local

Here's an example of configuring a managed data source for local transactions by setting the `tx-level` attribute to `"local"` in the `data-sources.xml` file. The default value is `"global"`.

```
<managed-data-source
   jndi-name="jdbc/ManagedDS"
   name="Managed DataSource Name"
   connection-pool-name="myConnectionPool"
   tx-level="local"/>
```

Examples: Configuring Fast Connection Failover

The following is an example of configuring a connection factory for fast connection failover.

Thin

```
<connection-factory
   factory-class="oracle.jdbc.pool.OracleDataSource"
   user="scott"
   password="tiger"
   url="jdbc:oracle:thin:@(DESCRIPTION=
(ADDRESS=(PROTOCOL=TCP)(HOST=cluster_alias) (PORT=1521))
 (CONNECT_DATA=(SERVICE_NAME=service_name)))"
          <property name="connectionCachingEnabled" value="true"/>
          <property name="fastConnectionFailoverEnabled" value="true"/>
</connection-factory>
```

OCI

The following is an example connection factory definition using OCI:

```
<connection-factory
   factory-class="oracle.jdbc.pool.OracleDataSource"
   user="scott"
   password="tiger"
   url="jdbc:oracle:oci:@myAlias"
</connection-factory>
```

Using High Availability and Fast Connection Failover

OC4J's data sources are fully integrated with the Oracle 10G JDBC driver and therefore automatically take advantage of High Availability (HA) and Fast Connection Failover (FCF).

Additional High Availability and Fast Connection Failover information is available in the following documents:

- *Oracle Database High Availability Overview*

- "Part VI, High Availability" of the *Oracle Database JDBC Developer's Guide and Reference*
- Additional information at:
  `http://www.oracle.com/technology/tech/java/oc4j/index.html`

Fast Connection Failover is a RAC/FaN client implemented in the JDBC Implicit connection cache. Its primary purpose is to guarantee the validity and availability of a connection. Hence Fast Connection Failover on the client side provides the following features:

- Rapid Dead Connection Detection (DCD) of connections in the Implicit connection cache
- Removes such stale or bad connections from the cache.
- Propagates errors to the caller to facilitate retries at higher layers.
- Connection redistribution when new a RAC instance joins.

To enable the Fast Connection Failover mechanism, the following properties and attributes must be set on the `<connection-factory>` tag for an `OracleDataSource` object.

Property	Description
`connectionCachingEnabled`	This is a Boolean property, and enables connection caching when set to `true`. By default connection caching is disabled and the property value is set to `false`.
`fastConnectionFailoverEnabled`	This property, when set to `TRUE`, enables the Fast Connection Failover mechanism. By default, Fast Connection Failover is disabled and the property value is set to `FALSE`.
`url`	This is an attribute on the <connection-factory> tag. When enabling Fast Connection Failover, the URL must be set using the service name syntax. The service name specified on the connection URL is used to map the connection cache to the service. If a SID is specified on the URL, when Fast Connection Failover is enabled, then an exception is thrown.

The following examples show valid and invalid syntax for URL usage on a connection cache setup for Fast Connection Failover.

Valid URL Usage

```
url="jdbc:oracle:oci:@TNS_ALIAS"

url="jdbc:oracle:oci:@(DESCRIPTION=
   (LOAD_BALANCE=on)
   (ADDRESS=(PROTOCOL=TCP)(HOST=host1) (PORT=1521))
   (ADDRESS=(PROTOCOL=TCP)(HOST=host2)(PORT=1521))
   (CONNECT_DATA=(SERVICE_NAME=service_name)))"

url="jdbc:oracle:oci:@(DESCRIPTION=
   (ADDRESS=(PROTOCOL=TCP)(HOST=cluster_alias) (PORT=1521))
   (CONNECT_DATA=(SERVICE_NAME=service_name)))"

url = "jdbc:oracle:thin@//host:port/service_name"
url = "jdbc:oracle:thin@//cluster-alias:port/service_name"

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)))"

url = "jdbc:oracle:thin:@(DESCRIPTION=
   (ADDRESS=(PROTOCOL=TCP)(HOST=cluster_alias) (PORT=1521))
   (CONNECT_DATA=(SERVICE_NAME=service_name)))"
```

Invalid URL Usage

```
url = "jdbc:oracle:thin@host:port:SID"
```

Enabling Fast Connection Failover in the data-sources.xml File

The following example shows enabling fast connection failover for a Native Data Source in the `data-sources.xml` file:

```
<native-data-source>
   name="nativeDataSource"
   jndi-name="jdbc/nativeDS"
   description="Native DataSource"
   data-source-class="oracle.jdbc.pool.OracleDataSource"
   user="scott"
   password="tiger"
   url="jdbc:oracle:thin:@localhost:1521:oracle">
  <property name="connectionCacheName" value="ICC1"/>
  <property name="connectionCachingEnabled" value="true"/>
  <property name="fastConnectionFailoverEnabled" value="false"/>
</native-data-source>
```

Using JDBC Drivers

This section discusses:

- Oracle JDBC Drivers
- JDBC Drivers for non-Oracle Databases

Oracle JDBC Drivers

This section has information about the Oracle JDBC OCI driver and the Oracle JDBC thin driver.

For more information about Oracle JDBC drivers, see the *Oracle Database JDBC Developer's Guide and Reference*.

OCI

The examples of Oracle data source definitions in this chapter use the Oracle JDBC Thin driver. However, you can use the Oracle JDBC OCI driver as well. Do the following before you start the OC4J server:

1. Install the Oracle Client on the same system on which OC4J is installed.
2. Set the OLE_HOME variable.

3. Set LD_LIBRARY_PATH (or the equivalent environment variable for your OS) to $OLE_HOME/lib.
4. 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 <connection-factory> element definition can have any of these forms:

- jdbc:oracle:oci:@ This TNS entry is for a database on the same system as the client, and the client connects to the database in IPC mode.
- jdbc:oracle:oci:@ TNS_service_name The TNS service name is an entry in the instance tnsnames.ora file.
- jdbc:oracle:oci:@ full_TNS_listener_description For more TNS information, see the Oracle Net Administrator's Guide.

Here is an example connection factory definition using OCI:

```
<connection-factory
    factory-class="oracle.jdbc.pool.OracleDataSource"
    user="scott"
    password="tiger"
    url="jdbc:oracle:oci:@myAlias"
</connection-factory>
```

Thin

The examples of Oracle data source definitions in this chapter use the Oracle JDBC Thin driver.

Notes on Oracle JDBC-OCI driver upgrade in the Oracle Application Server

It is not possible to upgrade to an arbitrary Oracle JDBC-OCI driver version due to client library compatibility constraints. Upgrading to OCI driver versions with matching Oracle Client libraries that are installed within the Oracle Application Server 10*g* (10.1.2) is supported. For example, Oracle JDBC 10.1.x drivers are supported, but the Oracle JDBC 9.2.x drivers are not.

Where the use of JDBC-OCI within the Oracle Application Server is supported, it is also necessary for the `opmn.xml` entry for each OC4J instance to propagate appropriate ORACLE_HOME and library path values to its startup environment.

The environment variable `ORACLE_HOME` is common to all platforms, but the name of the environment variable that specifies the library path is different depending on the operating systems:

- `LD_LIBRARY_PATH` for Solaris
- `SLIB_PATH` for AIX
- `SHLIB_PATH` for HP-UX
- `PATH` for Windows

The generic syntax for specifying the library paths in `opmn.xml` looks like this:

```
<prop name="<LIB_PATH_VARIABLE>" value="<LIB_PATH_VARIABLE_VALUE>"/>
```

where `<LIB_PATH_VARIABLE>` should be replaced with the appropriate platform-specific variable name that specifies the library path, and

`<LIB_PATH_VARIABLE_VALUE>`

should be replaced with that variable's value.

Here is an example, assuming the Solaris OS:

```
    <process-type id="OC4J_SECURITY" module-id="OC4J">
      <environment>
        <variable id="ORACLE_HOME"
  value="/u01/app/oracle/product/inf10120"/>
        <variable
          id="LD_LIBRARY_PATH"
          value="/u01/app/oracle/product/inf10120/lib"
        />
      </environment>
      ...
```

JDBC Drivers for non-Oracle Databases

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 DB2, Sybase, Informix, and SQLServer. If you want to use DataDirect drivers with OC4J, then add corresponding entries for each database in the `data-sources.xml` file.

Installing and Setting Up DataDirect JDBC Drivers

Install the DataDirect JDBC drivers as described in the *DataDirect Connect for JDBC User's Guide and Reference*

After you have installed the drivers, follow these instructions to set them up.

Note: In the following instructions, note these definitions:

- `OC4J_INSTALL`: In a standalone OC4J environment, the directory into which you unzip the file `oc4j_extended.zip`. In an Oracle Application Server, `OC4J_INSTALL` is `ORACLE_HOME`.
- In both a standalone OC4J environment and an Oracle Application Server, `DDJD_INSTALL` is the directory into which you unzip the content of the DataDirect JDBC drivers.
- In a standalone OC4J environment, `INSTANCE_NAME` is `home`.
- In an Oracle Application Server, `INSTANCE_NAME` is the OC4J instance into which you install the DataDirect JDBC drivers.

1. Unzip the content of the DataDirect JDBC drivers to the directory *`DDJD_INSTALL`*.
2. Create the directory *`OC4J_INSTALL`*`/j2ee/`*`INSTANCE_NAME`*`/applib` if it does not already exist.
3. Copy the DataDirect JDBC drivers in *`DDJD_INSTALL`*`/lib` to the *`OC4J_INSTALL`*`/j2ee/`*`INSTANCE_NAME`*`/applib` directory.
4. Verify that the file `application.xml` contains a library entry that references the `j2ee/home/applib` location, as follows:

```
<library path="../../INSTANCE_NAME/applib" />
```

5. Add data sources to the file `data-source.xml` as described in "Example DataDirect Data Source Entries" on page 4-39.

Example DataDirect Data Source Entries

This section shows an example data source entry for each of the following non-Oracle databases:

- DataDirect DB2
- DataDirect Sybase
- DataDirect Informix

You can also use vendor-specific data sources in the class attribute directly. That is, it is not necessary to use an OC4J-specific data source in the class attribute.

For more detailed information, refer to the *DataDirect Connect for JDBC User's Guide and Reference.*

Note: OC4J does not work with non-Oracle data sources in the non-emulated case. That is, you cannot use a non-Oracle data source in a two-phase commit transaction.

Additional data source configuration examples are shown in the article *Data Source Configuration in Oracle Application Server 10g* available at `http://www.oracle.com/technology/tech/java/newsletter/articles/oc4j_datasource_config.html`

DataDirect DB2

Notes: When using a DataDirect JDBC driver to connect to DB2, the following constraints apply:

- This item is exceptional behavior:

  If you use `com.oracle.ias.jdbcx.db2.DB2DataSource` as the connection factory class, the `<url>` is a required element but the values passed in the url attributes are ignored. For this reason, setting the url alone is enough.

  In order for the data source to work properly, you must set the `serverName`, `portNumber`, and `databaseName` in `<property>` elements, even though the `<property>` tag is "technically" optional.

- This item is normal behavior:

  If you use `com.oracle.ias.jdbc.db2.DB2Driver` as the connection factory class, then it is not necessary to set the `serverName`, `portNumber`, and `databaseName` in <property> elements. You can set the `serverName`, `portNumber`, and `databaseName` in the url attributes. The url is read completely and the values passed in the url attributes are read.

```
<managed-data-source
   name="db2"
   jndi-name="jdbc/db2"
   connection-pool-name="db2 Connection Pool"/>

<connection-pool name="db2 Connection Pool">
    <connection-factory
    factory-class="com.oracle.ias.jdbcx.db2.DB2DataSource"
    user="user1"
    password="user1"

 url="jdbc:oracle:db2://localhost:50000;DatabaseName=sample;PackageName=JDBCPKG">
    <property name="databaseName" value="sample"/>
    <property name="packageName" value="JDBCPKG"/>
    <property name="serverName" value="localhost"/>
    <property name="portNumber" value="50000"/>
    <xa-recovery-config>
        <password-credential>
        <username>system</username>
        <password>manager</password>
         </password-credential>
    </xa-recovery-config>
    </connection-factory>
</connection-pool>
```

DataDirect Sybase

```
<managed-data-source
   name="Sybase"
   jndi-name="jdbc/Sybase"
   connection-pool-name="Sybase Connection Pool"/>

<connection-pool name=" Sybase Connection Pool">
    <connection-factory
    factory-class="com.oracle.ias.jdbcx.sybase.SybaseDataSource"
    user="user1"
    password="password"
    url="jdbc:oracle:sybase://localhost:4101">
    <property name="serverName" value="localhost"/>
    <property name="portNumber" value="4101"/>
    <xa-recovery-config>
        <password-credential>
        <username>system</username>
        <password>manager</password>
        </password-credential>
    </xa-recovery-config>
    </connection-factory>
</connection-pool>
```

DataDirect Informix

```
<managed-data-source
   name="Informix"
   jndi-name="jdbc/Informix"
   connection-pool-name="Informix Connection Pool"/>

<connection-pool name=" Informix Connection Pool">
    <connection-factory
    factory-class="com.oracle.ias.jdbc.informix.InformixDriver"
    user="user1"
    password="password"


url="jdbc:oracle:informix://localhost:3900;informixServer=gtw93;DatabaseName=gatew
aydb">

    <xa-recovery-config>
        <password-credential>
        <username>userid</username>
        <password>pword</password>
        </password-credential>
    </xa-recovery-config>
    </connection-factory>
</connection-pool>
```

DataDirect SQLServer

This section shows a SQLServer managed data source example and a SQLServer native source example.

SQLServer Managed Data Source

```
  <connection-pool name="ConnectionSqlserver"
                 max-connections="20"
                 min-connections="1">
        <connection-factory
factory-class="com.oracle.ias.jdbcx.sqlserver.SQLServerDataSource"
                 user="msuser"
                 password="mspass"

url="jdbc:oracle:sqlserver://myserver\\myinstance;User=msuser;Password=mspass" >
             <property name="serverName" value="myserver\\myinstance"/>
             <xa-recovery-config>
                  <password-credential>
                       <username>msuser</username>
                       <password>mspass</password>
                  </password-credential>
             </xa-recovery-config>
        </connection-factory>
    </connection-pool>
.
    <managed-data-source connection-pool-name="ConnectionSqlserver"
                         jndi-name="jdbc/mysqlserver"
                         name="mysqlserver" />
```

SQLServer Native Data Source

```
    <native-data-source
           jndi-name="jdbc/mysqlserver"
           name="mysqlserver"

data-source-class="com.oracle.ias.jdbcx.sqlserver.SQLServerDataSource"
```

```
           user="msuser"
           password="mspass"

url="jdbc:oracle:sqlserver://myserver\\myinstance;User=msuser;Password=mspass" >
             <property name="serverName" value="myserver\\myinstance"/>
    </native-data-source>
```

Additional Data Source Configuration Examples

The following additional data source configuration examples show various permutations of data source type, connection factory type, and other variables. These examples are taken from the article *Data Source Configuration in Oracle Application Server 10g* available at

http://www.oracle.com/technology/tech/java/newsletter/articles/oc4j_datasource_config.html

Native Data Source - Oracle JDBC to Oracle Database

```
<native-data-source
  name="nativeDataSource"
  jndi-name="jdbc/nativeDS"
  description="Native DataSource"
  data-source-class="oracle.jdbc.pool.OracleDataSource"
  user="scott"
  password="tiger"
  url="jdbc:oracle:thin:@//dbhost:1521/dbservicename">
</native-data-source>
```

Native Data Source - DataDirect JDBC to DB2 UDB

```
<native-data-source
  name="nativeDataSource"
  jndi-name="jdbc/nativeDS"
  description="Native DataSource"
  data-source-class="com.ddtek.jdbcx.db2.DB2DataSource"
  user="frank"
  password="frankpw"
  url="jdbc:datadirect:db2://server_name:50000;DatabaseName=your_database">
</native-data-source>
```

Native Data Source - DB2 Universal JDBC to DB2 UDB

```
<native-data-source
  name="nativeDataSource"
  jndi-name="jdbc/nativeDS"
  description="Native DataSource"
  data-source-class="com.ibm.db2.jcc.DB2DataSource"
  user="db2adm"
  password="db2admpwd"

url="jdbc:db2://sysmvs1.stl.ibm.com:5021/dbname:user=db2adm;password=db2admpwd;">
</native-data-source>
```

Managed Data Source Using an XADataSource Connection Factory

```
<managed-data-source
  jndi-name="jdbc/ManagedDS"
  description="Managed DataSource"
  connection-pool-name="myConnectionPool"/>

<connection-pool
  name="myConnectionPool"
  min-connections="10"
  max-connections="30"
  inactivity-timeout="30">
    <connection-factory
        factory-class="oracle.jdbc.xa.client.OracleXADataSource"
        user="scott"
        password="tiger"
        url="jdbc:oracle:thin:@//dbhost:1521/dbservicename"/>
    </connection-factory>
</connection-pool>
```

Managed Data Source Using a DataSource Connection Factory

```
<managed-data-source
  jndi-name="jdbc/ManagedDS"
  description="Managed DataSource">
  connection-pool-name="myConnectionPool"/>

<connection-pool
  name="myConnectionPool"
  min-connections="10"
  max-connections="30"
  inactivity-timeout="30">
    <connection-factory
        factory-class="oracle.jdbc.pool.OracleDataSource"
        user="scott"
        password="tiger"
        url="jdbc:oracle:thin:@//dbhost:1521/dbservicename"/>
    </connection-factory>
</connection-pool>
```

Managed Data Source Using a Driver Connection Factory

```
<managed-data-source
  jndi-name="jdbc/ManagedDS"
  description="Managed DataSource">
  connection-pool-name="myConnectionPool"/>

<connection-pool
  name="myConnectionPool"
  min-connections="10"
  max-connections="30"
  inactivity-timeout="30">
    <connection-factory
        factory-class="oracle.jdbc.OracleDriver"
        user="scott"
        password="tiger"
        url="jdbc:oracle:thin:@//dbhost:1521/dbservicename"/>
    </connection-factory>
```

```
</connection-pool>
```

Legacy Configuration

The following points are pertinent for the configuration of versions prior to 10.1.3:

- There are two syntaxes for `data-sources.xml`.
  - One is the new 10.1.3 syntax, which follows the `http://xmlns.oracle.com/oracleas/schema/data-sources-10_1.xsd` schema.
  - The other is the pre-10.1.3 syntax, which follows the `http://xmlns.oracle.com/ias/dtds/data-sources-9_04.dtd`.

  Both syntaxes are legal but the new syntax is preferred.
- When `data-sources.xml` is persisted due to a change made in the Application Server Control Console the syntax is always written in the new 10.1.3 format.
- The `data-sources.xml` file can be converted from the old legacy syntax to the new 10.1.3 syntax explicitly via the `admin.jar` utility using the `-convertDataSourceConfiguration` option.

5 OC4J Transaction Support

This chapter discusses Transaction Support in Oracle Containers for J2EE (OC4J). It contains the following topics:

Transaction Management Tasks

This chapter describes the following transaction management tasks:

Configuring the OC4J Transaction Manager

Configuring the In-Database Transaction Coordinator

Managing Transaction Demarcation. See Demarcating Transactions

Manual Commit and Rollback Operations

Monitoring the OC4J Transaction Manager

What's New in Transaction Support for OC4J 10.1.3

The following OC4J Transaction Support features and behaviors are new for this release:

- Middle-Tier Two-Phase Commit (2PC) Coordinator that supports all XA-compatible resources, not just those from Oracle. This feature is referred to as a "heterogeneous middle-tier coordinator".
- New configuration
- Transaction administration
- Transaction Propagation between OC4J Processes over ORMI
- J2CA 1.5 transaction inflow support
- **Deprecated**

  The following item is deprecated in this release and will be desupported in the future:

  - The use of the in-database transaction coordinator by OC4J is deprecated as of release 10.1.3. Oracle recommends that the middle-tier transaction coordinator be used going forward. For additional information, see Configuring the In-Database Transaction Coordinator on page 5-15.

- **No Longer Supported**

  The following items are no longer supported in this release:

  - Use of `transaction-timeout` in `server.xml`. Moved to `transaction-manager.xml`.

Before Using OC4J in Production

Before using OC4J in production, Oracle recommends that the following tasks be performed:

- Configure the mid-tier coordinator to use a persistent store. The persistent store (and thereby logging) is disabled by default. See "transaction-manager.xml" on page 5-12.
- Change the default transaction recovery password. See "Recovery" on page 5-23

Additional Documentation

The How-To documents at the following site provide information about OC4J 10g Release 3 (10.1.3) features, including feature overviews and code excerpts relevant to the feature.

`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`

Introduction to OC4J Transaction Support

The Java Transaction API (JTA) is a specification developed by Sun Microsystems to provide support for global (distributed) transactions in the J2EE environment. Global transactions combine multiple enterprise systems - such as databases and message queues - into a single unit of work.

The JTA maps the specifications based on the Open Group Distributed Transaction Processing model into the Java environment. The Open Group XA specification defines the interface between transaction managers and resource managers. The JTA specification defines the interface between applications and transaction managers.

The JTA specification is available at `http://java.sun.com/products/jta`

What Is a Transaction?

A transaction is a mechanism that ensures correct outcomes in a system undergoing state changes. A transaction allows a programmer to scope a number of state changes in a system into a single unit of work. JTA transactions consist of enlisting resources and demarcating transactions. The changes that are scoped by the transaction may be either committed or rolled back.

Typically, a transaction is started by an application, the application performs some work against shared resources (such as one or more database systems), and the application commits the transaction. In the case of container-managed transactions, the application server starts the transaction.

For more about transaction processing in J2EE, you can refer to the following book:

Little, Maron, Pavlik: Java Transaction Processing: Design and Implementation, Prentice Hall, 2004

ACID

The transaction model that is supported by J2EE and most major information management systems are usually designed to preserve the ACID properties:

- **Atomic** - Either all changes scoped by the transaction are committed or all changes are rolled back.
- **Consistent** - The system moves from valid state to valid state.
- **Isolated** - The results of a transaction are not visible outside of the transaction until the transaction has been committed.
- **Durable** - The state changes scoped by the transaction are made permanent.

Note that the system infrastructure itself cannot maintain all of these properties. The consistency property requires that the application logic make valid changes to the system. For example, a transaction whose logic inserts an identifier representing a planet instead of a person into a human resource management database would create an inconsistent outcome - though the application server and database may have no way of knowing that the planet name was a meaningless value.

Note also that ACID guarantees place a burden on infrastructure and applications. If you need to reduce the overhead of ACID, design the work to use only one local resource, if possible, and not to use two-phase commit.

Middle-Tier Two-Phase Commit (2PC) Coordinator

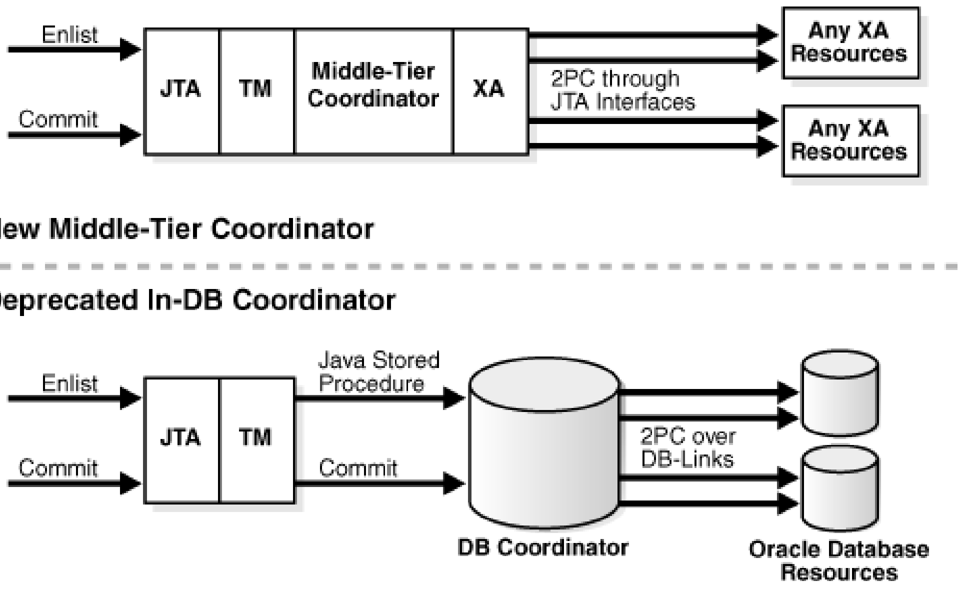
In this release, transaction coordination functionality is now located in OC4J, replacing in-database coordination, which is now deprecated. Also, the middle-tier coordinator is now "heterogeneous", meaning that it supports all XA-compatible resources, not just those from Oracle.

The middle tier coordinator provides the following features:

- Supports any XA compliant resource
- Supports interpositioning and transaction inflow
- Last Resource Commit Optimization
- Recovery Logging

Figure 5–1, "New Middle-Tier Coordinator vs. Deprecated In-Database Coordinator" shows the differences between the new middle tier coordinator and the deprecated in-database coordinator.

Figure 5–1 New Middle-Tier Coordinator vs. Deprecated In-Database Coordinator



Local and Global Transactions

The complexity of a transaction is determined by how many resources the application enlists within the transaction.

A **local transaction** involves only one resource and the transaction activity is scoped and coordinated locally to the resource itself. A local transaction uses the one-phase commit (1pc) protocol.

A **global transaction** (also called a distributed transaction) enlists more than one resource in the transaction. For example, a global transaction can be used to scope work on two databases. Transaction processing systems that support global transactions usually have several distinct logical components: a transaction factory, resource managers, coordinator and application. To achieve atomic outcomes in the global transaction, a termination protocol known as the two-phase commit (2pc) protocol is used.

The Two-Phase Commit Protocol

The two-phase commit (2pc) protocol is a mechanism to arrive at consensus among multiple participants in a global transaction. The protocol, as the name suggests, is divided into two phases. The first phase is often referred to as the voting phase. Each participant is asked to prepare the work in the transaction to be committed. If the participant is able to commit the work, it sends a message to the coordinator voting to commit.

During the voting, if ANY participant cannot prepare a transaction to commit, then all participants are instructed to roll back.

To guarantee the ACID properties, some transaction systems combine the two-phase commit protocol with the two-phase locking protocol. In practice, this means that no work may be performed in a transaction after the prepare phase has begun.

In order to provide atomic outcomes in the event of system failures, the two-phase commit protocol requires that the transaction manager log the transaction progress.

Using Multiple Resources

Resources are enlisted automatically by the application server when used within the scope of a transaction. However, to guarantee ACID properties in a two-phase commit transaction all participating resources must be XA compliant, with the exception of the last resource commit optimization.

Last Resource Commit

Last resource commit allows for a single non-XA-compliant resource to participate in an XA transaction. If more than one non-XA-compliant resource is enlisted in the transaction, then an exception is thrown from the enlistment attempt.

During the prepare phase, all XA- compliant resources are prepared. If all of the XA resources return OK from prepare, the coordinator performs a transfer of control to the non-XA compliant resource. If the non-XA compliant resource returns that it has committed, then the coordinator logs a commit decision, else a rollback decision is logged. The coordinator then notifies all of the XA resources of its decision.

Note: Although this optimization works correctly a great majority of the time, there is some risk in using the last resource commit optimization. If a failure occurs during the transfer of control to the non-XA compliant resource, the coordinator has no way of knowing whether the resource committed or rolled back. This could lead to non-atomic outcomes, which would be very hard to rectify.

Because the last resource commit optimization only allows a single one-phase commit resource to be enlisted in a given transaction, OC4J restricts the enlistment of the one-phase commit resource to the root OC4J process. If an attempt is made to enlist a one-phase commit resource on a subordinate OC4J process, then an exception is thrown indicating that the server was unable to enlist the resource. This restriction is necessary because of the overhead involved in ensuring that only one one-phase commit resource was enlisted in a global transaction tree if subordinates were allowed to enlist one-phase commit resources when using the last resource commit optimization.

Notes:

- It is possible to enlist more than one non-XA compliant resource in a global transaction if transaction logging is set to `"none"`, however, there are no ACID guarantees nor recovery in this case. Setting transaction logging is described in "transaction-manager.xml" on page 5-12.
- If the transaction manager is not using a persistent store, enlistment is allowed on the subordinate node.

To turn on last resource commit, enable transaction logging by configuring the transaction manager with a persistent store. The persistent store (and thereby logging) is disabled by default. See "transaction-manager.xml" on page 5-12.

In addition to allowing a single non-XA resource to participate in a global transaction, last resource commit can also be used as an optimization. By enlisting an XA-capable resource as a non-XA resource and using last resource commit, a gain in performance is achieved because the resource doesn't perform logging and a network call is eliminated because the coordinator would only make one call to the resource instead of two. Also, the resource would never be put in doubt, which would prevent resources from being locked. Although last resource commit can be used as a performance optimization, it is at the cost of guaranteed correctness.

Additional discussion of the last resource commit feature in OC4J can be found in the "Transaction Management" chapter of the *Oracle Containers for J2EE Resource Adapter Administrator's Guide*.

Resource Manager

A resource manager is responsible for managing shared resources. A common example of a resource manager is a relational database system.

Transaction Manager

A transaction manager combines the roles of transaction factory and coordinator. Applications communicate with the transaction manager to begin and end transactions and to enlist resources. When the application requests that a transaction be committed, the transaction manager will coordinate the two-phase commit protocol.

Heuristics

To achieve consensus, two-phase commit is a blocking protocol. This means that, if a coordinator fails before delivering the final phase messages, the participants must remain blocked, holding onto resources. Modern transaction systems add heuristics to two-phase commit, which allows such participants to make unilateral decisions about whether they will commit or rollback. If a participant makes a choice that turns out to be different from the one taken by other participants, then non-atomic behavior occurs. These decisions are generally made by administrative intervention. This is described in Manual Commit and Rollback Operations on page 5-18.

Synchronizations

Synchronizations are objects that are registered with a transaction and notified before and after running the two-phase commit protocol. For example, an application will often maintain it's own state and persist the cache to the resource only when necessary or just before completion and then release resources after completion.

Synchronizations are not available via the `UserTransaction` interface, so applications are not able to register synchronizations directly. This can only be done by the application server. In the case of CMP, the persistence layer handles this.

Information on EJBs is available in the *Oracle Containers for J2EE Enterprise JavaBeans Developer's Guide*.

Programming Models - Container-Managed and Bean-Managed Transactions

Enterprise Java Beans use JTA 1.0.1B for managing transactions through either container-managed transactions or bean-managed transactions.

Note: Other terms for "container-managed transactions" are "declarative transactions" and "CMT".

Other terms for "bean-managed transactions" are "programmatic transactions" and "BMT".

Container-Managed Transactions

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.

Bean-Managed Transactions

Bean-managed transactions are programmatically demarcated within your bean implementation. The transaction boundaries are completely controlled by the application.

Demarcating Transactions

Demarcating a transaction means to initiate and terminate the transaction.

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 based on the transaction attributes specified in the EJB deployment descriptor.

Container-managed transaction is available to all EJBs. However, the bean-managed transactions are available for session beans and message-driven beans (MDBs) only. In other words, entity beans, designed for data access, must use container-managed transaction demarcation. Session beans can use either model.

Note: Transactions cannot be demarcated from the application client container.

Specify the type of demarcation in the bean deployment descriptor. The following example shows a session bean that is declared as container-managed transactional by defining the `<transaction-type>` element as `Container`. To configure the bean for bean-managed transactional demarcation, define the `<transaction-type>` element as `Bean`.

Example: 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>
```

Demarcating Container-Managed Transactions

If you define your bean to use container-managed transactions (CMTs), then you must specify how the container manages the JTA transaction for this bean in the `<trans-attribute>` element in the bean deployment descriptor as shown in the following example. The following table lists and describes the transaction attribute settings.

Table 5–1 <trans-attribute> Element Settings

Setting	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. For message-driven beans (MDBs), this is the default.
`Required`	The bean must be involved in a transaction. If the invoker is involved in a transaction, then the bean uses the invoker's transaction. If the invoker is not involved in a transaction, then the container starts a new transaction for the bean. This is the default. For CMP 2.0 entity beans, this is the default.
`Supports`	Whatever transactional state that the invoker is involved in is used for the bean. If the invoker has begun a transaction, then the invoker's transaction context is used by the bean. If the invoker is not involved in a transaction, then neither is the bean. This is the default for all entity beans except CMP 2.0 entity beans and MDBs.
`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 `<trans-attribute>` setting 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`.

The following example shows the `<container-transaction>` portion of the EJB deployment descriptor. It demonstrates how this bean specifies the `RequiresNew` transaction attribute for all (`*`) methods of the `myEmployee` EJB.

Example: <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 these functions based on the transaction attribute that is specified in the deployment descriptor.

Demarcating Bean-Managed Transactions

Web components (JSP, servlets) and stateless and stateful session beans can use programmatic transaction demarcation. Entity beans cannot, and thus must use container-managed transaction demarcation.

Note: Client-side transaction demarcation is not supported in the application client container:
OC4J does not support 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.

If you declare the bean as bean-managed transactional (BMT) within the `<transaction-type>` element of the bean deployment descriptor, then the bean implementation must demarcate the start, commit, or rollback for the global transaction.

For bean-managed (programmatic) transaction demarcation, the bean developer can use the `UserTransaction` interface to demarcate global transactions or RM-specific methods to demarcate RM local transactions.

The Web component or bean writer must explicitly issue the `begin()`, `commit()`, and `rollback()` methods of the `UserTransaction` interface, as shown in the following example:

```
Context initCtx = new Initial Context();
```

```
ut = (UserTransaction)  initCtx.lookup("java:comp/UserTransaction");

ut.begin();
// Do work.
…
try {
    // Commit the transaction.
    ut.commit();
// Handle exceptions.  Should really catch specific exceptions, not Exception
} catch (Exception ex) { … }
```

Configuring the OC4J Transaction Manager

This section discusses the following topics:

- Configuring the Middle-Tier Transaction Manager in the Application Server Control Console and the JTA Resource MBean
- Configuring Middle-Tier OC4J Transaction Support in XML Files
- Configuring the In-Database Transaction Coordinator

Configuring the Middle-Tier Transaction Manager in the Application Server Control Console and the JTA Resource MBean

The primary tool for configuring the Transaction Manager is the Oracle Enterprise Manager 10*g* Application Server Control Console. The Application Server Control Console is the preferred method for configuring OC4J.

Path to the Transaction Manager page in the Application Server Control Console:

OC4J:Home > Administration tab > Task Name: Services > Transaction Manager: Go To Task

From the Transaction Manager page:

- The `Performance` tab presents OC4J Transaction Support statistics, as described in "Monitoring the OC4J Transaction Manager" on page 5-18.
- The `Transactions` tab presents the detail of all transactions that are currently running or being recovered and displays the details returned from the `CurrentTransactionDetail` attribute and allows for the administration of these transactions. The `CurrentTransactionDetail` attribute is available from the `JTAResource` MBean in the Application Server Control Console.
- The `Administration` tab presents the current OC4J Transaction Support configuration and allows for it to be altered.

In addition to the Transaction Manager page, the `JTAResource` MBean, accessible through the Application Server Control Console, provides configuration and management services for OC4J Transaction Support.

Path to the JTAResource MBean:

OC4J:Home > Administration tab > Task Name: JMX > System MBean Browser: Go To Task

The descriptions in the following tables refer to settings made on the Transaction Manager page, in the `JTAResource` MBean, and in the XML files. The XML files are discussed in "Configuring Middle-Tier OC4J Transaction Support in XML Files" on page 5-12.

Table 5–2 JTAResource MBean configureCoordinator Operation Parameters

Parameter	Description
`commitCoordinator`	The two-phase commit coordinator type, either `database` or `middle-tier`. The use of the in-database coordinator by OC4J is deprecated as of release 10.1.3. Oracle recommends that the middle tier coordinator be used going forward. For additional information, see Configuring the In-Database Transaction Coordinator on page 5-15.
`logType`	The log type, either `none`, `file`, or `database`. This setting applies only when `commitCoordinator` is set to `middle-tier`.
`logLocation`	The log location, either directory path for file or jndi location for database logging.
`userName`	The database username.
`password`	The database password.
`retryCount`	The number of times the coordinator will, when valid, retry commands to resource managers synchronously. Retries may occur, for example, if a resource manager returns `XA_RETRY`.

Coordinator Configuration Notes:

- Server restart is required for coordinator configuration changes to take effect and all transactions must be purged before shutdown.
- If there are any transactions that are not purged, the server will wait until they have been administratively resolved.
- If the server shutdown is forced while transactions are still in doubt, the configuration change will not be written to file.

Table 5–3 JTAResource MBean Attributes

Attribute	Description
`transactionTimeout`	The default transaction timeout in seconds
`maxConcurrentTransactions`	The maximum number of concurrent transactions in the server before System Exceptions will be thrown. (`-1` indicates unlimited.)
`transactionManagerConfigurationDetail`	Details of the transaction manager configuration

Note:

Changes to the `JTAResource` MBean attributes are persisted to the `transaction-manager.xml` configuration file and take effect immediately. A server restart is not required.

Configuring Middle-Tier OC4J Transaction Support in XML Files

The following configuration files are used to configure OC4J Transaction Support:

- server.xml
- transaction-manager.xml
- oc4j-ra.xml
- data-sources.xml

server.xml

The `server.xml` file must simply contain the following element for the transaction manager to be configured:

```
<transaction-manager-config path="./transaction-manager.xml"/>
```

Note: As of 10.1.3, all transaction manager configuration, such as the `transaction-timeout` attribute, is now contained in the `transaction-manager.xml` file and is no longer contained in the `server.xml` file.

For reference documentation of `server.xml`, see *Oracle Containers for J2EE Configuration and Administration Guide* Appendix B - Configuration Files Used in OC4J, Section - "Overview of the OC4J Server Configuration File (server.xml)".

transaction-manager.xml

The `transaction-manager.xml` file takes the following default form:

```
<?xml version="1.0" encoding="UTF-8"?>

<transaction-manager
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"


xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/transaction-manager-10_0.xsd
"
  transaction-timeout="30"
  max-concurrent-transactions="-1"
>
  <!-- transaction-timeout is in seconds and defaults to 30 -->
  <!-- a max-concurrent-transactions of -1 indicates unlimited concurrent transactions -->
  <commit-coordinator retry-count="4">
    <middle-tier>
      <!-- specify 'none' to turn off logging and increase performance, however,  note that
recovery is impossible       -->
       <log type="none"/>

       <!-- specify 'file' to log to flat file (use the 'location' attribute to override the
default directory and file-logging-performance attributes to override default settings)
       <log type="file">
         <file-logging-performance min-pool-size="40" max-open-files="256"
old-file-release-size="20" />
       </log>
       -->
       <!-- specify 'database' to log to the native-data-source specified by the 'location'
attribute (use database-logging-performance attributes to override default settings
       <log type="database" location="jdbc/logging">
         <identity user="system" password="manager"/>
         <database-logging-performance batch-create-interval="10" batch-state-interval="10"
batch-purge-interval="100" />
       </log>
       -->

     </middle-tier>

     <!-- specify the following database element instead of the middle-tier when using the in-db
coordinator
     <database location="jdbc/OracleDS">
       <identity user="system" password="manager"/>
     </database>
     -->

  </commit-coordinator>
</transaction-manager>
```

Notes on Middle-tier Coordinator Database Logging:

- The database must be Oracle.
- The schema located at

  `ORACLE_HOME/j2ee/home/database/j2ee/jta/oracle/2pc_jdbcstore.sq`
  `l`

  must be installed on the database.
- The data source used for logging must be a native data source and not a managed data source.
- The data source used for logging must be deployed to the default application.

Performance Settings

This section describes the performance settings for file store logging and for database store logging in the `transaction-manager.xml` file.

The following table lists and describes the settings for file store logging in the `transaction-manager.xml` file.

Table 5–4 File store logging attributes

Attribute	Description
`minPoolSize`	The number of files that will be pre-allocated to the pool during startup. Default is `40`. Optimal value is enough to cover the maximum number of concurrent requests.
`maxOpenFiles`	The maximum number of file channels that can remain open/active. When this number is exceeded, the oldest channels will be released until the xid is requested again. Default is `256`. Optimal value is enough to cover the maximum number of concurrent requests.

Table 5–4 (Cont.) File store logging attributes

Attribute	Description
`oldFileReleaseSize`	The number of the oldest file handles that will be closed when max-open-files has been exceeded. Default is `20`. Optimal value is a function of the odds of `maxOpenFiles` being exceeded, which should be avoided if at all possible or kept to a minimum otherwise.

The following table lists and describes the settings for database store logging in the `transaction-manager.xml` file.

Table 5–5 Database store logging attributes

Attribute	Description
`batchCreateInterval`	The time in milliseconds between each batch write of transactions. Default is `10`. Optimal value is small but relates to the cost of database call (such as network latency) and jvm heap size.
`batchStateInterval`	The time in milliseconds between each batch write of transaction state changes. Default is `10`. Optimal value is small but relates to the cost of database call (such network latency) and jvm heap size
`batchPurgeInterval`	The time in milliseconds between each batch purge of completed transactions. Default is `100`. Optimal value is large but relates to the cost of database call (such as network latency) and jvm heap size

oc4j-ra.xml

`oc4j-ra.xml` specifies XA and recovery information necessary for JMS or any other connectors to participate in 2pc transactions and recovery

An `XAConnectionFactory` must be used in order to participate in two-phase commit processing and the `connection-factory` must define an `xa-recovery-config` element if the runtime user does not have permission to conduct the necessary `XAResource.recover` call.

For more information, see the "Transaction Management" chapter and the "OC4J Resource Adapter Configuration Files" appendix in the *Oracle Containers for J2EE Resource Adapter Administrator's Guide*.

The `xa-recovery-config` element takes the following form:

```
 <connection-factory>
...
 <xa-recovery-config>
   <password-credential>
   <username>adapter_admin_user</username>
   <password>adapter_admin_pw</password>
   </password-credential>
 </xa-recovery-config>
...
 </connection-factory>
```

The following list shows examples of vendor-specific permissions. Those relating to RDBMS resource managers are specified in the `data-sources.xml`.

- Oracle: `select` privileges on `DBA_PENDING_TRANSACTIONS` as well as `execute` privileges on `sys.dbms` system in version 9.2 and later of the RDBMS
- DB2: `sysadmin` role
- MSSQL: `execute` privileges on XA stored procedures
- MQSeries: `sysadmin` role

data-sources.xml

The `data-source.xml` file is used to specify recovery information necessary for data-sources to participate in 2pc transactions and recovery if the runtime user does not have XAResource.recover privileges.

```
//
<connection-pool
 name="cmt Connection Pool">
 min-connections="10"
 max-connections="30"
 inactivity-timeout="30"
 <connection-factory   factory-class="oracle.jdbc.xa.client.OracleXADataSource"
   user="scott"
   password="tiger"
   url="jdbc:oracle:thin:@localhost:1521:ORCL">

 <xa-recovery-config>
   <password-credential>
   <username>system</username>
   <password>manager</password>
   </password-credential>
 </xa-recovery-config>
 </connection-factory>
 </connection-pool>

<managed-data-source connection-pool-name="cmt Connection Pool"
           jndi-name="jdbc/cmt"
           name="cmt"/>
```

Note: Data sources must be designated as managed data sources in order to participate in OC4J global transactions.

Configuring the In-Database Transaction Coordinator

This section discusses requirements for using the in-database two-phase commit coordinator.

Note: The use of the in-database two-phase commit coordinator by OC4J is deprecated as of release 10.1.3. Oracle recommends that the middle tier coordinator be used going forward.

You must configure the in-database two-phase commit engine with the following:

- Fully-qualified database links from the coordinating database 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.

Note: For in-database two-phase commit (2pc) functionality, use Oracle Database version 9.2.0.4 or later. The in-database two-phase commit (2pc) function is not available with Oracle Database version 9.2 or earlier.

Notes on In-Database Coordinator Configuration:

- The database used must be Oracle.
- The data source specified as the coordinator must be a native data source and not a managed data source.
- The in-database coordinator cannot be used within propagated transactions (neither as the root nor interposed coordinator) and does not provide a last resource commit optimization]
- The data source specified as the coordinator must be deployed to the default application.

Designate and configure an eligible Oracle database as the two-phase commit engine as follows:

1. Define a managed data source in the JDBC Resources page of the Application Server Control Console or in the `data-sources.xml` file.

   The following example shows the definition in the `data-sources.xml` file.

```
 <connection-pool
      name="OracleCommitDS">
      min-connections="10"
      max-connections="30"
      inactivity-timeout="30"
      <connection-factory
          factory-class="oracle.jdbc.xa.client.OracleXADataSource"
          user="scott"
          password="tiger"
          url="jdbc:oracle:thin:@localhost:1521:ORCL">
      </connection-factory>
  </connection-pool>
<managed-data-source connection-pool-name="Example Connection Pool"
                          jndi-name="jdbc/OracleCommitDS"
                          name="OracleCommitDS"/>
```

2. Refer to the two-phase commit engine DataSource in the `transaction-manager.xml` as follows:

```
<database location="jdbc/OracleCommitDS">
    <identity user-name="COORDUSR" password="COORDPW"/>
</database>
```

The `identity` element is used only if it is necessary to override the `user` and `password` specified in the managed data source.

For more on `transaction-manager.xml`, see "transaction-manager.xml" on page 5-12.

3. Create a user and grant the appropriate permissions on all databases involved in the transaction.

   - Create the user on the two-phase commit engine that facilitates the transaction.
   - The user opens a session from the two-phase commit engine to each of the involved databases.
   - Grant the CONNECT, RESOURCE, CREATE SESSION privileges for the user to connect to each of these databases. The FORCE ANY TRANSACTION privilege allows the user to commit or roll back the transaction.

   For example, if the user that facilitates the transaction is COORDUSR, then the following example shows the operations 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;
   ```

4. Configure fully-qualified public database links (using the `CREATE PUBLIC DATABASE LINK` command) from the two-phase commit engine to each database that is to be involved in the global transaction. These links are necessary for the two-phase commit engine to communicate with each database at the end of the transaction. These links enable the `COORDUSR` to connect to all participating databases.

5. For each managed data source participating in the transaction, add the additional property of the fully-qualified-database link from the two-phase commit engine to this database.

   Add the property in the Connection Pool page of the Application Server Control Console or in the `data-sources.xml` file.

   The following example shows the definition in the `data-sources.xml` file.

   ```
    <connection-pool
        name="OracleDS1">
        min-connections="10"
        max-connections="30"
        inactivity-timeout="30"
        <connection-factory
            factory-class="oracle.jdbc.xa.client.OracleXADataSource"
            user="scott"
            password="tiger"
            url="jdbc:oracle:thin:@host1:1521:ORCL">
         <property name="dblink"
    value="DBLINK1.REGRESS.RDBMS.DEV.US.OLE.COM"/>
            </connection-factory>
     </connection-pool>
   <managed-data-source connection-pool-name="Example Connection Pool"
   ```

```
                         jndi-name="jdbc/OracleDS1"
                         name="OracleDS1"/>
 <connection-pool
      name="OracleDS2">
      min-connections="10"
      max-connections="30"
      inactivity-timeout="30"
      <connection-factory
          factory-class="oracle.jdbc.xa.client.OracleXADataSource"
          user="scott"
          password="tiger"
          url="jdbc:oracle:thin:@host2:1521:ORCL">
       <property name="dblink"
 value="DBLINK2.REGRESS.RDBMS.DEV.US.OLE.COM"/>
      </connection-factory>
 </connection-pool>
<managed-data-source connection-pool-name="Example Connection Pool"
                         jndi-name="jdbc/OracleDS2"
                         name="OracleDS2"/>
```

Managing the OC4J Transaction Manager

This section discusses the following operations that can be performed during run time:

- Manual Commit and Rollback Operations
- Monitoring the OC4J Transaction Manager
- Managing OC4J Transaction Manager Recovery

Manual Commit and Rollback Operations

Path to the JTAResource MBean:

OC4J:Home > Administration tab > Task Name: JMX > System MBean Browser: Go To Task

The following operations of the `JTAResource` MBean, which is accessible through the Application Server Control Console, can be used to force the outcome of a current transaction:

Operation	Description
`heuristicCommit`	Make an autonomous commit decision for an in-doubt transaction
`heuristicRollback`	Make an autonomous rollback decision for an in-doubt transaction
`setRollbackOnly`	Make an autonomous setRollbackOnly decision for an active transaction

Monitoring the OC4J Transaction Manager

This section discusses the following features, which are available through the JTAResource MBean:

- OC4J Transaction Support Statistics
- Event Notifications

OC4J Transaction Support Statistics

The statistics listed in the following table are provided by the `JTAResource` MBean, which is accessible through the Application Server Control Console.

The attributes listed in the following table are provided by the `JTAResource` MBean.

The following `JTAResource` MBean operations are related to statistics.

Event Notifications

You can add JMX event notifications that are fired when specified OC4J Transaction CountStatistics exceed given thresholds and again at the specified count intervals.

Use the following procedure to add a Threshold Event:

1. Drill down to the `JTAResource` MBean.
2. Select the `addThresholdEvent` operation and enter the following parameters:
   - `statName` - the name of a valid OC4J Transaction Count Statistic to monitor
   - `threshold` - the count at which to broadcast the event
   - `repeatNotificationInterval` - the interval count at which to broadcast subsequent events

To remove a threshold event, select the `removeThreshold` operation of the MBean and enter the name of the statistic as the argument.

You can subscribe to notification broadcasts on the Notification Subscriptions page of the Application Server Control Console.

Path to the Notification Subscriptions Page:

OC4J:Home > Administration tab > Task Name: JMX > Notification Subscriptions

The notifications received can then be viewed on the Notifications Received page of the Application Server Control Console.

Path to the Notifications Received Page:

OC4J:Home > Administration tab > Task Name: JMX > Notification Received

Managing OC4J Transaction Manager Recovery

Managing Recovery

In the normal case, for example where either OC4J or a resource participating in a 2PC transaction crashes and is subsequently brought back up, recovery occurs automatically, assuming proper configuration.

In the OPMN environment, if an OC4J instance crashes, then OPMN starts a new instance, which uses the crashed instance's logs.

In a standalone environment, if an OC4J instance crashes and cannot or should not be brought back for some reason, then the logs can be migrated to another OC4J instance.

The process for doing so depends on whether the instance is a root transaction manager, interposed transaction manager, or both and whether the file or database logging mechanism is used.

For a root transaction manager using the file store, you must either change the log location configuration of the new OC4J instances to the location of the crashed one or manually move the logs to the new OC4J instance log location. This must be done while the destination server is offline or shut down.

For an interposed transaction manager using the file store, the process is the same, however, any parent transaction managers of this interposed transaction manager must now update their references to this new location. This is possible using the `-updateTransactionLogs` offline admin command.

For a root transaction manager using the database store, the logs are migrated by updating the instance field of the `oc4j_transaction` table using the `-updateTransactionLogs` offline admin command. If the new OC4J instance is to use a completely new database, then the log file must be exported/imported as well.

For an interposed transaction manager using the database store, the process is the same, however, any parent transaction managers of this interposed transaction manager must now update their references to this new location. This is conducted using the `-updateTransactionLogs` offline admin command.

Two offline admin commands are available via the `admin.jar`:

- `-analyzeTransactionLogs`
- `-updateTransactionLogs`

Note: These notes apply to the context of standalone file migration, but they can be useful in certain situation in the opmn environment as well.

The `-analyzeTransactionLogs` command provides offline analysis of transaction log files. Do not use this utility if OC4J is running (use the Application Server Control Console instead). Arguments are:

Argument	Description
`-logType ["file" | "database"]`	The store type.
`-location [location]`	The location of the store. The directory for file logging. The connection url for database logging.
`-username [username]`	Applies only to database logging.
`-password [password]`	Applies only to database logging.

The `-updateTransactionLogs` command provides offline updates of transaction log files. Do not use this utility if OC4J is running (use the Application Server Control Console instead). Arguments are:

Argument	Description
`-logType ["file" | "database"]`	The store type.

Argument	Description
`-location [location]`	The location of the store. The directory for file logging. The connection url for database logging.
`-username [username]`	Applies only to database logging.
`-password [password]`	Applies only to database logging.
`-instanceId [old instance id] [new instanceid]`	The OC4J instance id associated with this log.
`-branchLocation [old branch location] [new branch location]`	The location of the branch, generally the jndi location prefixed by the application name.
`-branchArg [old branch arg] [new branch arg]`	The argument for the branch. For example, the container-managed sign-on username

Transaction Propagation between OC4J Processes over ORMI

This section discusses transaction propagation.

How Does Transaction Propagation Work?

Transaction context propagation makes it possible for multiple OC4J instances to participate in a single global transaction. Multiple OC4J instances need to participate in the same transaction if an OC4J instance makes a remote call into another OC4J instance in the scope of an existing transaction, assuming the EJB semantics for the method support scoping work in a client's transaction. An example of this is a servlet in OC4J instance:

- First, obtaining a reference to an EJB residing in OC4J instance.
- Then, starting a transaction and making a method call on the remote EJB in the scope of the transaction.

When multiple OC4J instances participate in a single transaction, all work done by the participating OC4J instances as part of the global transaction is guaranteed to be atomic.

Note: Transaction context propagation and subject propagation is supported between OC4J 10.1.3 instances only. These features are not supported between OC4J instances earlier than 10.1.3 or between an OC4J 10.1.3 instance and an earlier OC4J instance.

When a remote method invocation is made between OC4J instances, the requesting OC4J instance must create a transaction context that represents the current transaction and implicitly flow the context with the remote call over ORMI. This allows the OC4J instance that receives the remote method invocation to associate work done as part of the request with the transaction that is represented by the transaction context. The remote OC4J instance gets enlisted as a resource with the requesting OC4J instance, which creates a tree of OC4J instances in which the original OC4J instance is the root transaction coordinator and the child OC4J instances are nodes. It is possible for an OC4J instance to be both a parent and a child. In this case, the instance would act as a resource to its parent and as a coordinator to its children. This is called interpositioning. For more on ORMI, see Chapter 6, "Using Remote Method Invocation in OC4J".

When the root coordinator commits the transaction, it runs the two-phase commit (2pc) protocol and sends completion calls to all of its registered resources, including enlisted OC4J instances. The root coordinator is responsible for driving transaction completion using the two-phase commit (2pc) protocol, while the subordinate OC4J instances merely act as resources in the transaction. It is possible that the root coordinator is not an OC4J instance but instead is an external coordinator as would be the case when OC4J imports a transaction via the J2CA 1.5 Transaction Inflow contract. When an OC4J instance is enlisted as a resource, it behaves like any other enlisted resource and is only responsible for preparing and committing its branch of the transaction. When an OC4J instance receives a call to prepare, it attempts to prepare all of its enlisted resources and returns the result. Likewise, when a commit call is received, the OC4J instance calls commit on all of its enlisted resources and returns the result to its parent.

For more information on the Transaction Inflow Contract, see the "Using RAs for Inbound Connections" chapter of the *Oracle Containers for J2EE Resource Adapter Administrator's Guide*.

By propagating transactions between OC4J instances over ORMI, transactions can span more than one OC4J instance.

The How-To document "How-To Propagate a transaction context between OC4J instances" and the ZIP file at the following site contains information and an example about transaction context propagation between OC4J instances:

`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/how-to-transaction-propagation/doc/how-to-transaction-propagation.html`

Configuring Transaction Propagation

This section describes transaction propagation configuration issues.

Enabling/Disabling

Note: In most cases, transaction propagation is never disabled. Disabling transaction propagation is only done in exceptional circumstances such as troubleshooting. Exercise caution when disabling transaction propagation.

Transaction propagation is enabled by default in OC4J. To disable transaction propagation, set the system property `-Drmi.disablePropagation=true`. By setting this flag, all context propagation is disabled, which includes transaction and subject propagation. Enabling or disabling of transaction propagation affects all applications deployed to the server. Finer level granularity is not available. Great care should be taken when disabling transaction propagation. It is imperative that all OC4J processes that are involved in a request be configured identically in this regard. This means that all OC4J processes involved in a request must be enabled or that all of the OC4J processes must be disabled. If there is a mix of configurations, meaning that some OC4J processes in a request are enabled and that some are disabled, unexpected behavior may result. Also, if transaction propagation is disabled, then multiple OC4J processes cannot participate in a single global transaction. Only after fully understanding all of the consequences related to disabling transaction propagation should it be considered. Because of the possible adverse effects of disabling transaction propagation, it is recommended that this feature not be disabled.

Recovery

Because OC4J processes may act as transactional resources, they must be able to be recovered in the event of a failure. To facilitate transaction recovery, each OC4J process must have a configured password to be used by the transaction recovery manager at transaction recovery time. OC4J is shipped with a default password, which should be changed after install.

The recovery password is configured in the configuration file `jazn-data.xml`, which is in the `$J2EE_HOME/config` directory. To modify the transaction recovery password, change the credentials value for the user `JtaAdmin` in the `jazn-data.xml` file.

```
<user>
    <name>JtaAdmin</name>
    <display-name>JTA Recovery User</display-name>
    <description>Used to recover propagated OC4J transactions</description>
    <credentials>!newJtapassword</credentials>
</user>
```

Caution: Even if OC4J is configured to use a security service other than JAZN, such as OID, the transaction recovery password must still be configured in `jazn-data.xml`.

During transaction recovery, the recovery password in the actual security store for the OC4J process that is being recovered must match the password that was configured in `jazn-data.xml` during transaction processing. If the password does not match, the recovery manager cannot contact the subordinate OC4J process and therefore the recovery manager cannot complete recovery.

Logging

Although transaction propagation does not require any additional transaction logging configuration, it is important to note that because a transaction my span multiple OC4J processes, each OC4J process must be configured independently with regards to logging.

Transaction Propagation Constraints

This section discusses the following constraints that apply to transaction propagation functionality:

- Backwards Compatibility
- EJB Failover

Backwards Compatibility

OC4J 10.1.3 is the first version to support propagation, all previous versions did not. When an OC4J instance that supports transaction propagation makes a remote method invocation on a bean that is deployed on an older version of OC4J that does not support transaction propagation, no transaction context is propagated. For this reason, even though the caller may be in the scope of a transaction, the work done on the remote machine is not executed in the scope of the caller's transaction. The Application Deployer or Admin must understand the transactional requirements of an application prior to deployment if the application is to be deployed to various OC4J versions.

EJB Failover

When a transaction is propagated to an OC4J process, any failure that causes requests to that server to be failed over to a secondary server while in the scope of the propagated transaction will result in the transaction being rolled back.

EJB failover behavior is discussed in the *Oracle Containers for J2EE Enterprise JavaBeans Developer's Guide*.

Debugging and Troubleshooting

The following hints apply to debugging and troubleshooting:

- The `j2ee-logging.xml` file can be used to turn on debugging. Transaction issues are often related to the transaction participants and so turning on the J2CA, JDBC, and other logging will be of great contextual help.
- The logger name for the transaction manager is `oracle.j2ee.transaction`. The `j2ee-logging.xml` file is discussed in the "Logging in OC4J" chapter of the *Oracle Containers for J2EE Configuration and Administration Guide*.
- The Application Server Control Console provides information on all transactions both active and recovering.
- DMS must be enabled for all statistics.
- The transaction manager will throw an exception at the time that an attempt is made to enlist a resource in a global transaction that is currently participating in an outstanding JCA local transaction. This is new behavior in release 10.1.3. This behavior is mandated by section 7.8.3 of the J2CA 1.5 specification.

6 Using Remote Method Invocation in OC4J

This chapter includes the following topics:

What Is RMI?

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.

OC4J supports RMI over both the proprietary Oracle Remote Method Invocation (ORMI) protocol and over the standard Internet Inter-ORB Protocol (IIOP). Objects running within OC4J instances can invoke one another using RMI/ORMI. Objects can invoke one another across different J2EE containers—for example, between OC4J and BEA WebLogic servers—using RMI/IIOP.

Additional Documentation

- The How-To document "How-To Propagate a transaction context between OC4J instances" and the ZIP file at the following site contains information and an example about transaction context propagation between OC4J instances using ORMI:

  `http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/how-to-transaction-propagation/doc/how-to-transaction-propagation.html`

- The How-To documents at the following site provide additional information about OC4J 10g Release 3 (10.1.3) features, including feature overviews and code excerpts relevant to the feature.

  `http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`

Choosing RMI/ORMI or RMI/IIOP

When can you use RMI/ORMI versus RMI/IIOP?

- Use RMI/ORMI to invoke methods on remote objects within or across OC4J instances.

  See "Using Oracle Remote Method Invocation (RMI/ORMI)" on page 6-2 for details.

- Use RMI/IIOP to invoke methods on remote objects across OC4J and non-Oracle J2EE containers, such as BEA WebLogic.

  See "Switching from ORMI to IIOP Transport" on page 6-18 for details.

Using Oracle Remote Method Invocation (RMI/ORMI)

This section describes Oracle Containers for J2EE (OC4J) support for allowing objects—such as EJBs—to invoke one another across OC4J server instances using the proprietary Remote Method Invocation (RMI)/Oracle RMI (ORMI) protocol.

This section covers the following topics:

- Introducing RMI/ORMI
- Configuring RMI in a Standalone OC4J Installation
- Configuring RMI in an Oracle Application Server Environment
- Client-Side Requirements to Use RMI/ORMI

Introducing RMI/ORMI

The Oracle Remote Method Invocation (ORMI) is Oracle's proprietary implementation of the RMI protocol that is optimized for use with OC4J.

By default, EJBs within OC4J server instances exchange RMI calls over ORMI. Alternatively, you can convert an EJB to exchange RMI calls over IIOP, making it possible for EJBs to invoke one another across different EJB containers from different vendors, such as OC4J and BEA WebLogic. See "Switching from ORMI to IIOP Transport" on page 6-18 for more.

Note: For the OC4J 10*g* Release 3 (10.1.3) implementation, load balancing and failover are supported only for ORMI, not for IIOP.

Features of ORMI

ORMI is enhanced for OC4J and provides the following features:

- Increased RMI Message Throughput
- Enhanced Threading Support
- Co-Located Object Support

Increased RMI Message Throughput Using ORMI, OC4J can process at a very high transaction rate. This is reflected in Oracle's SpecJ Application Server benchmarks at `http://www.spec.org/`.

One way ORMI achieves this performance is by using messages that are much smaller than IIOP messages. Smaller messages take less bandwidth to send and receive, and less processing time to encode and decode.

ORMI message size is further reduced by optimizing how much state information is exchanged between client and server. Using ORMI, some state is cached on the server so that it does not need to be transmitted in every RMI call. This does not violate the RMI requirement to be stateless because in the event of a failover, the client code resends all the state information required by the new server.

Enhanced Threading Support ORMI is tightly coupled with the OC4J threading model to take full advantage of its queuing, pooling, and staging capabilities.

ORMI uses one thread per client. For multi-threaded clients, OC4J multiplexes each call through one connection. However, OC4J does not serialize them, so multiple threads do not block each other.

This feature ensures that each client (single-threaded or multi-threaded) has one connection to the remote server.

Co-Located Object Support For co-located objects, RMI/ORMI detects the co-located scenario and avoids the extra, unnecessary socket call.

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

Compatibility Patches for 9.0.4.x and 10.1.2.x

In order to use ORMI to invoke a method on a remote object when the invoking object and the invoked object are running on different OC4J versions, you must install a patch on the older version. This applies when the newer version is 10.1.3 and the older version is 9.0.4.x or 10.1.2.x. This applies both ways; that is when invoking from the older version to the newer version or when invoking from the newer version to the older version.

Some examples:

- A servlet running on OC4J 9.0.4.3 invoking a method on an EJB running on OC4J 10.1.3.
- An EJB running on OC4J 10.1.3 invoking a JMS object running on OC4J 10.12.0.2.
- A JSP running on OC4J 10.1.2 invoking a method on an EJB running on OC4J 10.1.3.

Note: Invoking between 9.0.4.x and 10.1.2.x does not require a patch; these versions are compatible already.

The patches can be downloaded from `http://metalink.oracle.com`.
The following table lists the older versions to be patched and the corresponding patch identifiers.

OC4J Version to be Patched	Patch Identifier
9.0.4.3	BUG 4712885
10.1.2.2	BUG 4712552
10.1.2.0.0	BUG 4742351
10.1.2.0.2	BUG 4740687

Configuring RMI in a Standalone OC4J Installation

In a standalone OC4J installation, you must specify RMI server data in the RMI configuration file, `rmi.xml`. You must also specify the location of this file in `server.xml`, the OC4J configuration file.

The `rmi.xml` file and the `server.xml` file are installed in `ORACLE_HOME/j2ee/home/config` by default

1. Specify the path to the RMI configuration file—`rmi.xml`—in the `<rmi-config>` element in `server.xml`, the OC4J server configuration file.

   The syntax is as follows:

   ```
   <rmi-config path="RMI_PATH" />
   ```

   The typical value for *`RMI_PATH`* is `./rmi.xml`.

2. Add an `<rmi-server>` element specifying the host, port, and user name and password to use to connect to (and accept connections from) remote RMI servers to the `rmi.xml` file on the OC4J instance. This file is installed in `ORACLE_HOME/j2ee/home/config` by default.

   For example:

   ```
   <rmi-server host="hostname" port="port">
   </rmi-server>
   ```

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

   - `host`: The host name or IP address from which the RMI server accepts RMI requests. If you omit this attribute, the RMI server accepts RMI requests from any host.
   - `port`: The port number on which the RMI server listens for RMI requests. In an OC4J standalone environment, if you omit this attribute, it defaults to `23791`.

3. Optionally configure the `<rmi-server>` element with one or more `<server>` elements that each specify a remote (point-to-point) RMI server that your application can contact over RMI.

   For example:

   ```
   <rmi-server host="hostname" port="port">
       <server host="serverhostname" username="username" port="serverport"
       password="password"/>
   </rmi-server>
   ```

   The `host` attribute is required; the remaining attributes are optional. Here are the user-replaceable attributes of the `server` element:

   - *`serverhostname`*: the host name or IP address on which the remote RMI server listens for RMI requests
   - *`username`*: the user name of a valid principal on the remote RMI server
   - *`serverport`*: the port number on which the remote RMI server listens for RMI requests
   - *`password`*: the password used by the principal *username*

Access Restrictions

ORMI and ORMIS enable you to restrict incoming IP access by defining ACL masks within `rmi.xml` using the `<access-mask>` element.

Access controls can either be exclusive or inclusive.

- In the exclusive mode, access must be explicitly granted to an IP address or host name. The default mode for the access mask `default="deny"` specifies that the access control is exclusive.
- In the inclusive mode, access is available to all and exceptions must be granted individually. The default mode for the access mask `default="allow"` specifies that the access control is inclusive.

The `<host-access>` and `<ip-access>` sub elements are used to specify exceptions to the default access mode.

For additional information, see Appendix A, "Web Module Administration"", in the *Oracle Containers for J2EE Servlet Developer's Guide*

An example of an exclusive mode configuration to allow only `localhost` and the `192.168.1.*` subnet is shown in the following example:

```
<rmi-server
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
   "http://xmlns.oracle.com/oracleas/schema/rmi-server-10_0.xsd"
  port="23791"
  ssl-port="23943"
  schema-major-version="10"
  schema-minor-version="0">

      <access-mask default="deny" >
         <host-access domain="localhost" mode="allow"/>
         <ip-access ip="192.168.1.0" netmask="255.255.255.0" mode="allow"/>
      </access-mask>

  <log>
    <file path="../log/rmi.log" />
  </log>

  <ssl-config
    keystore="../wallets/wallet-server-a/ewallet.p12"
    keystore-password="serverkey-a"
   />

   </rmi-server>
```

Client-Side Requirements to Use RMI/ORMI

This section lists the ZIP files that you use to install the Oracle and J2EE standard JAR files that enable EJB and JMS lookup using RMI/ORMI.

The following ZIP files are available from
`http://www.oracle.com/technology/software/products/ias/htdocs/utilsoft.html`

- To enable EJB lookup using ORMI, download and expand `oc4j_client.zip`.
- To enable other lookups, such as JMS, download and expand `oc4j_extended.zip` instead of `oc4j_client.zip`.

Once the appropriate ZIP file is expanded, make sure that `oc4jclient.jar` is included in the CLASSPATH.

The ZIP files contain all the JAR files required by the client. The JAR files contain the classes necessary for client interaction. You must only add `oc4jclient.jar` to your `CLASSPATH`, because all other JAR files required by the client are referenced in the `oc4jclient.jar` manifest classpath.

To enable EJB lookup, make sure that the following JAR files are included on the client side.

Table 6–1 Client-side JAR Files Required for EJB Lookup

JAR	ORACLE_HOME Path
`oc4jclient.jar`	`/j2ee/<instance>`
`ejb.jar`	`/j2ee/<instance>/lib`

To enable resource adapter lookup, include the following JAR files on the client side.

Table 6–2 Client-side JAR Files Required for JMS Connector Lookup

JAR	ORACLE_HOME Path
`oc4jclient.jar`	`/j2ee/<instance>`
`jta.jar`	`/j2ee/<instance>/lib`
`jms.jar`	`/j2ee/<instance>/lib`
`jndi.jar`	`/j2ee/<instance>/lib`
`javax77.jar`	`/j2ee/<instance>/lib`
`adminclient.jar`	`/j2ee/<instance>/lib`
`oc4j-internal.jar`	`/j2ee/<instance>/lib`
`connector.jar`	`/j2ee/<instance>/lib`
`jmxri.jar`	`/j2ee/<instance>/lib`
`jazncore.jar`	`/j2ee/<instance>`
`oc4j.jar`	`/j2ee/<instance>`

To enable internal OEMS JMS In-Memory and File-Based lookup, make sure that the JAR files listed in Table 6–3, " Client-side JAR Files Required for OEMS JMS In-Memory and File-Based Lookup" are included on the client side.

Table 6–3 Client-side JAR Files Required for OEMS JMS In-Memory and File-Based Lookup

JAR	ORACLE_HOME Path
`oc4jclient.jar`	`/j2ee/<instance>`
`jta.jar`	`/j2ee/<instance>/lib`
`jms.jar`	`/j2ee/<instance>/lib`
`jndi.jar`	`/j2ee/<instance>/lib`

Table 6–3 (Cont.) Client-side JAR Files Required for OEMS JMS In-Memory and File-Based Lookup

JAR	ORACLE_HOME Path
`javax77.jar`	`/j2ee/<instance>/lib`
`optic.jar` (Required only if the `opmn:ormi` prefix is used in Oracle Application Server environment.)	`/opmn/lib`

To enable internal OEMS JMS Database lookup directly from an application client, make sure that the JAR files listed in Table 6–4, " Client-side JAR Files Required for OEMS JMS Database Lookup" are included on the client side.

Table 6–4 Client-side JAR Files Required for OEMS JMS Database Lookup

JAR	ORACLE_HOME Path
`oc4jclient.jar`	`/j2ee/<instance>`
`ejb.jar`	`/j2ee/<instance>/lib`
`jta.jar`	`/j2ee/<instance>/lib`
`jms.jar`	`/j2ee/<instance>/lib`
`jndi.jar`	`/j2ee/<instance>/lib`
`javax77.jar`	`/j2ee/<instance>/lib`
`adminclient.jar`	`/j2ee/<instance>/lib`
`ojdbc14dms.jar`	`/j2ee/<instance>/../../oracle/jdbc/lib`
`dms.jar`	`/j2ee/<instance>/../../oracle/lib`
`bcel.jar`	`/j2ee/<instance>/lib`
`optic.jar` (Required only if the `opmn:ormi` prefix is used in Oracle Application Server environment.)	`/opmn/lib`

Configuring RMI in an Oracle Application Server Environment

In an Oracle Application Server environment, you must edit the `opmn.xml` file to specify the port range on which this local RMI server listens for RMI requests.

Note that manual changes to configuration files in an Oracle Application Server environment must be manually updated on each OC4J instance.

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="default-web-site" range="12501-12600" protocol="ajp" />
```

```
            <port id="rmi" range="12401-12500" />
            <port id="rmis" range="12801-12900" />
            <port id="jms" range="12601-12700" />
            <process-set id="default_group" numprocs="1"/>
        </process-type>
    </ias-component>
```

For more information on configuring the `opmn.xml` file, see the *Oracle Application Server Administrator's Guide*.

2. Apply changes by running the following `opmnctl` command:

```
opmnctl reload
```

Remote Object Lookup Using RMI/ORMI

To invoke methods on an object, you must first be able to locate the object.

- Setting JNDI Properties for RMI
- Configuring ORMI Request Load Balancing
- Example Lookups Using ORMI

Setting JNDI Properties for RMI

The following RMI/ORMI properties are specified in the client's `jndi.properties` file:

- `java.naming.provider.url` (see "Setting the Java Naming Provider URL" on page 6-8)
- `java.naming.factory.initial` (see "Specifying the Context Factory" on page 6-10)

Setting the Java Naming Provider URL

Use the following syntax to define one or more OC4J hosts as the value of `java.naming.provider.url`:

```
<protocol>://<host>:[<port>]:<oc4j_instance>/<appName>
```

For example, specify the following for an application running within Oracle Application Server:

```
java.naming.provider.url=opmn:ormi://myHost:home/ejbsamples
```

Table 6–5 describes arguments used in this syntax.

Note:

- For details on setting the `java.naming.provider.url` for applications that will utilize HTTP tunneling, see "Configuring ORMI Tunneling through HTTP" on page 6-13.
- For `java.naming.provider.url` values for applications using ORMI over SSL, see the CSIv2 chapter in the *Oracle Containers for J2EE Security Guide*.

Table 6–5 Naming Provider URL

Parameter	Description
`protocol`	■ Applications on Oracle Application Server: Use `opmn:ormi://` ■ Applications on standalone OC4J: Use `ormi://` ■ Applications that must interoperate with non-OC4J containers: Use `corbaname` (see "Specifying the corbaname URL" on page 6-21).
`host`	■ Applications on Oracle Application Server: Specify the name of the OPMN host as defined in the `opmn.xml` file. Although OPMN is often located on the same node as the OC4J instance, specify the host name in case it is located on another machine. ■ Applications on standalone OC4J: Specify the OC4J host name as defined by the `<rmi-server>` element `host` attribute in the `rmi.xml` file.
`port`	■ Applications on Oracle Application Server 10*g* Release 3 (10.1.3): Specify the OPMN `request` port. The `opmn` process will forward RMI requests to the RMI port that it selected for the appropriate OC4J instance (see "Specifying the opmn Request Port in Oracle Application Server 10g Release 3 (10.1.3)" on page 6-9). If omitted, the default OPMN `request` port value 6003 is used. ■ Applications on Oracle Application Server 10*g* Release 2 (10.1.2) and earlier: Specify the RMI port that `opmn` selected for your OC4J instance (see "Specifying the RMI Port in Oracle Application Server 10g Release 2 (10.1.2) And Earlier" on page 6-10). ■ Applications on standalone OC4J: Specify the port number defined in the `port` attribute of the `<rmi.server>` element in `rmi.xml`. ■ Applications that must interoperate with non-OC4J containers and use the `corbaname` prefix: See "Specifying the corbaname URL" on page 6-21 for information on what port to specify. The port is optional and is determined by the protocol. The ORMI protocol defaults to port `23791`. The ORMIS protocol defaults to port `23943`.
`oc4j_instance`	■ For Oracle Application Server applications: The name of the OC4J instance. The name of the default OC4J instance is `home`. ■ For standalone OC4J applications: This variable is not applicable.
`appName`	The name of your application.

Specifying the opmn Request Port in Oracle Application Server 10g Release 3 (10.1.3)

In Oracle Application Server 10*g* Release 3 (10.1.3), you can specify the port defined for the `request` attribute of the `notification-server` element's `port` element configured in the `opmn.xml` file (default: `6003`). When `opmn` receives an RMI request on its `request` port, it forwards the RMI request to the RMI port that it selected for the appropriate OC4J instance.

For example, consider the following `opmn.xml` file excerpt:

```
<notification-server>
    <port local="6100" remote="6200" request="6004"/>
    <log-file path="$OLE_HOME/opmn/logs/ons.log" level="4"
        rotation-size="1500000"/>
    <ssl enabled="true" wallet-file="$OLE_HOME/opmn/conf/ssl.wlt/default"/>
</notification-server>
```

In this example, the port defined for the `request` attribute of the `notification-server` element's `port` element is `6004`, so you would use `6004` as the port in your JNDI naming provider URL.

For an example of how this URL is used, see "OC4J in Oracle Application Server 10g Release 3 (10.1.3)" on page 6-12.

Specifying the RMI Port in Oracle Application Server 10g Release 2 (10.1.2) And Earlier

In releases prior to Oracle Application Server 10*g* Release 3 (10.1.3), you must specify the RMI port assigned by opmn for each OC4J instance. To get the assigned RMI port, use the following `opmnctl` command on the OC4J host:

```
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: server817.company.us.com
-------------------+--------------------+-------+ ... +------
ias-component      | process-type       |   pid | ... | ports
-------------------+--------------------+-------+ ... +------
OC4J               | home               |  2012 | ... | iiop:12701,jms:12601,rmi:12401
HTTP_Server        | HTTP_Server        | 28818 | ... | http2:7200,http1:7778,http:7200
```

In this example, opmn has selected port `12401` for RMI on the OC4J instance. Use this value as the port in your JNDI naming provider URL for this OC4J instance.

Specifying the Context Factory

The initial context factory creates the initial context class for the client.

Set the`java.naming.factory.initial` property to one of the following:

- `oracle.j2ee.naming.ApplicationClientInitialContextFactory`
- `com.evermind.server.ApplicationInitialContextFactory`
- `oracle.j2ee.rmi.RMIInitialContextFactory`

Note: The following initial context factories are deprecated:

- `com.evermind.server.ApplicationClientInitialContext Factory`
- `com.evermind.server.RMIInitialContextFactory`

The `ApplicationClientInitialContextFactory` is used when looking up remote objects from standalone application clients. It uses the `refs` and `ref-mappings` found in `application-client.xml` and `orion-application-client.xml`. It is the default initial context factory when the initial context is instantiated in a Java application.

The `RMIInitialContextFactory` is used when looking up remote objects between different OC4J server instances using the ORMI protocol.

The type of initial context factory that you use depends on what the client is:

- If the client is a pure Java client outside the OC4J container, then use the `ApplicationClientInitialContextFactory` class.
- If the client is an EJB or servlet client within the OC4J container, then use the `ApplicationInitialContextFactory` class. This is the default class; thus, each time you create a new `InitialContext` without specifying an initial context factory class, your client uses the `ApplicationInitialContextFactory` class.
- If the client is an administrative class that is going to manipulate or traverse the JNDI tree, then use the `RMIInitialContextFactory` class.
- If the client is going to use DNS load balancing, then use the `RMIInitialContextFactory` class.

Configuring ORMI Request Load Balancing

Load balancing of client ORMI requests made to EJBs deployed into multiple clustered OC4J instances is supported in OC4J.

The default behavior is for the client to connect to the same OC4J instance with each call. Specifically, each time the client calls `InitialContext()` using the same `user/password/provider` URL combination, the cached `Context` object created the first time the client was invoked will be returned. Thus, the client will send requests to the same OC4J instance defined by this `Context` object.

In situations where the number of client calls is fairly low, connecting in this manner is efficient and results in the best performance.

However, in situations where heavy request volume is expected, load balancing of requests across OC4J instances may be desired. Load balancing is configurable using the `oracle.j2ee.rmi.loadBalance` property, which can be set in the client's `jndi.properties` file or in a Hashtable in the client code. The values for this property are outlined in the following table.

Table 6–6 oracle.j2ee.rmi.loadBalance Property Values

Value	Description
`client`	If specified, the client interacts with the OC4J process that was initially chosen at the first lookup for the entire conversation.
`context`	Used for a Web client (servlet or JSP) that will access EJBs in a clustered OC4J environment. If specified, a new `Context` object for a randomly-selected OC4J instance will be returned each time `InitialContext()` is invoked.
`lookup`	Used for a standalone client that will access EJBs in a clustered OC4J environment. If specified, a new `Context` object for a randomly-selected OC4J instance will be created each time the client calls `Context.lookup()`.

The following example illustrates how this property can be set to `lookup` in a `Hashtable`:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,"com.evermind.server.rmi.RMIInitialContext
Factory");
env.put(Context.SECURITY_PRINCIPAL, "jazn.com/admin");
env.put(Context.SECURITY_CREDENTIALS, "password");
env.put(Context.PROVIDER_URL,"opmn:ormi://<hostname>:oc4j_inst1/ejbsamples");
env.put("oracle.j2ee.rmi.loadBalance","lookup");
```

Example Lookups Using ORMI

This section provides examples of how to look up an EJB using ORMI in:

- Standalone OC4J 10g Release 3 (10.1.3)
- OC4J in Oracle Application Server 10g Release 3 (10.1.3)
- OC4J in Oracle Application Server Releases Before 10g Release 3 (10.1.3)

Standalone OC4J 10*g* Release 3 (10.1.3)

The following example shows how to look up an EJB called `MyCart` in the J2EE application `ejbsamples` deployed in a standalone OC4J instance. The application is located on a node configured to listen on RMI port 23792:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,"oracle.j2ee.rmi.RMIInitialContextFactory"
);
env.put(Context.SECURITY_PRINCIPAL, "admin");
env.put(Context.SECURITY_CREDENTIALS, "password");
env.put(Context.PROVIDER_URL, "ormi://<hostname>: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 10*g* Release 3 (10.1.3)

In Oracle Application Server 10*g* Release 3 (10.1.3), you can use the following type of lookup in the URL to look up an EJB in an Oracle Application Server environment.

The following example shows how to look up the EJB named `MyCart` in the J2EE application `ejbsamples` in an Oracle Application Server 10*g* Release 3 (10.1.3) environment. The differences between this invocation and the standalone invocation are: the `opmn` prefix to `ormi`, the specification of the OC4J instance name `oc4j_inst1` to which the EJB application is deployed, and no requirement to specify the RMI port:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,"oracle.j2ee.rmi.RMIInitialContextFactory"
);
env.put(Context.SECURITY_PRINCIPAL, "jazn.com/admin");
env.put(Context.SECURITY_CREDENTIALS, "password");
env.put(Context.PROVIDER_URL,"opmn:ormi://<hostname>:oc4j_inst1/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* Release 3 (10.1.3)

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* Release 3 (10.1.3), if you are accessing an EJB in Oracle Application Server, you have to know the RMI ports assigned by `opmn`. If you have only one JVM for your OC4J instance, then you have to limit the port ranges for RMIs to a specific number, for example: 3101-3101.

The following example shows how to look up an EJB called `MyCart` in the J2EE application `ejbsamples` in an Oracle Application Server environment in releases before 10*g* Release 3 (10.1.3). The application is located on a node configured to listen on RMI port 12401:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,"com.evermind.server.rmi.RMIInitialContext
Factory");
env.put(Context.SECURITY_PRINCIPAL, "jazn.com/admin");
env.put(Context.SECURITY_CREDENTIALS, "welcome");
env.put(Context.PROVIDER_URL, "ormi://<hostname>:12401/ejbsamples");

Context context = new InitialContext(env);
Object homeObject = context.lookup("MyCart");
CartHome home = (CartHome)PortableRemoteObject.narrow(homeObject,CartHome.class);
```

Configuring ORMI Tunneling through HTTP

To enable EJB communication through firewalls, ORMI utilizes HTTP tunneling to wrap RMI calls within an HTTP POST request. The request is then forwarded to the `default` Web application within the target OC4J instance. Note that tunneling is supported only with RMI/ORMI; you cannot perform HTTP tunneling using RMI/IIOP.

The HTTP tunneling URL set as the value of `java.naming.provider.url` has the following syntax. See "Setting JNDI Properties for RMI" on page 6-8 for details on this URL.

```
ormi:http://<hostname:[http_port]>/<context_uri>/<appName>
```

Table 6–7 HTTP tunneling URL syntax

Parameter	Description
`OHS_hostname`	The name of the Oracle HTTP Server host that will receive the request.
`http_port`	The Oracle HTTP Server port. This value is optional; if omitted, it defaults to `80`.
`context_uri`	The value of the root attribute in the `<default-web-app>` element in `ORACLE_HOME/j2ee/<instanceName>/default-web-site.xml`. The value for the OC4J `home` instance is `/j2ee`.
`appName`	The name of the target application.

You can target an application deployed to a specific OC4J instance - for example, `home` or `home2` - by specifying the URI context for the `default` Web application instance within the OC4J instance hosting the target application. This context URI is defined in `default-web-site.xml` as the value of the `root` attribute of the `<default-web-app>` element.

The following entry in `ORACLE_HOME/j2ee/home/default-web-site.xml` defines the context URI as `/j2ee` for `default` Web application in the `home` instance:

```
<default-web-app application="default" name="defaultWebApp" root="/j2ee" />
```

To target a request to the `acme` application deployed to the `home` instance, you would specify this context URI in the ORMI tunneling URL (assuming Oracle HTTP Server is configured to listen on port 7777):

```
ormi:http://OHShost:7777/j2ee/acme
```

If your HTTP traffic goes through a proxy Web server, specify the `proxyHost` and (optionally) `proxyPort` in the command line that starts the EJB client:

```
-Dhttp.proxyHost=<proxy_host> -Dhttp.proxyPort=<proxy_port>
```

If omitted, *`proxy_port`* defaults to `80`.

Using ORMI/SSL (ORMIS) in OC4J

ORMI over SSL (ORMIS) is a new feature in Oracle Application Server 10*g* Release 3 (10.1.3). With this feature, OC4J now supports Secure Socket Layer (SSL) RMI communication between objects across OC4J server instances.

ORMIS is disabled by default in OC4J, client and server keystores must be created before this feature can be used.

See the *Oracle Containers for J2EE Security Guide* for information on the following topics:

- Creating keystores and wallets
- Enabling ORMI/SSL (ORMIS) in OC4J
- Configuring OC4J to Use ORMIS
- Configuring Clients to Use ORMIS
- EJB Server Security Properties (internal-settings.xml)
- CSIv2 Security Properties
- CSIv2 Security Properties (internal-settings.xml)
- CSIv2 Security Properties (ejb_sec.properties)
- CSIv2 Security Properties (orion-ejb-jar.xml)
- EJB Client Security Properties (ejb_sec.properties)

Using J2EE Interoperability (RMI/IIOP)

This section describes OC4J support for EJBs and other objects to invoke one another across different J2EE containers—for example, between OC4J and BEA WebLogic servers—using the standard Remote Method Invocation (RMI)/Internet Inter-Orb Protocol (IIOP).

This section covers the following topics:

- Introduction to RMI/IIOP
- Switching from ORMI to IIOP 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 introduced 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 from ORMI to IIOP Transport" on page 6-18 discusses redeployment details.

EJB interoperability consists of the following:

- Transport interoperability, through CORBA IIOP Internet Inter-ORB Protocol, where ORB is Object Request Broker)
- Naming interoperability, through the CORBA CosNaming Service (CORBA Object Service Naming, part of the OMG CORBA Object Service specification)
- Security interoperability, through Common Secure Interoperability Version 2 (CSIv2)
- Transaction interoperability, through the CORBA Transaction Service (OTS)

OC4J furnishes transport, naming, and security interoperability.

Transport

By default, OC4J EJBs use RMI/Oracle Remote Method Invocation (ORMI), a proprietary protocol, to communicate as described in "Using Oracle Remote Method Invocation (RMI/ORMI)" on page 6-2.

However, you can easily convert an EJB to use RMI/IIOP, making it possible for EJBs to invoke one another across different EJB containers. This section describes configuring and using RMI/IIOP.

Note: For the OC4J 10*g* Release 3 (10.1.3) implementation, load balancing and failover are supported only for ORMI, not for 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.

For information about security topics, see the CSIv2 chapter in the *Oracle Containers for J2EE Security Guide*.

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.

The rmic.jar Compiler

To invoke or be invoked by CORBA objects, RMI objects must have corresponding stubs, skeletons, and Internet Description Language (IDL). Use the `rmic.jar` compiler to generate stubs and skeletons from Java classes or to generate IDL.

For use with RMI/IIOP, be sure to compile using the `-iiop` option.

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 6–8.

Table 6–8 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.
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.

Table 6–8 (Cont.) Interoperability Configuration Files

Context	File	Description
	`ejb_sec.properties`	This file specifies client-side security properties for an EJB.
	`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 6-17 for details.

For information about security topics, see the CSIv2 chapter in the *Oracle Containers for J2EE Security Guide*.

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.

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

  For details on corbaname URLs, see "Specifying the corbaname URL" on page 6-21. For details on the OPMN URL, see "Specifying the OPMN URL" on page 6-21.

- `contextFactory` can be either `ApplicationClientInitialContextFactory` or the class `IIOPInitialContextFactory`.

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

Context Factory Usage

- `oracle.j2ee.naming.ApplicationClientInitialContextFactory` is used when looking up remote objects from standalone application clients. It uses the `refs` and `ref-mappings` found in `application-client.xml` and `orion-application-client.xml`. It is the default initial context factory when the initial context is instantiated in a Java application.
- `oracle.j2ee.iiop.IIOPInitialContextFactory` is used when looking up remote objects between different containers using the IIOP protocol.

Client-Side Requirements to Use IIOP

This section lists the ZIP files that you use to install the Oracle and J2EE standard JAR files that enable EJB and JMS lookup using RMI/IIOP.

The following ZIP files are available from `http://www.oracle.com/technology/software/products/ias/index.html`:

- To enable EJB lookup using IIOP, download and expand `oc4j_iiop_client.zip`.

- To enable other lookups, such as JMS, you must also download and expand `oc4j_extended.zip`.

Once these ZIP files are expanded, make sure that `oc4jclient.jar` is included in the CLASSPATH.

The ZIP files contain all the JAR files required by the client. The JAR files contain the classes necessary for client interaction. You must only add `oc4jclient.jar` to your `CLASSPATH`, because all other JAR files required by the client are referenced in the `oc4jclient.jar` manifest classpath.

While running the IIOP Client, the following properties must be set for IIOP clients:

- `-Dorg.omg.CORBA.ORBInitialHost=${orb.host}`
- `-Dorg.omg.CORBA.ORBInitialPort=${orb.port}`
- `-Djavax.rmi.CORBA.PortableRemoteObjectClass=com.sun.corba.ee.impl.javax.rmi.PortableRemoteObject`
- `-Dcom.oracle.CORBA.OrbManager=com.oracle.corba.ee.impl.orb.ORBManagerImpl`

Switching from ORMI to IIOP Transport

In OC4J, EJBs use RMI/ORMI, a proprietary protocol, to communicate (as described in "Using Oracle Remote Method Invocation (RMI/ORMI)" on page 6-2). You can convert an EJB to use RMI/IIOP, making it possible for EJBs to invoke one another across different EJB containers from different vendors, such as OC4J and BEA WebLogic.

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 sections provide details on making the conversions:

- Configuring an EJB for Interoperability in a Standalone OC4J Environment
- Configuring an EJB for Interoperability in an Oracle Application Server Environment
- Specifying the corbaname URL
- Specifying the OPMN URL
- Exception Mapping
- Invoking OC4J-Hosted Beans from a Non-OC4J Container

Configuring an EJB for Interoperability in a Standalone OC4J Environment

Follow these steps to convert an EJB to use RMI/IIOP in a standalone OC4J environment:

1. Specify CSIv2 security policies for the bean in `orion-ejb-jar.xml` and in `internal-settings.xml`.

For information about security topics, see the CSIv2 chapter in the *Oracle Containers for J2EE Security Guide*.

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 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/_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 "Specifying the corbaname URL" on page 6-21.

> **Note:** IIOP stub and tie class code generation occurs at deployment time, unlike ORMI stub generation, which occurs at runtime. This is why you must add the JAR file to the `classpath` yourself. If you run in the server, a list of generated classes required by the server and IIOP stubs is made available automatically.

6. (Optional) To make the bean accessible to CORBA applications, run `rmic.jar` to generate IDL describing its interfaces.

Configuring an EJB for Interoperability in an Oracle Application Server Environment

This section describes 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`.

   For information about security topics, see the CSIv2 chapter in the *Oracle Containers for J2EE Security Guide*.

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_group" numprocs="1"/>
    </process-type>
</ias-component>
```

Note: If you choose to configure your client's JNDI property `java.naming.provider.url` to use an `OPMN` URL, then your client cannot connect to `iiops1` or `iiops2` ports because OPMN-allocated ports are not reported to OC4J.

3. Use `opmnctl` to restart all OC4J instances that are managed by OPMN. Use the `-DGenerateIIOP=true` flag.

```
opmnctl -DGenerateIIOP=true startall
```

4. Deploy your application specifying the `-enableIIOP` option. For information about deployment, see the *Oracle Containers for J2EE Deployment Guide*.

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/_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 "Specifying the corbaname URL" on page 6-21. For details on the OPMN URL, see "Specifying the OPMN URL" on page 6-21.

Note: IIOP stub and tie class code generation occurs at deployment time, unlike ORMI stub generation, which occurs at runtime. This is why you must add the JAR file to the `classpath` yourself. If you run in the server, a list of generated classes required by the server and IIOP stubs is made available automatically.

7. (Optional) To make the bean accessible to CORBA applications, run `rmic.jar` to generate IDL describing its interfaces.

Specifying the corbaname URL

To interoperate, an EJB must look up other beans using `CosNaming`. This means that the URL for looking up the root `NamingContext` must use the `corbaname` URL scheme instead of the `ormi` URL scheme. This section discusses the `corbaname` subset that EJB developers use most often. For a full discussion of the `corbaname` scheme, see section 2.5.3 of the CORBA Naming Service specification. The `corbaname` scheme is based on the `corbaloc` scheme, which section 13.6.10.1 of the CORBA specification discusses.

The most common form of the `corbaname` URL scheme is:

```
corbaname::host[:port]
```

This `corbaname` URL specifies a conventional DNS host name or IP address, and a port number. For example,

```
corbaname::example.com:8000
```

A `corbaname` URL can also specify a naming context by following the host and port by `#` and `NamingContext` in string representation. The `CosNaming` service on the specified host is responsible for interpreting the naming context.

```
corbaname::host[:port]#namingcontext
```

For example:

```
corbaname::example.com:8000#Myapp
```

Specifying the OPMN URL

This section describes OPMN URL details that are specific to RMI/IIOP. For general information about the OPMN URL, see "Setting JNDI Properties for RMI" on page 6-8.

In an Oracle Application Server environment, IIOP ports for all OC4J processes within each Oracle Application Server instance are dynamically managed by OPMN. Because of this, it may not be possible for clients to know the ports on which OC4J processes are actively listening for IIOP requests. To enable clients to successfully make RMI/IIOP requests in an Oracle Application Server environment without having to know the IIOP ports for all active OC4J processes, modify the `jndi.naming.provider.url` property (in the client's `jndi.properties` file) with a URL of the following format:

opmn:corbaname::*host*[:*port*][#*instance-name*]#*appname*

For example:

```
opmn:corbaname::dlsun74:6003#oc4j_inst1#ejbsamples
```

Notes:

- For the OC4J 10*g* Release 3 (10.1.3) implementation, load balancing and failover are supported only for ORMI, not for IIOP.
- If you use an OPMN URL, your client cannot connect to `iiops1` or `iiops2` (`ssl-port` or `ssl-client-server-auth-port`) ports.

Exception Mapping

When EJBs are invoked over IIOP, OC4J must map system exceptions to CORBA exceptions. Table 6–9 lists the exception mappings.

Table 6–9 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.

7 Java Object Cache

This chapter describes the Oracle Containers for J2EE (OC4J) Java Object Cache (JOC), including its architecture and programming features. This chapter covers the following topics:

What's New for 10.1.3

The following OC4J JOC features and behaviors are new for this release:

- DMS support
- Enhanced classloader support -

  Object names as well as the cached objects themselves can now be loaded using a region specific class loader.
- Enhanced remote cache access -

  It is now possible to add or retrieve an object from a specific remote cache using `CacheAccess.replaceRemote` and `CacheLoader.getFromRemote` respectively.

  `CacheAccess.getAllCached` allows the retrieval of a particular named object from all caches in the system. This is returned as a list. This is particularly useful in retrieving statistics from each cache.

  Previously, all listeners were executed asynchronously. The request would typically return before the event listener executed. In 10.1.3, synchronized update listeners have been added to allow listeners to be executed as part of the event rather than in a separate thread.
- **Deprecated**

  The following item is deprecated in this release:

– `Attributes.logging` levels have been replaced by those specified in `java.util.logging.Levels`.

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 Containers for J2EE JSP Tag Libraries and Utilities Reference.*

Java Object Cache Basic Architecture

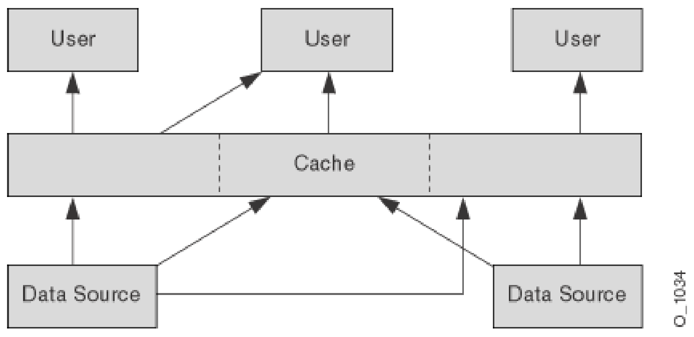
Figure 7–1, "Java Object Cache Basic Architecture" shows the basic architecture for the Java Object Cache. The cache delivers information to a user process. The process could be a servlet application that generates HTML pages, or any other Java application.

The Java Object Cache is an in-process, process-wide caching service for general application use. That is, objects are cached within the process memory space, and the Java Object Cache is a single service that is shared by all threads running in the process, in contrast to a service that runs in another process. The Java Object Cache can manage any Java object. To facilitate sharing of cached objects, all objects within the cache are accessed by name. The caching service does not impose a structure on objects being cached. The name, structure, type, and original source of the object are all defined by the application.

To maximize system resources, all objects within the cache are shared. However, access to cached objects is not serialized by access locks, allowing for a high level of concurrent access. When an object is invalidated or updated, the invalid version of the object remains in the cache as long as there are references to that particular version of the object. It is thus possible to have multiple versions of an object in the cache at the same time; however, there is never more than one valid version of the object. The old or invalid versions of an object are visible only to applications that had references to the version before it was invalidated. If an object is updated, a new copy of the object is created in the cache, and the old version is marked as invalid.

Objects are loaded into the cache with a user-provided `CacheLoader` object. This loader object is called by the Java Object Cache when a user application requests an object from the cache and it is not already present. Figure 7–1 is a graphical representation of the architecture. The application interacts with the cache to retrieve objects, and the cache interacts through the `CacheLoader` with the data source. This process gives a clean division between object creation and object use.

Figure 7–1 Java Object Cache Basic Architecture



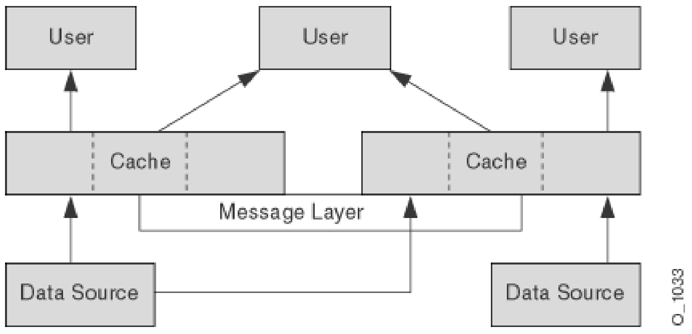
Distributed Object Management

The Java Object Cache can be used in an environment in which multiple Java processes are running the same application or working on behalf of the same application. In this environment, it is useful to have identical objects cached in different processes. For simplicity, availability and performance, the Java Object Cache is specific to each process. There is no centralized control of which objects are loaded into a process. However, the Java Object Cache coordinates object updating and invalidation between processes. If an object is updated or invalidated in one process, then it is also updated or invalidated in all other associated processes. This distributed management allows a system of processes to stay synchronized without the overhead of centralized control.

Figure 7–2, "Java Object Cache Distributed Architecture" 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 7–2 Java Object Cache Distributed Architecture



How the Java Object Cache Works

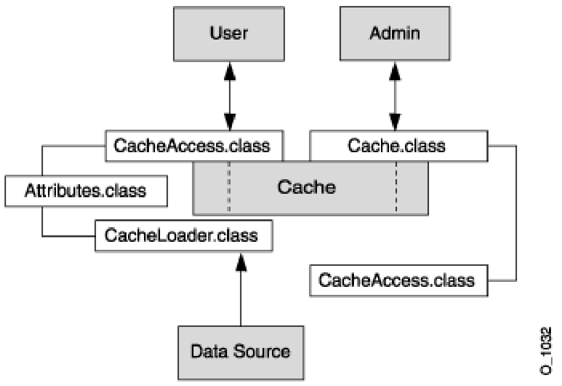
The Java Object Cache manages Java objects within a process, across processes, and on a local disk. The Java Object Cache provides a powerful, flexible, and easy-to-use service that significantly improves Java performance by managing local copies of Java objects. There are very few restrictions on the 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 7–3, "Java Object Cache Basic APIs" illustrates the basic Java Object Cache APIs. Figure 7–3, "Java Object Cache Basic APIs" does not show   management.

Figure 7–3   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 7–1, " Cache Organizational Construct" contains a summary of the constructs in the cache environment and the cache object types.

Table 7–1   Cache Organizational Construct

Cache Construct	Description
Attributes	Functionality associated with cache regions, groups, and individual objects. Attributes affect how the Java Object Cache manages objects.
Cache region	An organizational name space for holding collections of cache objects within the Java Object Cache.
Cache subregion	An organizational name space for holding collections of cache objects within a parent region, subregion, or group.
Cache group	An organizational construct used to define an association between objects. The objects within a region can be invalidated as a group. Common attributes can be associated with objects within a group.
Memory object	An object that is stored and accessed from memory.
Disk object	An object that is stored and accessed from disk.
Pooled object	A set of identical objects that the Java Object Cache manages. The objects are checked out of the pool, used, and then returned.
`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:

- 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

StreamAccess objects are special cache objects set up to be accessed using the Java `InputStream` and `OutputStream` classes. The Java Object Cache determines how to access the `StreamAccess` object, based on the size of the object and the capacity of the cache. Smaller objects are accessed from memory; larger objects are streamed directly from disk. All `streamAccess` objects are stored on disk.

The cache user's access to the `StreamAccess` object is through an `InputStream`. All the attributes that apply to memory objects and disk objects also apply to `StreamAccess` objects. A `StreamAccess` object does not supply a mechanism to manage a stream—for example, `StreamAccess` objects cannot manage socket endpoints. `InputStream` and `OutputStream` objects are available to access fixed-sized, potentially large objects.

Pool Objects

Pool objects are a special class of objects that the Java Object Cache manages. A pool object contains a set of identical object instances. The pool object itself is a shared object; the objects within the pool are private objects. Individual objects within the pool can be checked out to be used and then returned to the pool when they are no longer needed.

Attributes, including `TimeToLive` or `IdleTime` can be associated with a pool object. These attributes apply to the pool object as a whole.

The Java Object Cache instantiates objects within a pool using an application-defined factory object. The size of a pool decreases or increases based on demand and on the values of the `TimeToLive` or `IdleTime` attributes. A minimum size for the pool is specified when the pool is created. The minimum size value is interpreted as a request rather than a guaranteed minimum value. Objects within a pool object are subject to removal from the cache because of a lack of space, so the pool can decrease below the requested minimum value. A maximum pool size value can be set by putting a hard limit on the number of objects available in the pool.

Java Object Cache Environment

The Java Object Cache environment includes the following:

- Cache Regions
- Cache Subregions
- Cache Groups
- Region and Group Size Control
- Cache Object Attributes

This section describes these Java Object Cache environment constructs.

Cache Regions

The Java Object Cache manages objects within a cache region. A *cache region* defines a name space within the cache. Each object within a cache region must be uniquely named, and the combination of the cache region name and the object name must uniquely identify an object. Thus, cache region names must be unique from other region names, and all objects within a region must be uniquely named relative to the region. (Multiple objects can have the same name if they are within different regions or subregions.)

You can define as many regions as you need to support your application. However, most applications require only one region. The Java Object Cache provides a default region; when a region is not specified, objects are placed in the default region.

Attributes can be defined for a region and are then inherited by the objects, subregions, and groups within the region.

Cache Subregions

The Java Object Cache manages objects within a cache region. Specifying a subregion within a cache region defines a child hierarchy. A *cache subregion* defines a name space within a cache region or within a higher cache subregion. Each object within a cache subregion must be uniquely named, and the combination of the cache region name, the cache subregion name, and the object name must uniquely identify an object.

You can define as many subregions as you need to support your application.

A subregion inherits its attributes from its parent region or subregion unless the attributes are defined when the subregion is defined. A subregion's attributes are inherited by the objects within the subregion. If a subregion's parent region is invalidated or destroyed, the subregion is also invalidated or destroyed.

Cache Groups

A *cache group* creates an association between objects within a region. Cache groups allow related objects to be manipulated together. Objects are typically associated in a cache group because they must be invalidated together, or they use common attributes. Any set of cache objects within the same region or subregion can be associated using a cache group, which can, in turn, include other cache groups.

A Java Object Cache object can belong to only one group at any given time. Before an object can be associated with a group, the group must be explicitly created. A group is defined with a name. A group can have its own attributes, or it can inherit its attributes from its parent region, subregion, or group.

Group names are not used to identify individual objects, but rather to define a set or collection of objects that have something in common. A group does not define a hierarchical name space. Object type does not distinguish objects for naming purposes; therefore, a region cannot include a group and a memory object with the same name. You must use subregions to define a hierarchical name space within a region.

Groups can contain groups, with the groups having a parent and child relationship. The child group inherits attributes from the parent group.

Region and Group Size Control

With the 10*g* Release 3 (10.1.3) version of the Java Object Cache, you can specify the maximum size of a region or group as either the number of objects in the region or group, or the maximum number of bytes allowed. If the number of bytes controls the region capacity, then set the size attribute for all objects in the region. This can be set either directly by the user when the object is created, or automatically by setting the `Attributes.MEASURE` attribute flag. You can set the size of a region or group at multiple levels in the naming hierarchy—that is, at the region and subregion level, or at the group level within a region or another group.

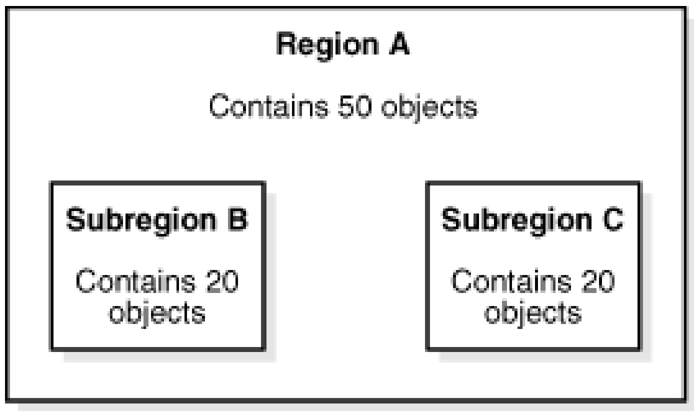
When the capacity of a region or group is reached, the `CapacityPolicy` object associated with that region or group, if defined, is called. If no capacity policy has been specified, then the default policy is used. The default policy follows: If a nonreferenced object of lesser or equal priority is found, then it is invalidated in favor of the new object. If the priority attribute has not been set for an object, then the priority is assumed to be `Integer.MAX_VALUE`. When searching for an object to remove, all objects in the immediate region or group and all subregions and subgroups are searched. The first object that can be removed, based on the capacity policy, is removed. So, for example, this may not be the object of lowest priority in the search area.

Figure 7–4, "Capacity Policy Example" illustrates the example.

The capacity of region A is set to 50 objects, with subregion B and subregion C set to 20 objects each. If the object count of region A reaches 50, with 10 directly in region A and 20 each in subregions B and C, then the capacity policy for region A is called. The object that is removed can come from region A or from one of its subregions. Figure 7–4, "Capacity Policy Example" shows this situation.

If subregion B reaches 20 before the capacity of region A is reached, then the capacity policy for subregion B is called, and only objects within subregion B are considered for removal.

Figure 7–4 Capacity Policy Example



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 set 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 7–2, " Java Object Cache Attributes–Set at Object Creation" summarizes these attributes. None of the attributes shown in Table 7–2, " Java Object Cache Attributes–Set at Object Creation" 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 7–3, " Java Object Cache Attributes" summarizes these attributes.

Note: Some attributes do not apply to certain types of objects. See the "Object Types" sections in the descriptions in Table 7–2, " Java Object Cache Attributes–Set at Object Creation" and Table 7–3, " Java Object Cache Attributes".

Using Attributes Defined Before Object Loading

The attributes shown in Table 7–2, " Java Object Cache Attributes–Set at Object Creation" 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 7–2, " Java Object Cache Attributes–Set at Object Creation" (by setting the values of an `Attributes` object argument).

- `CacheAccess.defineRegion()`
- `CacheAccess.defineSubRegion()`
- `CacheAccess.defineGroup()`

- `CacheAccess.defineObject()`
- `CacheAccess.getAccess()`
- `CacheAccess.getSubRegion()`
- `CacheAccess.put()`
- `CacheAccess.createPool()`
- `CacheLoader.createDiskObject()`
- `CacheLoader.createStream()`
- `CacheLoader.SetAttributes()`

Note: You cannot reset the attributes shown in Table 7–2, " Java Object Cache Attributes–Set at Object Creation" by using the `CacheAccess.resetAttributes()` method.

Table 7–2 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.
`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.

Table 7–2 (Cont.) Java Object Cache Attributes–Set at Object Creation

Attribute Name	Description
`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 spooled to disk.
`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.)

Table 7–2 (Cont.) Java Object Cache Attributes–Set at Object Creation

Attribute Name	Description
`MEASURE`	Specifies that the size attribute of the cached object is calculated, automatically, when the object is loaded or replaced in the cache. The capacity of the cache or region can then be accurately controlled based on object size, rather than object count. Object Types: When set on a region, subregion, or group, this attribute applies individually to each object within the region, subregion, or group unless explicitly set for the object. Default Value: OFF. (The size of an object is not automatically calculated.)
`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.)
`Classloader`	Specifies the classloader that should be used when an object or object name is instantiated from disk or when received from another cache over the network. Default Value: The default is to use the system classloader. The loader the `cache.jar` was loaded by.

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 7–3, " Java Object Cache Attributes" lists these attributes. These attributes can be set using the methods in the list under "Using Attributes Defined Before Object Loading" on page 7-10, and can be reset using the `CacheAccess.resetAttributes()` method.

Table 7–3 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.
`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.

Table 7–3 (Cont.) Java Object Cache Attributes

Attribute Name	Description
`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, 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`.
`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 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 `$OLE_HOME/javacache/lib` on UNIX or in `%OLE_HOME%\javacache\lib` on Windows.

To use the Java Object Cache, import `oracle.ias.cache`, as follows:

```
import oracle.ias.cache.*;
```

Defining a Cache Region

All access to the Java Object Cache is through a `CacheAccess` object, which is associated with a cache region. You define a cache region, usually associated with the name of an application, using the `CacheAccess.defineRegion()` static method. If the cache has not been initialized, then `defineRegion()` initializes the Java Object Cache.

When you define the region, you can also set attributes. Attributes specify how the Java Object Cache manages objects. The `Attributes.setLoader()` method sets the name of a cache loader. Example 7–1, "Setting the Name of a CacheLoader" shows this.

Example 7–1 Setting the Name of a CacheLoader

```
Attributes attr = new Attributes();
MyLoader mloader = new MyLoader;
attr.setLoader(mloader);
attr.setDefaultTimeToLive(10);

final static String APP_NAME_ = "Test Application";
CacheAccess.defineRegion(APP_NAME_, attr);
```

The first argument for `defineRegion` uses a `String` to set the region name. This static method creates a private region name within the Java Object Cache. The second argument defines the attributes for the new region using the default cache attributes.

Defining a Cache Group

Create a cache group when you want to create an association between two or more objects within the cache. Objects are typically associated in a cache group because they must be invalidated together or because they have a common set of attributes.

Any set of cache objects within the same region or subregion can be associated using a cache group, including other cache groups. Before an object can be associated with a cache group, the cache group must be defined. A cache group is defined with a name and can use its own attributes, or it can inherit attributes from its parent cache group, subregion, or region. The code in Example 7–2, "Defining a Cache Group" defines a cache group within the region named `Test Application`:

Example 7–2 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 7–3, "Defining a Cache Subregion" defines a cache subregion within the region named `Test Application`.

Example 7–3 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, before an individual object is loaded, how the object should be managed within the cache. 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 7–4, "Setting Cache Attributes" shows how to set attributes for a cache object. The example assumes that the region `APP_NAME_` has already been defined.

Example 7–4 Setting Cache Attributes

```
import oracle.ias.cache.*;
final static String APP_NAME_ = "Test Application";
CacheAccess cacc = null;
try
{
   cacc = CacheAccess.getAccess(APP_NAME_);
// set the default IdleTime for an object using attributes
   Attributes attr = new Attributes();
// set IdleTime to 2 minutes
   attr.setIdleTime(120);

// define an object and set its attributes
   cacc.defineObject("Test Object", attr);

// object is loaded using the loader previously defined on the region
// if not already in the cache.
   result = (String)cacc.get("Test Object");
}  catch (CacheException ex){
     // handle exception
   } finally {
      if (cacc!= null)
         cacc.close();
}
```

Implementing a CacheLoader Object

The Java Object Cache has two mechanisms for loading an object into the cache:

- The object can be put into the cache directly by the application using the `CacheAccess.put()` method.
- You can implement a `CacheLoader` object.

In most cases, implementing the `CacheLoader` is the preferred method. With a cache loader, the Java Object Cache automatically determines if an object needs to be loaded into the cache when the object is requested. And the Java Object Cache coordinates the load if multiple users request the object at the same time.

A `CacheLoader` object can be associated with a region, subregion, group, or object. Using a `CacheLoader` allows the Java Object Cache to schedule and manage object loading, and handle the logic for "if the object is not in cache then load."

If an object is not in the cache, then when an application calls the `CacheAccess.get()` or `CacheAccess.preLoad()` method, the cache executes the `CacheLoader.load` method. When the `load` method returns, the Java Object Cache inserts the returned object into the cache. Using `CacheAccess.get()`, if the cache is full, the object is returned from the loader, and the object is immediately invalidated in the cache. (Therefore, using the `CacheAccess.get()` method with a full cache does not generate a `CacheFullException`.)

When a `CacheLoader` is defined for a region, subregion, or group, it is taken to be the default loader for all objects associated with the region, subregion, or group. A `CacheLoader` object that is defined for an individual object is used only to load the object.

Note: A `CacheLoader` object that is defined for a region, subregion, or group or for more than one cache object must be written with concurrent access in mind. The implementation should be thread-safe, because the `CacheLoader` object is shared.

Using CacheLoader Helper Methods

The `CacheLoader` cache provides several helper methods that you can use from within the `load()` method implementation. Table 7–4, " CacheLoader Methods Used in load()" summarizes the available `CacheLoader` methods.

Table 7–4 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.
`getFromRemote()`	Retrieve an object from a specified remote cache.
`exceptionHandler()`	Converts noncache exceptions into `CacheExceptions`, with the base set to the original exception.
`log()`	Records messages in the cache service log.

Example 7–5, "Implementing a CacheLoader" illustrates a `CacheLoader` object using the `cacheLoader.netSearch()` method to check whether the object being loaded is available in distributed Java Object Cache caches. If the object is not found using `netSearch()`, then the load method uses a more expensive call to retrieve the object. (An expensive call may involve an HTTP connection to a remote Web site or a connection to the Oracle9*i* Database Server.) For this example, the Java Object Cache stores the result as a `String`.

Example 7–5 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, then `getNextObject` or `getNamedObject` should be called first to set the correct context.

The `CacheAccess.loadList` method takes as an argument an array of object names to be loaded. The cache processes this array of objects. Any objects that are not currently in the cache are added to a list that is passed to the `CacheListLoader` object that is defined for the cached objects. If a `CacheListLoader` object is not defined for the objects or the objects have different `CacheListLoader` objects defined, then each object is loaded individually, using the `CacheLoader.load` method defined.

It is always best to implement both the `CacheListLoader.loadList` method and the `CacheListLoader.load` method. Which method is called depends on the order of the user requests to the cache. For example, if the `CacheAccess.get` method is called before the `CacheAccess.loadList` method, then the `CacheListLoader.load` method is used rather than the `CacheAccess.loadList` method.

As a convenience, the invalidate and destroy methods have been overloaded to also handle an array of objects.

Example 7–6, "Sample CacheListLoader" shows a sample `CacheListLoader`, and Example 7–7, "Sample Usage" shows sample usage.

Example 7–6   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 7–7   Sample Usage

```
// Assumes the cache has already been initialized

CacheAccess   cacc;
Attributes    attr;
ListLoader    loader = new
ListLoader();
String        objList[];
Object        obj;

// set the CacheListLoader for the region
attr  = new Attributes();
attr.setLoader(loader);

//define the region and get access to the cache
CacheAccess.defineRegion("region name", attr);
cacc = CacheAccess.getAccess("region name");

// create the array of object names
objList = new String[3];
for (int j = 0; j < 3; j++)
      objList[j] = "object " + j;

// load the objects in the cache via the CacheListLoader.loadList method
cacc.loadList(objList);

// retrieve the already loaded object from the cache
obj = cacc.get(objList[0]);

// do something useful with the object

// load an object using the CacheListLoader.load method
obj = cache.get("another object")

// do something useful with the object
```

Java Object Cache Configuration

The Java Object Cache is NOT initialized automatically upon OC4J startup. You can force OC4J to initialize `jcache` by using `-Doracle.ias.jcache=true` in `opmn.xml`, as shown in the following example:

```
   <ias-component id="OC4J">
            <process-type id="home" module-id="OC4J" status="enabled">
              <module-data>
                  <category id="start-parameters">
                    <data id="java-options" value="-server
-Djava.security.policy=$OLE_HOME/j2ee/home/config/java2.policy
 -Djava.awt.headless=true
-DApplicationServerDebug=true
-Ddatasource.verbose=true
-Djdbc.debug=true -Doracle.ias.jcache=true"/>
                  </category>
                  <category id="stop-parameters">
                    <data id="java-options"
value="-Djava.security.policy=$OLE_HOME/j2ee/home/config/java2.
policy -Djava.awt.headless=true"/>
                  </category>
              </module-data>
              <start timeout="600" retry="2"/>
              <stop timeout="120"/>
              <restart timeout="720" retry="2"/>
              <port id="ajp" range="3301-3400"/>
              <port id="rmi" range="3201-3300"/>
              <port id="jms" range="3701-3800"/>
              <process-set id="default_group" numprocs="1"/>
            </process-type>
        </ias-component>
```

The OC4J runtime initializes the Java Object Cache using configuration settings defined in the file `javacache.xml`. The file path is specified in the `<javacache-config>` tag of the OC4J `server.xml` file. The default relative path values of `javacache.xml` in `server.xml` are the following:

```
<javacache-config path="../../../javacache/admin/javacache.xml"/>
```

The rules for writing `javacache.xml` and the default configuration values are specified in an XML schema. The XML schema file `ora-cache.xsd` and the default `javacache.xml` are in the directory `$OLE_HOME/javacache/admin` on UNIX and in `%OLE_HOME%\javacache\admin` on Windows.

For reference documentation of `server.xml`, see *Oracle Containers for J2EE Configuration and Administration Guide* Appendix B - Configuration Files Used in OC4J, Section - "Overview of the OC4J Server Configuration File (server.xml)".

In earlier versions of Java Object Cache (before 9.0.4), configuration was done through the file `javacache.properties`. Starting with version 10g (9.0.4), Java Object Cache configuration is done through `javacache.xml`.

Note: If you install both a release that uses `javacache.properties` (before 9.0.4) and a release that uses `javacache.xml` (9.0.4 or later) on the same host, then you must ensure that the `javacache.xml` `discovery-port` attribute and `javacache.properties` `coordinatorAddress` attribute are not configured to the same port. If they are, then you must manually change the value in one or the other to a different port number. The default range is 7000-7099.

A sample configuration follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<cache-configuration
xmlns=http://www.oracle.com/oracle/ias/cache/configuration
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.oracle.com/oracle/ias/cache/configuration
ora-cache.xsd">
  <logging>
    <location>javacache.log</location>
    <level>ERROR</level>
  </logging>
  <communication>
    <isDistributed>true</isDistributed>
    <discoverer discovery-port="7000"/>
  </communication>
  <persistence>
    <location>diskcache</location>
    <disksize>32</disksize>
  </persistence>
  <max-objects>1000</max-objects>
  <max-size>48</max-size>
  <clean-interval>30</clean-interval>
</cache-configuration>
```

Table 7–5, " Java Object Cache Configuration Properties" contains the valid property names and the valid types for each property.

Table 7–5 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 7–3 describes these attributes.) Default value: `60`	Positive integer
`ping-interval`	Specifies the time, in seconds, between each cache death detection for determining the availability of the remote cache systems. Default value: `60`	Positive integer
`max-size`	Specifies the maximum size of the memory, in megabytes, available to the Java Object Cache. Default value: `10`	Positive integer

Table 7–5 (Cont.) Java Object Cache Configuration Properties

Configuration XML Element	Description	Type
`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
`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 7-26. Also see "Examples" on page 7-25. **Note**: The file path of the declarative cache XML schema is *`OLE_HOME`*`/javacache/admin/cache.xsd`. Refer to the XML schema when writing a declarative cache file. Default value: To 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. If the `distribute` property is set for an object, then updates and invalidation for that object 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 7-25. Default value: Cache is not distributed (`isDistributed` subelement is set to `false`).	Complex (has subelements)

`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`, `TE`, `INFO`, and `DEBUG`. See "Examples" on page 7-25.
	
	These log level are deprecated. Java cache now uses levels as specified in `java.util.logging.Level`.
	
	Default log file name:
	
	on UNIX:
	
	`$OLE_HOME/javacache/admin/logs/javacache.log`
	
	on Windows:
	
	`%OLE_HOME%\javacache\admin\logs\javacache.log`
	
	Default 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 10 MB. The unit of the disk cache size is megabytes. See "Examples" on page 7-25. 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 illustrates the use of the `<preload-file>` element:

- Specify a declarative cache configuration file:

```
<preload-file>/app/oracle/javacache/admin/decl.xml</preload-file>
```

The following examples illustrate the use of the `<communication>` element:

- Turn off distributed cache:

```
<communication>
  <isDistributed>false</isDistributed>
</communication>
```

- Distribute cache among multiple JVMs in local machine:

```
<communication>
  <isDistributed>true</isDistributed>
</communication>
```

- Specify the initial discovery port that the Java Object Cache initially contacts to join the caching system in the local node:

```
<communication>
  <isDistributed>true</isDistributed>
  <discoverer discovery-port="7000">
</communication>
```

- Specify the IP address and initial discovery port that the Java Object Cache initially contacts to join the caching system.

```
<communication>
<isDistributed>true</isDistributed>
<discoverer ip="192.10.10.10" discovery-port="7000">
</communication>
```

- Specify multiple IP addresses and the initial discovery port that the Java Object Cache initially contacts to join the caching system. If the first specified address is not reachable, it contacts the next specified address:

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

The following examples illustrate the use of the `<persistence>` element:

- Specify a root path for the disk cache using the default disk size:

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

- Specify a root path for the disk cache with a disk size of 20 MB:

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

The following examples illustrate the use of the `<logging>` element:

- Specify a log file name:

```
<logging>
<location>/app/9iAS/javacache/admin/logs/my_javacache.log</location>
</logging>
```

- Specify log level as `INFO`:

```
<logging>
<location>/app/9iAS/javacache/admin/logs/my_javacache.log</location>
<level>INFO</level>
</logging>
```

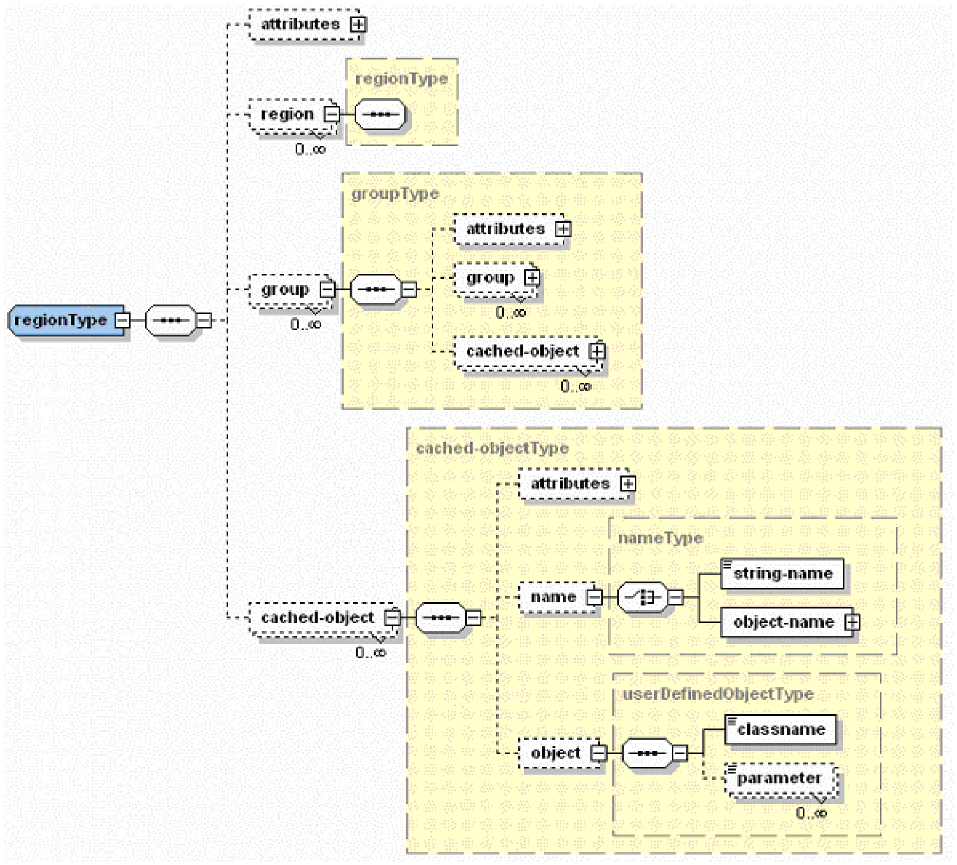
Declarative Cache

With the 10*g* Release 3 (10.1.3) 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. 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 7–5, "Declarative Cache Architecture" shows the declarative cache.

Figure 7–5 Declarative Cache Architecture



You can set up the Java Object Cache to automatically load a declarative cache file during system initialization. Example 7–8, "Automatically Load Declarative Cache" shows this. Example 7–9, "Programmatically Read Declarative Cache File" shows how to programmatically read the declarative cache file.

Example 7–8 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 7–9 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 *`OLE_HOME`*`/javacache/admin/cache.xsd`.

Table 7–6, " Description of Declarative Cache Schema (cache.xsd)" lists the elements of the declarative cache schema, their children, and the valid types for each element. See "Examples" on page 7-30 for code that shows usage for most elements.

Table 7–6 Description of Declarative Cache Schema (cache.xsd)

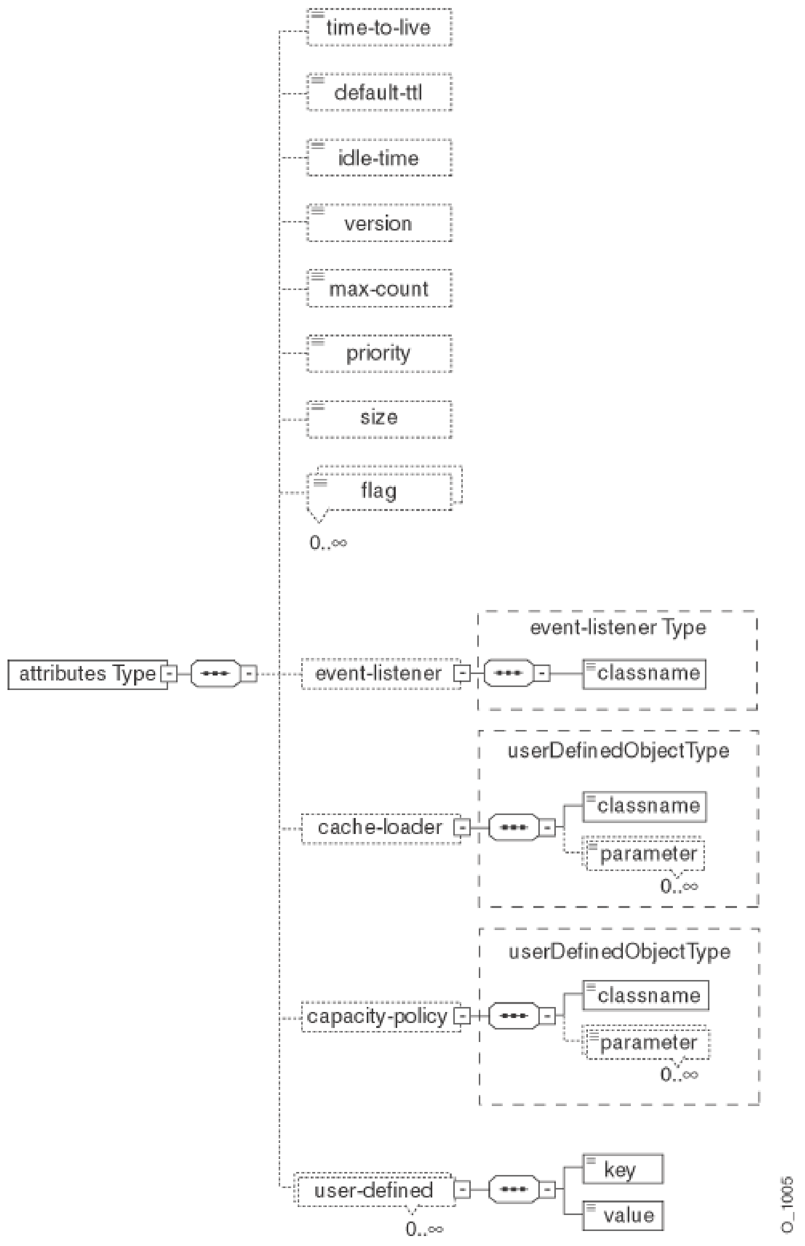
Element	Description	Children	Type
region	Declare a cache region or subregions.	<attributes> <region> <group> <cached-object>	regionType
group	Declare a cache group or subgroup.	<attributes> <group> <cached-object>	groupType
cached-object	Declare a cache object.	<attributes> <name> <object>	objectType

Table 7–6 (Cont.) Description of Declarative Cache Schema (cache.xsd)

Element	Description	Children	Type
name	Declare the name for a cached object. The name can use a simple string type or it can be a type of a specified Java object.	<string-name> <object-name>	nameType
object	Declare a user-defined Java object. The class of the specified object must implement the declarable interface of the `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 7–6, "Declarative Cache Schema Attributes" shows the attributes of the declarative cache schema.

Figure 7–6 Declarative Cache Schema Attributes



Examples

The following examples show the use of elements in Table 7–6, " Description of Declarative Cache Schema (cache.xsd)":

- Declare cache region and subregions with the `<region>` element:

```
<region name="themes">
    <region name="cartoon">
```

```
      <!-- sub region definition -->
    </region>
  <group name="colors">
    <!-- group definition -->
  </group>
</region>
```

- Declare cache group and subgroups with the `<group>` element:

```
<group name="colors">
  <group name="dark">
    <!-- sub group definition -->
  </group>
</group>
```

- Declare a cached object with the `<cached-object>` element:

```
<cached-object>
  <name>
    <string-name>DialogHandler</string-name>
  </name>
  <object>
    <classname>com.acme.ConfigManager</classname>
    <parameter name="color">blue</parameter>
  </object>
</cached-object>
```

- Declare the name for a cached object with the `<name>` element using a string:

```
<name>
  <string-name>DialogHandler</string-name>
</name>
```

  Declare the name for a cached object with the `<name>` element using an object:

```
<name>
  <object-name>
    <classname>DialogHandler</classname>
    <parameter name="color">green</parameter>
  </object-name>
</name>
```

- Declare a user-defined Java object with the `<object>` element:

```
<object>
  <classname>com.acme.CustomConfigManager</classname>
  <parameter name="color">blue</parameter>
</object>

// Implementation of CustomConfigManager.java
package com.acme;
import oracle.ias.cache.Declarable;
public class CustomConfigManager implements Declarable {
}
```

- Declare an attributes object for a cache region, group, or cache object with the `<attributes>` element:

```
<attributes>
  <time-to-live>4500</time-to-live>
  <default-ttl>6000</default-ttl>
  <version>99</version>
```

```
    <max-count>8000</max-count>
    <priority>50</priority>
    <flag>spool</flag>
    <flag>allownull</flag>
    <flag>distribute</flag>
    <flag>reply</flag>
    <cache-loader>
      <classname>MyLoader</classname>
      <parameter name="debug">false</parameter>
    </cache-loader>
  </attributes>
```

- Declare user-defined string type attributes with the `<user-defined>` element:

```
  <attributes>
    <user-defined>
      <key>color</key>
      <value>red</value>
    </user-defined>
  </attributes>
```

Declarable User-Defined Objects

The topology of the cache objects, object attributes, and user-defined objects can all be described in the declarative cache file. For the system to load and instantiate a user-defined Java object (including `CacheLoader`, `CacheEventListener`, and `CapacityPolicy`) declared in the declarative cache file, the object must be an instance of the `oracle.ias.cache.Declarable` interface. That is, you must implement the `oracle.ias.cache.Declarable` interface for any Java objects declared in the declarative cache file. You must be aware that all user-defined Java objects are loaded by the JVM's default class loader instead of the application's class loaders. After the declarable object is instantiated, the system implicitly invokes its `init(Properties props)` method. The method uses the user-supplied parameters (name-value pair) defined in the declarative cache file to perform any necessary initialization task. Example 7–10, "Define An Object by Declaratively Passing in a Parameter" shows how to define an object by declaratively passing in a parameter (color = yellow).

Example 7–10 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 7–11, "Declarable CacheLoader Implementation" shows a declarable `CacheLoader` implementation.

Example 7–11 Declarable CacheLoader Implementation

```
import oracle.ias.cache.*;
import java.util.Properties;

public class MyCacheLoader extends CacheLoader implements Declarable {

  public Object load(Object handle, Object argument) {
    // should return meaningful object based on argument
    return null;
  }

  public void init(Properties prop) {
  }
}
```

Initializing the Java Object Cache in a Non-OC4J Container

To use the Java Object Cache in any Java application but run it in a non-OC4J runtime, insert the following reference to where the application (Java class) is initialized:

```
Cache.open(/path-to-ocnfig-file/javacache.xml);
```

If you invoke `Cache.open()` without any parameter in your code, then the Java Object Cache uses its internal default configuration parameter. You can also initialize the Java Object Cache by invoking `Cache.init(CacheAttributes)`. This allows you to derive the configuration parameters from your own configuration file or generate them programmatically.

If the Java Object Cache is not used in the OC4J runtime, you must include `cache.jar` in the classpath where the JVM is launched. You must also initialize the Java Object Cache explicitly by invoking `Cache.open(String` *`config_filename`*`)`, where *`config_filename`* is the full path to a valid `javacache.xml` file, or by invoking `Cache.init(CacheAttributes)`.

Use any of the following method invocations to initialize the Java Object Cache explicitly in a non-OC4J container:

- `Cache.open();`

Use the default Java Object Cache configuration stored in the `cache.jar` file.

- `Cache.open(`*`/path-to-oracle-home/javacache/admin/javacache.xml`*`);`

  Use the configuration defined in the `javacache.xml` file.
- `Cache.open(`*`/path-to-user's-own-javacache.xml`*`);`

  Use the configuration defined in the specific `javacache.xml` file.
- `Cache.init(CacheAttributes);`

  Use the configuration that is set in a `CacheAttributes` object.

For J2EE applications running in an OC4J container, the path to the `javacache.xml` file can be configured in the OC4J `server.xml` configuration file. The cache can be initialized automatically when the OC4J process is started. See OC4J configuration for details.

In a non-OC4J container, if you do not use any of the preceding method invocations, the Java Object Cache is initialized implicitly (using default configuration settings stored in `cache.jar`) when you invoke `Cache.getAccess()` or `Cache.defineRegion()`.

Capacity Control

The capacity control feature allows the cache user to specify the policy for determining which objects should be removed from the cache when the capacity of the cache, region, or group has been reached. To specify the policy, extend the abstract class `CapacityPolicy`, and set the instantiated object as an attribute of the cache, region, or group.

For regions and groups, the `CapacityPolicy` object is called when the region or group has reached its capacity and a new object is being loaded. An object in the region or group must be found to invalidate, or the new object is not saved in the cache. (It is returned to the user but is immediately invalidated.)

The `CapacityPolicy` object that is associated with the cache as a whole is called when capacity of the cache reaches some "high water mark," some percentage of the configured maximum. When the high water mark is reached, the cache attempts to remove objects to reduce the load in the cache to 3% below the high water mark. The high water mark is specified by the `capacityBuffer` cache attribute. If the `capacityBuffer` is set to 5, then the cache begins removing objects from the cache when it is 95% full (100% -5%) and continues until the cache is 92% full (95% - 3%). The default value for `capacityBuffer` is 15.

The capacity policy used for the cache can be different from those used for specific regions or groups.

By default, the capacity policy for groups and regions is to remove a nonreferenced object of equal or lesser priority when a new object is added and capacity has been reached. For the cache, the default policy is to remove objects that have not been referenced in the last two clean intervals, with preference to objects of priority—that is, low priority objects that have not been referenced recently are removed first.

To help create a capacity policy, many statistics are kept for objects in the cache and aggregated across the cache, regions, and groups. The statistics are available to the `CapacityPolicy` object. For cache objects, the following statistics are maintained:

- Priority
- Access count—the number of times the object has been referenced

- Size—the size of the object in bytes (if available)
- Last access time—the time in milliseconds that the object was last accessed
- Create time—the time in milliseconds when the object was created
- Load time—the number of milliseconds required to load the object (if the object was added to the cache with `CacheAccess.put`, this value is 0)

Along with these statistics, all attributes associated with the object are available to the `CapacityPolicy` object.

The following aggregated statistics are maintained for the cache, regions, and groups. For each of these statistics, the low, high, and average value is maintained. These statistics are recalculated at each clean interval or when `Cache.updateStats()` is called.

- Priority
- Access count—the number of times that the object has been referenced
- Size—the size of the object in bytes (if available)
- Last access time—the time in milliseconds that the object was last accessed
- Load time—the number of milliseconds required to load the object (if the object was added to the cache with `CacheAccess.put`, this value is 0)

Example 7–12, "Sample CapacityPolicy Based on Object Size" is a sample `CapacityPolicy` object for a region, based on object size.

Example 7–12 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 7–13, "Sample CapacityPolicy Based on Access Time and Reference Count" 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 7–13 Sample CapacityPolicy Based on Access Time and Reference Count

```
class SizePolicy extends CapacityPolicy
{
public boolean policy (Object victimHandle, AggregateStatus aggStatus, long
 currentTime , Object newObjectHandle) throws CacheException
{
   long          lastAccess;
   int           accessCount;
   int           avgAccCnt;
   lastAccess    = getStatus(victimHandle).getLastAccess();
   accessCount   = getStatus(victimHandle).getAccessCount();
   avgAccCnt     = aggStatus.getAccessCount(AggregateStatus.AVG);

   if (lastAccess + 30000 < currentTime && accessCount < avgAccCnt)
      return true;
   }

}
```

Implementing a Cache Event Listener

Many events can occur in the life cycle of a cached object, including object creation and object invalidation. This section shows how an application can be notified when cache events occur.

To receive notification of the creation of an object, implement event notification as part of the `cacheLoader`. For notification of invalidation or updates, implement a `CacheEventListener`, and associate the `CacheEventListener` with an object, group, region, or subregion using `Attributes.setCacheEventListener()`.

`CacheEventListener` is an interface that extends `java.util.EventListener`. The cache event listener provides a mechanism to establish a callback method that is registered and then executes when the event occurs. In the Java Object Cache, the event listener executes when a cached object is invalidated or updated.

An event listener is associated with a cached object, group, region, or subregion. If an event listener is associated with a group, region, or subregion, then by default, the listener runs only when the group, region, or subregion itself is invalidated. Invalidating a member does not trigger the event. The `Attributes.setCacheEventListener()` call takes a boolean argument that, if `true`, applies the event listener to each member of the region, subregion, or group, rather than to the region, subregion, or group itself. In this case, the invalidation of an object within the region, subregion, or group triggers the event.

The `CacheEventListener` interface has one method, `handleEvent()`. This method takes a single argument, a `CacheEvent` object that extends `java.util.EventObject`. This object has a number of methods to help process events:

- `getID()`, which returns the type of event (`OBJECT_INVALIDATION`, `OBJECT_UPDATED`, or `OBJECT_UPDATED_SYNC`)
- `getSource()`, which returns the object being invalidated. For groups and regions, the `getSource()` method returns the name of the group or region.
- `getName()`, which returns the name of the object associated with the event
- `getRegion()`, which returns the region containing the object associated with the event
- `getReason()`, which returns a reason for the event. This currently only applies to invalidation events.

The `handleEvent()` method is executed in the context of a background thread that the Java Object Cache manages. Avoid using Java Native Interface (JNI) code in this method, because the expected thread context may not be available.

Example 7–14, "Implementing a CacheEventListener" illustrates how a `CacheEventListener` is implemented and associated with an object or a group.

Example 7–14 Implementing a CacheEventListener

```
import oracle.ias.cache.*;

    // A CacheEventListener for a cache object
    class MyEventListener implements CacheEventListener
    {

        public void handleEvent(CacheEvent ev) throws CacheException
        {
           MyObject obj = (MyObject)ev.getSource();
           obj.cleanup();
        }
     }

 class MyObject
 {
   public void cleanup()
   {
     // do something
   }
 }

 import oracle.ias.cache.*;

    // A CacheEventListener for a group object
     class MyGroupEventListener implements CacheEventListener
     {
        public void handleEvent(CacheEvent ev) throws CacheException
        {
           String groupName = (String)ev.getSource();
           notify("group " + groupName + " has been invalidated");

        }
        void notify(String str)
        {
          // do something
        }
     }
```

Use the `Attributes`.setCacheEventListener() method to specify the `CacheEventListener` for a region, subregion, group, or object.

Example 7–15, "Setting a Cache Event Listener on an Object" illustrates how to set a cache event listener on an object. Example 7–16, "Setting a Cache Event Listener on a Group" illustrates how to set a cache event listener on a group.

Example 7–15 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 7–16 Setting a Cache Event Listener on a Group

```
import oracle.ias.cache.*;
try
{
   CacheAccess cacc = CacheAccess.getAccess(myRegion);
   Attributes attr = new Attributes ();

   MyGroupEventListener listener = new MyGroupEventListener();
   attr.setCacheEventListener(CacheEvent.OBJECT_INVALIDATED, listener);

   cacc.defineGroup("myGroup", attr);
   //....
   cacc.close();

}catch(CacheException ex)
{
   // handle exception
}
```

Restrictions and Programming Pointers

This section covers restrictions and programming pointers when using the Java Object Cache.

- Do not share the `CacheAccess` object between threads. This object represents a user to the caching system. The `CacheAccess` object contains the current state of the user's access to the cache: what object is currently being accessed, what objects are currently owned, and so on. Trying to share the `CacheAccess` object is unnecessary and may result in unpredictable behavior.
- A `CacheAccess` object holds a reference to only one cached object at a time. If multiple cached objects are being accessed concurrently, then use multiple `CacheAccess` objects. For objects that are stored in memory, the consequences of not doing this are minor, because Java prevents the cached object from being garbage collected, even if the cache believes it is not being referenced. For disk objects, if the cache reference is not maintained, the underlying file could be removed by another user or by time-based invalidation, causing unexpected exceptions. To optimize resource management, keep the cache reference open as long as the cached object is being used.
- Always close a `CacheAccess` object when it is no longer being used. The `CacheAccess` objects are pooled. They acquire cache resources on behalf of the user. If the access object is not closed when it is not being used, then these resources are not returned to the pool and are not cleaned up until they are garbage collected by the JVM. If `CacheAccess` objects are continually allocated and not closed, then degradation in performance may occur.
- When local objects (objects that do not set the `Attributes.DISTRIBUTE` attribute) are saved to disk using the `CacheAccess.save()` method, they do not survive the termination of the process. By definition, local objects are visible only to the cache instance where they were loaded. If that cache instance goes away for

any reason, then the objects that it manages, including on disk, are lost. If an object must survive process termination, then both the object and the cache must be defined `DISTRIBUTE`.

- The cache configuration, also called the cache environment, is local to a cache; this includes the region, subregion, group, and object definitions. The cache configuration is not saved to disk or propagated to other caches. Define the cache configuration during the initialization of the application.
- If a `CacheAccess.waitForResponse()` or `CacheAccess.releaseOwnership()` method call times out, then you must call it again until it returns successfully. If `CacheAccess.waitForResponse()` does not succeed, then you must call `CacheAccess.cancelResponse` to free resources. If `CacheAccess.releaseOwnership()` doesn't succeed, then you must call `CacheAccess.releaseOwnership` with a timeout value of -1 to free resources.
- When a group or region is destroyed or invalidated, distributed definitions take precedence over local definitions. That is, if the group is distributed, then all objects in the group or region are invalidated or destroyed across the entire cache system, even if the individual objects or associated groups are defined as local. If the group or region is defined as local, then local objects within the group are invalidated locally; distributed objects are invalidated throughout the entire cache system.
- When an object or group is defined with the `SYNCHRONIZE` attribute set, ownership is required to load or replace the object. However, ownership is not required for general access to the object or to invalidate the object.
- In general, objects that are stored in the cache should be loaded by the system class loader that is defined in the `classpath` when the JVM is initialized, rather than by a user-defined class loader. Specifically, any objects that are shared between applications or can be saved or spooled to disk must be defined in the system `classpath`. Failure to do so can result in a `ClassNotFoundException` or a `ClassCastException`.
- On some systems, the open file descriptors can be limited by default. On these systems, you may need to change system parameters to improve performance. On UNIX systems, for example, a value of `1024` or greater can be an appropriate value for the number of open file descriptors.
- When configured in either local or distributed mode, at startup, one active Java Object Cache cache is created in a JVM process (that is, in the program running in the JVM that uses the Java Object Cache API).

Working with Disk Objects

The Java Object Cache can manage objects on disk as well as in memory.

This section covers the following topics:

- Local and Distributed Disk Cache Objects
- Adding Objects to the Disk Cache

Local and Distributed Disk Cache Objects

This section covers the following topics:

- Local Objects

- Distributed Objects

Local Objects

When operating in local mode, the cache attribute `isDistributed` is not set and all objects are treated as local objects (even when the `DISTRIBUTE` attribute is set for an object). In local mode, all objects in the disk cache are visible only to the Java Object Cache cache that loaded them, and they do not survive after process termination. In local mode, objects stored in the disk cache are lost when the process using the cache terminates.

Distributed Objects

If the cache attribute `isDistributed` is set to true, the cache will operate in distributed mode. Disk cache objects can be shared by all caches that have access to the file system hosting the disk cache. This is determined by the disk cache location configured. This configuration allows for better utilization of disk resources and allows disk objects to persist beyond the life of the Java Object Cache process.

Objects that are stored in the disk cache are identified using the concatenation of the path that is specified in the `diskPath` configuration property and an internally generated `String` representing the remaining path to the file. Thus, caches that share a disk cache can have a different directory structure, as long as the `diskPath` represents the same directory on the physical disk and is accessible to the Java Object Cache processes.

If a memory object that is saved to disk is also distributed, the memory object can survive the death of the process that spooled it.

Adding Objects to the Disk Cache

There are several ways to use the disk cache with the Java Object Cache, including:

- Automatically Adding Objects
- Explicitly Adding Objects
- Using Objects that Reside Only in Disk Cache

Automatically Adding Objects

The Java Object Cache automatically adds certain objects to the disk cache. Such objects can reside either in the memory cache or in the disk cache. If an object in the disk cache is needed, it is copied back to the memory cache. The action of spooling to disk occurs when the Java Object Cache determines that it requires free space in the memory cache. Spooling of an object occurs only if the `SPOOL` attribute is set for the object.

Explicitly Adding Objects

In some situations, you may want to force one or more objects to be written to the Java Object Cache disk cache. Using the `CacheAccess.save()` method, a region, subregion, group, or object is written to the disk cache. (If the object or objects are already in the disk cache, they are not written again.)

Note: Using `CacheAccess.save()` saves an object to disk even when the `SPOOL` attribute is not set for the object.

Calling `CacheAccess.save()` on a region, subregion, or group saves all the objects within the region, subregion, or group to the disk cache. During a `CacheAccess.save()` method call, if an object is encountered that cannot be written to disk, either because it is not serializable or for other reasons, then the event is recorded in the Java Object Cache log, and the save operation continues with the next object. When individual objects are written to disk, the write is synchronous. If a group or region is saved, then the write is performed as an asynchronous background task.

Using Objects that Reside Only in Disk Cache

Objects that you access only directly from disk cache are loaded into the disk cache by calling `CacheLoader.createDiskObject()` from the `CacheLoader.load()` method. The `createDiskObject()` method returns a `File` object that the application can use to load the disk object. If the attributes of the disk object are not defined for the disk object, then set them using the `createDiskObject()` method. The system manages local and distributed disk objects differently; the system determines if the object is local or distributed when it creates the object, based on the specified attributes.

Note: If you want to share a disk cache object between distributed caches in the same cache system, then you must define the `DISTRIBUTE` attribute when the disk cache object is created. This attribute cannot be changed after the object is created.

When `CacheAccess.get()` is called on a disk object, the full path name to the file is returned, and the application can open the file, as needed.

Disk objects are stored on a local disk and accessed directly from the disk by the application using the Java Object Cache. Disk objects can be shared by all Java Object Cache processes, or they can be local to a particular process, depending on the setting for the `DISTRIBUTE` attribute (and the mode that the Java Object Cache is running in, either distributed or local).

Example 7–17, "Creating a Disk Object in a CacheLoader" shows a loader object that loads a disk object into the cache.

Example 7–17 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 7–18, "Application Code that Uses a Disk Object" 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 7–17, "Creating a Disk Object in a CacheLoader" as the default loader for the region.

Example 7–18   Application Code that Uses a Disk Object

```
import oracle.ias.cache.*;

try
{
   FileInputStream in;
   File file;
   String filePath;
   CacheAccess cacc = CacheAccess.getAccess("Stock-Market");

   filePath = (String)cacc.get("file object");
   file = new File(filePath);
   in = new FileInputStream(filePath);
   in.read(buf);

// do something interesting with the data
   in.close();
   cacc.close();
}
catch (Exception ex)
{
// handle exception
}
```

Working with StreamAccess Objects

A `StreamAccess` object is accessed as a stream and automatically loaded to the disk cache. The object is loaded as an `OutputStream` and read as an `InputStream`. Smaller `StreamAccess` objects can be accessed from memory or from the disk cache; larger `StreamAccess` objects are streamed directly from the disk. The Java Object Cache automatically determines where to access the `StreamAccess` object, based on the size of the object and the capacity of the cache.

The user is always presented with a stream object, an `InputStream` for reading and an `OutputStream` for writing, regardless of whether the object is in a file or in memory. The `StreamAccess` object allows the Java Object Cache user to always access the object in a uniform manner, without regard to object size or resource availability.

Creating a StreamAccess Object

To create a `StreamAccess` object, call the `CacheLoader.createStream()` method from the `CacheLoader.load()` method when the object is loaded into the cache. The `createStream()` method returns an `OutputStream` object. Use the `OutputStream` object to load the object into the cache.

If the attributes have not already been defined for the object, then set them using the `createStream()` method. The system manages local and distributed disk objects differently; the determination of local or distributed is made when the system creates the object, based on the attributes.

Note: If you want to share a `StreamAccess` object between distributed caches in the same cache system, then you must define the `DISTRIBUTE` attribute when the `StreamAccess` object is created. You cannot change this attribute after the object is created.

Example 7–19, "Creating a StreamAccess Object in a Cache Loader" shows a loader object that loads a `StreamAccess` object into the cache.

Example 7–19 Creating a StreamAccess Object in a Cache Loader

```
import oracle.ias.cache.*;

class YourObjectLoader extends CacheLoader
{
   public Object load(Object handle, Object args) {
     OutputStream = out;
     Attributes attr = new Attributes();
     attr.setFlags(Attributes.DISTRIBUTE);

     try
     {
        out = createStream(handle, attr);
        out.write((byte[])getInfofromsomewhere());
     }
     catch (Exception ex) {
        // translate exception to CacheException, and log exception
        throw exceptionHandler("exception in write", ex)
     }
     return out;
     }
}
```

Working with Pool Objects

A pool object is a special cache object that the Java Object Cache manages. A pool object contains a set of identical object instances. The pool object itself is a shared object; the objects within the pool are private objects that the Java Object Cache manages. Users access individual objects within the pool with a check out, using a pool access object, and then return the objects to the pool when they are no longer needed.

This section covers the following topics:

- Creating Pool Objects
- Using Objects from a Pool

- Implementing a Pool Object Instance Factory
- Pool Object Affinity

Creating Pool Objects

To create a pool object, use `CacheAccess.createPool()`. The `CreatePool()` method takes as arguments:

- A `PoolInstanceFactory`
- An `Attributes` object
- Two integer arguments

The integer arguments specify the maximum pool size and the minimum pool size. By supplying a group name as an argument to `CreatePool()`, a pool object is associated with a group.

Attributes, including `TimeToLive` or `IdleTime`, can be associated with a pool object. These attributes can be applied to the pool object itself, when specified in the attributes set with `CacheAccess.createPool()`, or they can be applied to the objects within the pool individually.

Using `CacheAccess.createPool()`, specify minimum and maximum sizes with the integer arguments. Specify the minimum first. It sets the minimum number of objects to create within the pool. The minimum size is interpreted as a request rather than a guaranteed minimum. Objects within a pool object are subject to removal from the cache due to lack of resources, so the pool can decrease the number of objects below the requested minimum value. The maximum pool size puts a hard limit on the number of objects available in the pool.

Note: Pool objects and the objects within a pool object are always treated as local objects.

Example 7–20, "Creating a Pool Object" shows how to create a pool object.

Example 7–20 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 7–21, "Using a PoolAccess Object" 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 7–21 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 7–22, "Implementing Pool Instance Factory Methods" shows the calls required when implementing a `PoolInstanceFactory`.

Example 7–22 Implementing Pool Instance Factory Methods

```
import oracle.ias.cache.*;
   public class MyPoolFactory implements PoolInstanceFactory
   {
       public Object createInstance()
      {
        MyObject obj = new MyObject();
        obj.init();
        return obj;
       }
       public void destroyInstance(Object obj)
       {
           ((MyObject)obj).cleanup();
       }
   }
```

Pool Object Affinity

Object pools are a collection of serially reusable objects. A user "checks out" an object from the pool to perform a function, then "checks in" the object back to the pool when done. During the time the object is checked out, the user has exclusive use of that object instance. After the object is checked in, the user gives up all access to the object. While the object is checked out, the user can apply temporary modifications to the pool object (add state) to allow it to execute the current task. Since some cost is incurred to add these modifications, it would be beneficial to allow the user to, whenever possible, get the same object from the pool with the modifications already in place. Since the 9.0.2 version of the Java Object Cache, the only way to do this was never to check in the object, which would then defeat the purpose of the pool. To support the pool requirement described in this paragraph, the functionality described in the following two paragraphs has been added to the pool management of the Java Object Cache.

Objects checked into the pool using the `returnToPool` method of the `PoolAccess` object maintain an association with the last `PoolAccess` object that referenced the object. When the `PoolAccess` handle requests an object instance, the same object it had previously is returned. This association will be terminated if the `PoolAccess` handle is closed, or the `PoolAccess.release` method is called, or the object is given to another user. Before the object is given to another user, a callback is made to determine if the user is willing to give up the association with the object. If the user is not willing to dissolve the association, then the new user is not given access to the object. The interface `PoolAffinityFactory` extends the interface `PoolInstanceFactory`, adding the callback method `affinityRelease`. This method returns `true` if the association can be broken, and `false` otherwise.

If the entire pool is invalidated, the `affinityRelease` method is not called. Object instance cleanup is then performed with the `PoolInstanceFactory.instanceDestroy` method.

Running in Local Mode

When running in local mode, the Java Object Cache does not share objects or communicate with any other caches running locally on the same system or remotely across the network. Object persistence across system shutdowns or program failures is not supported when running in local mode.

By default, the Java Object Cache runs in local mode, and all objects in the cache are treated as local objects. When the Java Object Cache is configured in local mode, the cache ignores the `DISTRIBUTE` attribute for all objects.

Running in Distributed Mode

In distributed mode, the Java Object Cache can share objects and communicate with other caches running either locally on the same system or remotely across the network. Object updates and invalidations are propagated between communicating caches. Distributed mode supports object persistence across system shutdowns and program failures.

This section covers the following topics:

- Configuring Properties for Distributed Mode
- Using Distributed Objects, Regions, Subregions, and Groups
- Accessing Objects in Remote Caches

- Accessing Objects in Remote Caches
- Sharing Cached Objects in an OC4J Servlet
- Using User-Defined Class Loaders
- HTTP and Security for Distributed Cache

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 7-22 describes how to do this.

Setting the discoveryAddress Configuration Property

In distributed mode, invalidations, destroys, and replaces are propagated through the messaging system of the cache. The messaging system requires a known host name and port address to allow a cache to join the cache system when it is first initialized. Use the `discoverer` attribute in the communication section in the `javacache.xml` file to specify a list of host name and port addresses.

By default, the Java Object Cache sets the `discoverer` to the value `:12345` (this is equivalent to `localhost:12345`). To eliminate conflicts with other software on the site, have your system administrator set the `discoveryAddress`.

If the Java Object Cache spans systems, then configure multiple discoverer entries, with one *`hostname:port`* pair specified for each node. Doing this avoids any dependency on a particular system being available or on the order the processes are started. Also see "Java Object Cache Configuration" on page 7-22.

Note: All caches cooperating in the same cache system must specify the same set of host name and port addresses. The address list, set with the discoverer attributes, defines the caches that make up a particular cache system. If the address lists vary, then the cache system could be partitioned into distinct groups, resulting in inconsistencies between caches.

Using Distributed Objects, Regions, Subregions, and Groups

When the Java Object Cache runs in distributed mode, individual regions, subregions, groups, and objects can be either local or distributed. By default, objects, regions, subregions, and groups are defined as local. To change the default local value, set the `DISTRIBUTE` attribute when the object, region, or group is defined.

A distributed cache can contain both local and distributed objects.

Several attributes and methods in the Java Object Cache allow you to work with distributed objects and control the level of consistency of object data across the caches. Also see "Accessing Objects in Remote Caches" on page 7-51.

Using the REPLY Attribute with Distributed Objects

When updating, invalidating, or destroying objects across multiple caches, it may be useful to know when the action has completed at all the participating sites. Setting the `REPLY` attribute causes all participating caches to send a reply to the originator when a requested action has completed for the object. The `CacheAccess.waitForResponse()` method allows the user to block until all remote operations have completed.

To wait for a distributed action to complete across multiple caches, use `CacheAccess.waitForResponse()`. To ignore responses, use the `CacheAccess.cancelResponse()` method, which frees the cache resources used to collect the responses.

Both `CacheAccess.waitForResponse()` and `CacheAccess.cancelResponse()` apply to all objects that are accessed by the `CacheAccess` object. This feature allows the application to update several objects, then wait for all the replies.

Example 7–23, "Distributed Caching Using Reply" illustrates how to set an object as distributed and handle replies when the `REPLY` attribute is set. In this example, you can also set the attributes for the entire region. Additionally, you can set attributes for a group or individual object, as appropriate for your application.

Example 7–23   Distributed Caching Using Reply

```
import oracle.ias.cache.*;

CacheAccess cacc;
String     obj;
Attributes attr = new Attributes ();
MyLoader   loader = new MyLoader();

// mark the object for distribution and have a reply generated
// by the remote caches when the change is completed

attr.setFlags(Attributes.DISTRIBUTE|Attributes.REPLY);
attr.setLoader(loader);

CacheAccess.defineRegion("testRegion",attr);
cacc = CacheAccess.getAccess("testRegion"); // create region with
  //distributed attributes

obj = (String)cacc.get("testObject");
cacc.replace("testObject", obj + "new version"); // change will be
  // propagated to other caches

cacc.invalidate("invalidObject"); // invalidation is propagated to other caches

try
{
// wait for up to a second,1000 milliseconds, for both the update
// and the invalidate to complete
    cacc.waitForResponse(1000);

catch (TimeoutException ex)
{
   // tired of waiting so cancel the response
   cacc.cancelResponse();
}
cacc.close();
}
```

Using SYNCHRONIZE and SYNCHRONIZE_DEFAULT

When updating objects across multiple caches, or when multiple threads access a single object, you can coordinate the update action. Setting the `SYNCHRONIZE` attribute enables synchronized updates, and requires an application to obtain ownership of an object before the object is loaded or updated.

The `SYNCHRONIZE` attribute also applies to regions, subregions, and groups. When the `SYNCHRONIZE` attribute is applied to a region, subregion, or group, ownership of the region, subregion, or group must be obtained before an object can be loaded or replaced in the region, subregion, or group.

Setting the `SYNCHRONIZE_DEFAULT` attribute on a region, subregion, or group applies the `SYNCHRONIZE` attribute to all the objects within the region, subregion, or group. Ownership must be obtained for the individual objects within the region, subregion, or group before they can be loaded or replaced.

To obtain ownership of an object, use `CacheAccess.getOwnership()`. After ownership is obtained, no other `CacheAccess` instance is allowed to load or replace the object. Reads and invalidation of objects are not affected by synchronization.

After ownership has been obtained and the modification to the object is completed, call `CacheAccess.releaseOwnership()` to release the object. `CacheAccess.releaseOwnership()` waits up to the specified time for the updates to complete at the remote caches. If the updates complete within the specified time, ownership is released; otherwise, a `TimeoutException` is thrown. If the method times out, call `CacheAccess.releaseOwnership()` again. `CacheAccess.releaseOwnership()` must return successfully for ownership to be released. If the timeout value is `-1`, then ownership is released immediately, without waiting for the responses from the other caches.

Example 7–24, "Distributed Caching Using SYNCHRONIZE and SYNCHRONIZE_DEFAULT" illustrates distributed caching using `SYNCHRONIZE` and `SYNCHRONIZE_DEFAULT`.

Example 7–24 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();
}
```

Accessing Objects in Remote Caches

The Cache Service provides four methods to access the contents of remote caches. The protected methods `CacheLoader.getFromRemote()` and `CacheLoader.netSearch()` can only be accessed from `CacheLoader.load()`. `getFromRemote` searches a specified remote cache for a specified object to load into the local cache, while `netSearch` searches all caches in the system for a specified object to load into the local cache. `CacheAccess.replaceRemote()` replaces a specified object in a specified remote cache. `CacheAccess.getAllCached()` does not change the contents of caches, but returns a vector of all instances of a specified object in all caches in the distributed system.

If the object exists in the remote cache, `getFromRemote` will load it into the local cache and return a reference to it. If the `useRemoteTtl` boolean parameter is set, the local object attributes such as time to live or idle time will be set to the remote object values. `netSearch` is used in the same way, except there is no need to specify any particular remote cache because it searches all remote caches.

The following examples demonstrate using `getFromRemote/netSearch` in implementing `CacheLoader.load()`.

```
import oracle.ias.cache.*;
   class MyLoader extends CacheLoader
   {
      public Object load (Object handle, Object args) throws CacheException
      {
         Object obj = null;
         try
         {
            obj = getFromRemote(handle, (CacheAddress)args, 10000, true); // use
remote ttl
         }
         catch (CacheException ex)
         {
            throw ex;
         }
         return obj;
      }
   }
****************************************
import oracle.ias.cache.*;
   class MyLoader extends CacheLoader
   {
      public Object load (Object handle, Object args) throws CacheException
      {
         Object obj = null;
         try
         {
            obj = netSearch(handle, 10000, true);  // use remote ttl
         }
         catch (CacheException ex)
         {
            throw ex;
         }
         return obj;
      }
   }
```

While `getFromRemote` can be used with memory, disk, and stream objects, `netSearch` can only be used with memory objects.

When calling `replaceRemote`, the contents in the remote target cache take priority. If the object being replaced is marked as a local object in the remote cache, or if the remote cache is full, then the operation will fail with an exception. If the target address is the local cache address, then the operation will be treated as a localized replace. If the specified object does not exist in the remote cache, then the operation will be treated as a remote put. `replaceRemote` is also limited to memory objects.

The following example demonstrates using `replaceRemote`. After execution, a `cacc.get()` in `Cache1.java` on object `"obj"` will return the string `"object 1 replacement"`.

```
Cache1.java
-----------
import oracle.ias.cache.*;
   Attributes attr = new Attributes();
   attr.setFlags(Attributes.DISTRIBUTE);
   CacheAccess.defineRegion("region", attr);
   CacheAccess cacc = CacheAccess.getAccess("region");
   cacc.put("obj", "object 1");

Cache2.java
-----------
import oracle.ias.cache.*;
   Attributes attr = new Attributes();
   attr.setFlags(Attributes.DISTRIBUTE);
   CacheAccess.defineRegion("region", attr);
   CacheAccess cacc = CacheAccess.getAccess("region");
   CacheAddress cache1 = getCache1CacheAddress();
   cacc.replaceRemote("obj", "object 1 replacement", cache1);
```

Unlike `getFromRemote/netSearch` and `replaceRemote`, `getAllCached` does not change the contents of any cache. This method can deal with memory, stream, or disk objects, with the limitation that it can only deal with one type each call. In other words, when calling `getAllCached` on an object, that object's basic type should be consistent across all caches in the distributed system. If no cache currently has an object matching that name in cache, an empty vector will be returned. Depending on whether you care about any possible exceptions thrown at remote caches while trying to locate the object, you can set the optional boolean parameter `ignoreRemoteEx`. By default, it is set to `true`, and all remote exceptions are ignored.

The following example demonstrates using `getAllCached`.

```
Cache1.java
-----------
import oracle.ias.cache.*;
   Attributes attr = new Attributes();
   attr.setFlags(Attributes.DISTRIBUTE);
   CacheAccess.defineRegion("region", attr);
   CacheAccess cacc = CacheAccess.getAccess("region");
   MemObjType1 obj = new MemObjType1();
   cacc.put("obj", obj);

Cache2.java
-----------
import oracle.ias.cache.*;Attributes attr = new Attributes();
   attr.setFlags(Attributes.DISTRIBUTE);
   CacheAccess.defineRegion("region", attr);
   CacheAccess cacc = CacheAccess.getAccess("region");
   MemObjType2 obj = new MemObjType2();
   cacc.put("obj", obj);

...

Cache9.java
-----------
import oracle.ias.cache.*;Attributes attr = new Attributes();
   attr.setFlags(Attributes.DISTRIBUTE);
   CacheAccess.defineRegion("region", attr);
   CacheAccess cacc = CacheAccess.getAccess("region");
   MemObjType9 obj = new MemObjType9();
   cacc.put("obj", obj);

Cache10.java
------------
import oracle.ias.cache.*;Attributes attr = new Attributes();
   attr.setFlags(Attributes.DISTRIBUTE);
   CacheAccess.defineRegion("region", attr);
   CacheAccess cacc = CacheAccess.getAccess("region");
   Vector objects = cacc.getAllCached("obj");
```

Note that all objects that need to be communicated between caches must be Serializable objects.

Cached Object Consistency Levels

Within the Java Object Cache, each cache manages its own objects locally, within its JVM process. In distributed mode, when using multiple processes or when the system is running on multiple sites, a copy of an object can exist in more than one cache.

The Java Object Cache allows you to specify the consistency level that is required between copies of objects that are available in multiple caches. The consistency level that you specify depends on the application and the objects being cached. The supported levels of consistency vary, from none to all copies of objects being consistent across all communicating caches.

Setting object attributes specifies the level of consistency. The consistency between objects in different caches is categorized into the following four levels:

- Using Local Objects (No consistency requirements)

- Propagating Changes Without Waiting for a Reply
- Propagating Changes and Waiting for a Reply
- Serializing Changes Across Multiple Caches

Using Local Objects

If there are no consistency requirements between objects in distributed caches, then define an object as a local object. (When `Attributes.DISTRIBUTE` is unset, this specifies a local object.) Local is the default setting for objects. For local objects, all updates and invalidation are visible to only the local cache.

Propagating Changes Without Waiting for a Reply

To distribute object updates across distributed caches, define an object as distributed by setting the `DISTRIBUTE` attribute. All modifications to distributed objects are broadcast to other caches in the system. Using this level of consistency does not control or specify when an object is loaded into the cache or updated, and does not provide notification as to when the modification has completed in all caches.

Propagating Changes and Waiting for a Reply

To distribute object updates across distributed caches and wait for the change to complete before continuing, set the object's `DISTRIBUTE` and `REPLY` attributes. When you set these attributes, notification occurs when a modification has completed in all caches. When you set `Attributes.REPLY` for an object, replies are sent back to the modifying cache when the modification has been completed at the remote site. These replies are returned asynchronously—that is, the `CacheAccess.replace()` and `CacheAccess.invalidate()` methods do not block. Use the `CacheAccess.waitForResponse()` method to wait for replies and block.

Serializing Changes Across Multiple Caches

To use the highest level of consistency of the Java Object Cache, set the appropriate attributes on the region, subregion, group, or object to make objects act as synchronized objects.

When you set `Attributes.SYNCHRONIZE_DEFAULT` on a region, subregion, or group, it sets the `SYNCHRONIZE` attribute for all the objects within the region, subregion, or group.

When you set `Attributes.SYNCHRONIZE` on an object, it forces applications to obtain ownership of the object before the object can be loaded or modified. Setting this attribute effectively serializes write access to objects. To obtain ownership of an object, use the `CacheAccess.getOwnership()` method. When you set the `Attributes.SYNCHRONIZE` attribute, notification is sent to the owner when the update is completed. Use `CacheAccess.releaseOwnership()` to block until any outstanding updates have completed and the replies are received. This releases ownership of the object so that other caches can update or load the object.

Note: Setting `Attributes.SYNCHRONIZE` for an object is not the same as setting synchronized on a Java method. With `Attributes.SYNCHRONIZE` set, the Java Object Cache forces the cache to serialize creates and updates of the object, but does not prevent the Java programmer from obtaining a reference to the object and then modifying the object.

When using this level of consistency, with `Attributes.SYNCHRONIZE`, the `CacheLoader.load()` method calls `CacheLoader.netSearch()` before loading the object from an external source. Calling `CacheLoader.netSearch()` in the load method tells the Java Object Cache to search all other caches for a copy of the object. This process prevents different versions of the object from being loaded into the cache from an external source. Proper use of the `SYNCHRONIZE` attribute, along with the `REPLY` attribute and the invalidate method, supports consistency of objects across the cache system

Sharing Cached Objects in an OC4J Servlet

To take advantage of the distributed functionality of the Java Object Cache or to share a cached object among servlets, some minor modification to an application's deployment may be necessary. Any user-defined objects that will be shared among servlets or distributed among JVMs must be loaded by the system class loader. By default, objects that are loaded by a servlet are loaded by the context class loader. These objects are visible only to the servlets within the context that loaded them. The object definition is not available to other servlets or to the cache in another JVM. If the object is loaded by the system class loader, the object definition is available to other servlets and to the cache on other JVMs.

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 *`OLE_HOME`*`/javacache/sharedobjects/classes`, or add it to the JAR file *`OLE_HOME`*`/javacache/cachedobjects/share.jar`. Both the `classes` directory and the `share.jar` file have been included in the manifest for `cache.jar`.

Using User-Defined Class Loaders

You can place objects in the cache that require user-defined class loaders. The Cache Service supports user-defined class loaders by providing two methods: `Attributes.setClassLoader()` and `Attributes.getClassLoader()`. Once you have set a region or a group to use a user-defined class loader, all objects under that region or group are loaded using this class loader (by inheriting attributes). This attribute does not apply to objects that are not region or group, and will be ignored if set.

The following example demonstrates setting user-defined class loader. Object A is loaded using MyClassLoader.

```
import oracle.ias.cache.*;
import java.lang.ClassLoader;
Classloader loader = this.getClass().getClassLoader();
CacheAttributes cAttr = new CacheAttributes();
Attributes attr = new Attributes();
CacheAccess cacc;
Cache.init(cAttr);
attr.setClassLoader(loader);
CacheAccess.defineRegion("region A", attr);
cacc = CacheAccess.getAccess("region A");
cacc.get("object A")
```

HTTP and Security for Distributed Cache

This section discusses HTTP and security for distributed cache.

HTTP

By default, the Cache Service now uses the HTTP protocol to communicate between caches. For backward compatibility, the Cache Service will continue to support the proprietary TCP protocol to communicate between caches. While the proprietary protocol is kept for compatibility reasons, some of the newer functionalities are implemented exclusively for HTTP. In particular, in a distributed cache system, when getting disk or stream objects from remote caches, HTTP mode is required. `CacheAccess.getAllCached()`, `CacheLoader.getFromRemote()`, and `CacheLoader.netSearch()` are three example operations where HTTP mode is required for dealing with disk and stream objects. To enable the proprietary mode, you must use the `CacheAttributes.`setTransport() method, `CacheAttributes.setTransport(NamedCacheAttributes.TCP).`

All caches in the distributed system must be using the same communication protocol.

The following examples show two ways to enable TCP mode.

Example 1 - Enabling TCP in `cache_attributes.xml`

```
--------------------
<?xml version="1.0" encoding="UTF-8"?>
<cache-configuration xmlns="http://www.oracle.com/oracle/ias/cache/configuration"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <communication>
      <isDistributed>true</isDistributed>
      <transport>TCP</transport>
   </communication>
</cache-configuration>
```

```
import oracle.ias.cache.*;
Cache.open("cache_attributes.xml");
```

Example 2 - Enabling TCP in code:

```
import oracle.ias.cache.*;
CacheAttributes cAttr = new CacheAttributes();
cAttr.distribute = true;
cAttr.setTransport(NamedCacheAttributes.TCP;
Cache.init(cAttr);
```

SSL

For secure communication between caches, the Cache Service supports the SSL protocol. SSL is only available with the HTTP protocol. The JDK keytool program can be used to generate certificates and set up the keystore, as documented on Sun's J2SE 1.4.2 Key and Certificate Management Tool web page. The same key pair and certificate used for OC4J can be used for the Cache Service.

To use SSL, you must enable it on all of the caches within a distributed system.

After setting up the keystore, you need to tell the cache where the keystore is with this command:

```
java -jar $OLE_HOME/javacache/lib/cache.jar sslconfig <cache_attributes.xml>
<keystore_file> <password>
```

where

- `$OLE_HOME` is the home directory of the Oracle IAS instance.
- `cache_attributes.xml` is your cache configuration file.
- `keystore_file` is the full path to your keystore file as generated by keytool.
- `password` is the password you used in keytool to generate the key pair.

This generates an SSL configuration file to be used by the cache, where the name of the file is as specified in cache_attributes.xml. In addition, you need to set CacheAttributes.isSSLEnabled to true.

The following examples show two ways to enable SSL:

Example 1 - Enabling SSL in `cache_attributes.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<cache-configuration
 xmlns="http://www.oracle.com/oracle/ias/cache/configuration"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <communication>
      <isDistributed>true</isDistributed>
      <useSSL>true</useSSL>
      <keyStore>.keyStore</keyStore>
      <sslConfigFile>.sslConfig</sslConfigFile>
   </communication>
</cache-configuration>


import oracle.ias.cache.*;
Cache.open("cache_attributes.xml");
```

Example 2 - Enabling SSL in code:

```
import oracle.ias.cache.*;
CacheAttributes cAttr = new CacheAttributes();
cAttr.distribute = true;
cAttr.isSSLEnabled = true;
cAttr.keyStoreLocation = ".keyStore";
cAttr.sslConfigFilePath = ".sslConfig";
Cache.init(cAttr);
```

Two caches must be using the same set of keys to communicate with each other. If the caches in a system reside on multiple machines, then you need to copy the keystore file to all machines and run the `java -jar` ... command for every cache configuration file in the system.

Firewall

To make a distributed cache system work across a firewall, the current workaround is to enable a set of outbound TCP ports at the firewall and to define them in `cache_attributes.xml`.

For example, `cache_attributes.xml` might look something like this if the ports are within the range of `7100` to `7199`:

```
cache_attributes.xml
--------------------
<?xml version = '1.0' encoding = 'UTF-8'?>
<cache-configuration xmlns="http://www.oracle.com/oracle/ias/cache/configuration"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <communication>
      <isDistributed>true</isDistributed>
      <useSSL>false</useSSL>
      <sslConfigFile>.sslConfig</sslConfigFile>
      <port lower="7100" upper="7199"/>
      <discoverer discovery-port="7100" original="true" xmlns=""/>
   </communication>
</cache-configuration>
```

Make sure that the `discovery-port` is within the range specified.

Restricting Incoming Connections

For systems that are configured with more than one address to support multiple network subnets (private and public, for example), you can specify a configuration element, `localAddress`, in `cache_attributes.xml` to restrict incoming connections to a specified local address. By default, the distributed cache system will bind the listener socket to the primary host address returned by the operating system. If `localAddress` is specified, however, the cache will bind the listener socket to the specified address. The value specified for `localAddress` must be a fully qualified hostname or IP address. For example:

```
cache_attributes.xml
--------------------
<?xml version = '1.0' encoding = 'UTF-8'?>
<cache-configuration xmlns="http://www.oracle.com/oracle/ias/cache/configuration"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <communication>
      <isDistributed>true</isDistributed>
      <localAddress>123.456.78.90</localAddress>
      <discoverer discovery-port="7100" original="true" xmlns=""/>
   </communication>
</cache-configuration>
```

or

```
cache_attributes.xml
--------------------
<?xml version = '1.0' encoding = 'UTF-8'?>
<cache-configuration xmlns="http://www.oracle.com/oracle/ias/cache/configuration"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <communication>
      <isDistributed>true</isDistributed>
      <localAddress>computer.oracle.com</localAddress>
      <discoverer discovery-port="7100" original="true" xmlns=""/>
   </communication>
</cache-configuration>
```

In the latter case, even if the IP underneath a virtual hostname changes, JOC will remain unaffected.

Monitoring and Debugging

Besides `Cache.listCacheObjects()` and `Cache.dump()`, the Cache Service provides additional methods to reflect the current status of the cache and of the regions, groups, and individual objects within the cache. These methods can be found in the classes `CacheAccess`, `Cache`, `ObjectStatus`, and `AggregateStatus`.

The methods under `Cache` reflect the cache's own status. `getActiveHostInfo` returns an array of `CacheHostInfo` objects for all active caches in a cache system. `getCacheSize` estimates total space (bytes) taken up by memory objects in the cache. `getDistributedDiskCacheSize` and `getLocalDiskCacheSize` estimate total space (bytes) taken up by objects in the distributed and local disk caches, respectively. `getObjectCount` returns the current total number of objects in the cache.

The methods under `CacheAccess` reflect region, group, and individual object status. `listNames` names all objects under the region. `listObjects` names all objects under the region and also provides access to them. `listRegions` names all sub-regions under the region. These three methods are not recursive. For example, `listRegions` does not list the sub-regions under the sub-regions of the region.

`CacheAccess.getStatus()` reflects more detailed status information for a named individual object or group under the region in the form of an `ObjectStatus` object. This includes the cached object's access count, time of creation, size on disk (if stored on disk), last time of access, loading time (ms), priority (as set by the object's creator), and size in cache (bytes). If no object or group name is specified, `getStatus` returns the status of the region.

`CacheAccess.getAggregateStatus()`, on the other hand, returns overall statistics for a named group or region (sub-region) in the form of an `AggregateStatus` object. `AggregateStatus` reflects the low, average, and high values of attributes of the objects under the region or group. These attributes include access count, time of creation, last time of access, loading time, priority, and size in cache. In addition, the `AggregateStatus` object also includes the total object count for the region or group. Reflection methods in the `AggregateStatus` class allows you to access all of these numbers individually.

The Cache Service automatically compiles the information reflected by `getAggregateStatus` during every clean interval. To obtain the latest information, you need to call `Cache.updateStats()` before calling `getAggregateStatus`.

Here is an example of using `getAggregateStatus`

```
import oracle.ias.cache.*;
import java.util.Date;
import java.io.*;

CacheAccess cacc;

// create objects, load objects, etc.
...

AggregateStatus aggStats;
long            avgCreateTime;
Date            avg;

Cache.updateStats();
aggStats = cacc.getAggregateStatus();
avgCreateTime = aggStats.getCreateTime(AggregateStatus.AVG);
avg = new Date(avgCreateTime);

System.out.println("average creation time: " + avg);
```

JOC supports DMS by providing the following metrics at the process-wide cache level:

- Memory Size - the total number of bytes of memory consumed by objects in the cache. The accuracy of this number depends on whether `Attributes.MEASURE` is set for the objects and, if not set, then whether the object sizes specified by the user are accurate. If `MEASURE` is not set and the explicitly set size is not accurate, then memory size will not be accurate.
- Memory Object Count - the total number of objects in the cache.
- Disk Size - the total number of bytes of disk space consumed by objects in the cache.
- Worker Thread Count - the total number of background worker threads. The cache spawns these threads to execute background routine tasks and to respond to remote cache requests.
- Task Count - the current number of background asynchronous tasks. These include tasks such as spooling memory objects to disk and cleaning the cache periodically.
- Response Q Size - the current size of the response queue. The response queue keeps track of response objects that are used to manage responses to requests made to other caches in the system.
- Time Q Size - the current size of the time queue. The time queue is a sorted list that keeps track of expiration times of group/region objects.

At the region level, JOC tracks the following metrics for DMS. Disk metrics are only tracked if `diskcache` is enabled.:

- Memory Size - the total number of bytes of memory consumed by objects in the region. The accuracy of this number depends on whether `Attributes.MEASURE` is set for the objects and, if not set, then whether the object sizes specified by the user are accurate. If `MEASURE` is not set and the explicitly set size is not accurate, then memory size will not be accurate.
- Memory Object Count - the total number of objects in the region.
- Memory Average Load Time - the average load time for objects in the region.
- Memory Object Access Count - the total number of accesses of objects in the region.
- Disk Size - the total number of bytes of disk space consumed by objects in the region.
- Disk Count - the total number of disk objects in the region.
- Disk Average Load Time - the average load time for disk objects in the region.

CacheWatchUtil

By default, the Cache Service provides the `CacheWatchUtil` cache monitoring utility that can display current caches in the system, display a list of cached objects, display caches' attributes, reset cache logger severity, dump cache contents to the log, and so on.

To invoke `CacheWatchUtil`, while caches are running, type one of the following commands:

```
java oracle.ias.cache.CacheWatchUtil [-config=cache_config.xml] [-help]
```

or

```
java -jar $OLE_HOME/javacache/lib/cache.jar watch [-config=cache_config.xml]
[-help]
```

where "`-config=`" and "`-help`" are optional parameters, and `cache_config.xml` is a cache configuration file.

- "`help`" gives you a list of commands you can invoke in the cache watcher. Among these commands,
- "`set severity=<level> [CacheId]`" sets logger severity level for a particular cache. The levels are:
  - `-1` off
  - `0` fatal
  - `3` error
  - `4` default
  - `6` warning
  - `7` trace
  - `10` info
  - `15` debug
- "`set timeout=<value>`" sets group communication timeout for the cache system to value.
- "`dump [CacheId]`" dumps the contents of a particular cache to the log file.
- "`invalidate`" invalidates all objects in the cache system.
- "`destroy`" destroys all objects in the cache system (include memory, stream, and disk).

Typing "get `config [CacheId]`" returns the cache configuration information for a particular cache. You can retrieve remote cache configurations for verification, as shown in the following example.

```
cache> get config 3
ache 3 at localhost:53977
distribute = true
version = 10.1.3
max objects = 200
max cache size = 48
diskSize = 32
diskPath = <disk_path>
clean interval = 3
LogFileName = <log_file_name>
Logger = MyCacheLogger
Log severity = 3
cache address list = [127.0.0.1:22222, pos=-1, uid=0, orig, name=, pri=0
]
```

Typing "`list caches`" or "`lc`" lists all of the active caches in the system. The cache watcher also occupies a spot on the list, as shown in the following example. The UID column displays every cache's ID. The cache watcher does not detect caches that have been configured but are not active.

```
cache> lc
```

```
Current coordinator: [127.0.0.1:53957, pos=0, uid=0, tag=27979955, pri=0]
#       UID     CacheAddress
-       ---     ---------------------
1       0       localhost:53957
2       1       localhost:53965
3       2       localhost:53974
4       3       localhost:53977
5       4       localhost:53980
6       5       localhost:53997 <-- this cache watcher
```

Typing

`"list objects [CacheId] [region=<region>] [sort=<0...7>]"`

or

`"lo [CacheId] [region=<region>] [sort=<0...7>]"`

lists all objects in a specified cache, under a specified region, and in the order specified by the sort option. The sort options are:

- `0` by region name
- `1` by object name
- `2` by group name
- `3` by object type
- `4` by valid status
- `5` by reference count
- `6` by access count
- `7` by expiration

Without any options, `lo` lists all objects in all caches without sorting. The following example shows `lo` for cache 3, A-Region, sorted by object name. Columns have been adjusted to improve example readability.

```
cache> lo 3 region=A-Region sort=1
Cache 3 at localhost:53977
 REGION     OBJNAME    GROUP      TYPE   REFCNT   ACCCNT   EXPIRE      VALID   LOCK
--------   ---------  -------    ------ -------- -------- --------    ------- ------
[A-Region] [A-Group]  [A-Region] Group  0        1        None        true    null
[A-Region] [A-Region] [null]     Region 0        4        295 Seconds true    null
[A-Region] [B-Group]  [A-Region] Group  0        3        None        true    null
[A-Region] [bar]      [B-Group]  Loader 0        1        None        true    null
```

Finally, typing "`groupdump`" dumps all group communication information for all caches to the log file. It is unlikely that you will need to use this command or that you will find its output useful, but in the event of group communication errors, technical support might ask you to supply the information for problem diagnosis.

XML Schema for Cache Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.oracle.com/oracle/ias/cache/configuration"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns="http://www.oracle.com/oracle/ias/cache/configuration"
elementFormDefault="qualified" attributeFormDefault="unqualified">
   <xs:element name="cache-configuration" type="CacheConfigurationType">
      <xs:annotation>
         <xs:documentation>Oracle JavaCache implementation</xs:documentation>
      </xs:annotation>
   </xs:element>
   <xs:complexType name="CacheConfigurationType">
      <xs:sequence>
         <xs:element name="logging" type="loggingType" minOccurs="0"/>
         <xs:element name="communication" type="communicationType" minOccurs="0"/>
         <xs:element name="persistence" type="persistenceType" minOccurs="0"/>
         <xs:element name="preload-file" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
         <xs:element name="max-objects" type="xs:positiveInteger" default="1000" minOccurs="0"/>
         <xs:element name="max-size" type="xs:positiveInteger" default="1000" minOccurs="0"/>
         <xs:element name="clean-interval" type="xs:positiveInteger" default="60" minOccurs="0"/>
         <xs:element name="ping-interval" type="xs:positiveInteger" default="60" minOccurs="0"/>
         <xs:element name="cacheName" type="xs:string" minOccurs="0"/>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="loggingType">
      <xs:sequence>
         <xs:element name="location" type="xs:string" minOccurs="0"/>
         <xs:element name="level" type="loglevelType" minOccurs="0"/>
         <xs:element name="logger" type="xs:string" minOccurs="0"/>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="communicationType">
      <xs:sequence>
         <xs:element name="isDistributed" type="xs:boolean" default="false" minOccurs="0"/>
         <xs:element name="transport" type="transportType" minOccurs="0"/>
         <xs:element name="useSSL" type="xs:boolean" minOccurs="0"/>
         <xs:element name="sslConfigFile" type="xs:string" minOccurs="0"/>
         <xs:element name="keyStore" type="xs:string" minOccurs="0"/>
         <xs:element name="port" minOccurs="0">
            <xs:complexType>
               <xs:attribute name="lower" type="xs:nonNegativeInteger" use="optional" default="0"/>
               <xs:attribute name="upper" type="xs:nonNegativeInteger" use="optional" default="0"/>
            </xs:complexType>
         </xs:element>
         <xs:element name="localAddress" type="xs:string" minOccurs="0"/>
         <xs:element name="discoverer" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
               <xs:complexContent>
                  <xs:extension base="discovererType">
                     <xs:attribute name="order" type="xs:nonNegativeInteger"/>
                     <xs:attribute name="original" type="xs:boolean"/>
                  </xs:extension>
               </xs:complexContent>
            </xs:complexType>
         </xs:element>
         <xs:element name="discovererElection" type="electionType" minOccurs="0"/>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="discovererType">
      <xs:attribute name="ip" type="xs:string"/>
      <xs:attribute name="discovery-port" type="xs:positiveInteger" use="required"/>
   </xs:complexType>
   <xs:complexType name="persistenceType">
      <xs:sequence>
         <xs:element name="location" type="xs:string"/>
         <xs:element name="disksize" type="xs:positiveInteger" default="30" minOccurs="0"/>
      </xs:sequence>
   </xs:complexType>
   <xs:simpleType name="loglevelType">
      <xs:restriction base="xs:token">
         <xs:enumeration value="OFF"/>
         <xs:enumeration value="FATAL"/>
         <xs:enumeration value="ERROR"/>
         <xs:enumeration value="DEFAULT"/>
         <xs:enumeration value="WARNING"/>
         <xs:enumeration value="TE"/>
         <xs:enumeration value="INFO"/>
         <xs:enumeration value="DEBUG"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="transportType">
      <xs:restriction base="xs:token">
         <xs:enumeration value="TCP"/>
         <xs:enumeration value="HTTP"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:complexType name="electionType">
      <xs:sequence>
         <xs:element name="useMulticast" type="xs:boolean" minOccurs="0"/>
         <xs:element name="updateInterval" type="xs:positiveInteger" minOccurs="0"/>
         <xs:element name="resolutionInterval" type="xs:positiveInteger" minOccurs="0"/>
         <xs:element name="multicastAddress" minOccurs="0">
            <xs:complexType>
               <xs:attribute name="ip" type="xs:string" use="optional"/>
               <xs:attribute name="port" type="xs:string" use="optional"/>
               <xs:attribute name="TTL" type="xs:nonNegativeInteger" use="optional"/>
            </xs:complexType>
         </xs:element>
         <xs:element name="usePriorityOrder" type="xs:boolean" minOccurs="0"/>
      </xs:sequence>
   </xs:complexType>
</xs:schema>
```

XML Schema for Attribute Declaration

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.oracle.com/oracle/ias/cache/configuration/declarative"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns="http://www.oracle.com/oracle/ias/cache/configuration/declarative"
 elementFormDefault="qualified" attributeFormDefault="unqualified">
   <xs:complexType name="regionType">
      <xs:sequence>
         <xs:element name="attributes" type="attributesType" minOccurs="0"/>
         <xs:element name="region" type="regionType" minOccurs="0" maxOccurs="unbounded"/>
         <xs:element name="group" type="groupType" minOccurs="0" maxOccurs="unbounded"/>
         <xs:element name="cached-object" type="cached-objectType" minOccurs="0"
maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
   </xs:complexType>
   <xs:complexType name="attributesType">
      <xs:sequence>
         <xs:element name="time-to-live" type="xs:positiveInteger" minOccurs="0"/>
         <xs:element name="default-ttl" type="xs:positiveInteger" minOccurs="0"/>
         <xs:element name="idle-time" type="xs:positiveInteger" minOccurs="0"/>
         <xs:element name="version" type="xs:string" minOccurs="0"/>
         <xs:element name="max-count" type="xs:positiveInteger" minOccurs="0"/>
         <xs:element name="priority" type="xs:positiveInteger" minOccurs="0"/>
         <xs:element name="size" type="xs:positiveInteger" minOccurs="0"/>
         <xs:element name="flag" minOccurs="0" maxOccurs="unbounded">
            <xs:simpleType>
               <xs:restriction base="flagType">
                  <xs:enumeration value="distribute"/>
                  <xs:enumeration value="reply"/>
                  <xs:enumeration value="synchronize"/>
                  <xs:enumeration value="spool"/>
                  <xs:enumeration value="group_ttl_destroy"/>
                  <xs:enumeration value="original"/>
                  <xs:enumeration value="synchronize-default"/>
                  <xs:enumeration value="allownull"/>
                  <xs:enumeration value="measure"/>
               </xs:restriction>
            </xs:simpleType>
         </xs:element>
         <xs:element name="event-listener" type="event-listenerType" minOccurs="0"/>
         <xs:element name="cache-loader" type="userDefinedObjectType" minOccurs="0"/>
         <xs:element name="capacity-policy" type="userDefinedObjectType" minOccurs="0"/>
         <xs:element name="user-defined" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
               <xs:sequence>
                  <xs:element name="key" type="xs:string"/>
                  <xs:element name="value" type="xs:string"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
   </xs:complexType>
   <xs:simpleType name="flagType">
      <xs:list itemType="xs:token"/>
   </xs:simpleType>
   <xs:complexType name="userDefinedObjectType">
      <xs:sequence>
         <xs:element name="classname" type="xs:string"/>
         <xs:element name="parameter" type="propertyType" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="propertyType">
      <xs:simpleContent>
         <xs:extension base="xs:string">
            <xs:attribute name="name" type="xs:string" use="required"/>
         </xs:extension>
      </xs:simpleContent>
   </xs:complexType>
   <xs:complexType name="event-listenerType">
      <xs:sequence>
```

```
         <xs:element name="classname" type="xs:string"/>
      </xs:sequence>
      <xs:attribute name="handle-event" type="handle-eventType" use="required"/>
      <xs:attribute name="default" type="xs:boolean"/>
   </xs:complexType>
   <xs:simpleType name="handle-eventType">
      <xs:restriction>
         <xs:simpleType>
            <xs:list itemType="xs:token"/>
         </xs:simpleType>
         <xs:enumeration value="object-invalidated"/>
         <xs:enumeration value="object-updated"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:complexType name="groupType">
      <xs:sequence>
         <xs:element name="attributes" type="attributesType" minOccurs="0"/>
         <xs:element name="group" type="groupType" minOccurs="0" maxOccurs="unbounded"/>
         <xs:element name="cached-object" type="cached-objectType" minOccurs="0"
maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
   </xs:complexType>
   <xs:complexType name="cached-objectType">
      <xs:sequence>
         <xs:element name="attributes" type="attributesType" minOccurs="0"/>
         <xs:element name="name" type="nameType" minOccurs="0"/>
         <xs:element name="object" type="userDefinedObjectType" minOccurs="0"/>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="nameType">
      <xs:choice>
         <xs:element name="string-name" type="xs:string"/>
         <xs:element name="object-name" type="userDefinedObjectType"/>
      </xs:choice>
   </xs:complexType>
   <xs:element name="cache">
      <xs:complexType>
         <xs:sequence maxOccurs="unbounded">
            <xs:element name="region" type="regionType"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

8 XML Query Service

This chapter describes how to use XML Query Service (XQS), a new service in the OC4J 10.1.3 implementation that provides a convenient user model for the retrieval, analysis, integration, and transformation of enterprise data. The following topics are covered:

- Introduction to XML Query Service
- Overview of XQS Features and Functionality
- How to Enable XQS As an OC4J Extension
- How to Prepare to Use Your Data Sources
- How to Configure Your XQS Functions
- How to Design Your Queries
- How to Develop Your Application Code: Using the XQS Client Interfaces
- How to Use OC4JPackager to Package Your XQS Application
- Using XQS Performance Features
- Using XQS Error Handling Modes and APIs
- XQS Client APIs Reference
- XQS Configuration File Reference
- OC4JPackager Reference
- Summary of XQS MBeans and Administration
- XQS Troubleshooting
- XQS Sample

Introduction to XML Query Service

This section introduces the XML Query Service, providing a brief overview of XQS and related technologies that it is built upon. This discussion includes the following:

- What is XQS?
- Technologies Related to XQS
- Why Use XQS?
- Requirements, Limitations, and Special Notes for the Current Release

What is XQS?

XQS is an OC4J service built upon XQuery (the XML query language) to provide a simplified, declarative mechanism for creating integrated views of enterprise data.

Generally, without a service such as XQS, XQuery is limited to accessing XML documents. With XQS, you can also retrieve data from non-XML documents, relational databases, and other possibly non-XML enterprise information systems, through access mechanisms such as SOAP or SQL.

XQS simplifies your programming task. In the planning stages, you need only focus on the design of your XQuery expressions (queries, built-in function calls, or user-defined function calls) and the structure of the data. The details of accessing data sources are determined later, through configuration settings—this is where a binding occurs between your XQuery expressions and the data to be accessed. XQS then does the work of creating external XQuery functions, referred to as *XQS functions*, used in retrieving the data from the desired sources.

XQS supports *ad-hoc queries*, which are XQuery expressions passed in at runtime, or *XQS views*, which are queries you created and saved previously. A view becomes an XQuery function that you can access later by name. (XQuery itself does not yet support the concept of views.) Views allow you to reuse XQuery expressions in building hierarchical queries that are powerful and yet easily maintainable.

XQS offers a variety of client interfaces, allowing you to execute your XQuery expressions through an EJB, a JSP tag library, a Java class, or a Web service.

Technologies Related to XQS

This section offers a glance at technologies that XQS is built upon:

- A Quick Look at XQuery
- The Oracle XQuery Implementation
- Comparing XQS with the XQuery API for Java

Note: For introductory tutorials for these and related technologies, you can try the following site (among many others):

`http://www.w3schools.com/`

A Quick Look at XQuery

XQuery is a declarative language for querying and transforming XML data. You can intelligently query an XML source, using desired criteria, and retrieve and interpret elements and attributes and the data they contain. It is a standard and flexible technology that applies broadly across many and varied kinds of XML sources, including XML databases as well as XML documents. Its role is comparable to the role SQL plays in querying a relational database, with similar functionality.

In addition to querying data, XQuery can transform XML data. In this way, it can be used as a complement or an alternative to XSLT.

In XQuery, the basic construct is an *expression*, with the data model for expressions being based on XPath 2.0. Each XQuery result is returned as an *XML sequence*. This is a sequence of zero or more data items, where each item is a scalar value, XML node, or XML document. Type definitions for items are based on the XML Schema standard.

A query may consist of one or more fragments called *modules*, and a module may consist of a *prolog* followed by a query body. A **prolog** is a series of declarations and imports that define the processing environment for the module.

The general model for a query in XQuery is the "for-let-where-order-return" (FLWOR) expression. Here is pseudocode showing an example of what a FLWOR expression could accomplish:

```
For each department in depts.xml
Let [variable] represent each employee in each department
Where the headcount of employees in a department is at least 10
Order by average salary of department
Return headcount and average salary of each department
```

XQuery also allows the definition and use of predefined functions to perform queries. For more complex queries, external functions may be provided separately, then declared and invoked from your XQuery definition. External functions may be implemented in a variety of languages, including Java and SQL.

The use of external functions is a critical part of XQS functionality. XQS creates external functions, according to your XQS configuration, to execute the desired queries on your data.

Notes:

- XQuery 1.0 is an extension of XPath 2.0, so they share the same data model, functions, and syntax. The data model is also common to XSLT 2.0.
- For details about XQuery, refer to the specification *XQuery 1.0: An XML Query Language*, available at the following location:

  `http://www.w3.org/TR/xquery/`

The Oracle XQuery Implementation

The Oracle XQuery implementation in Java is an underlying engine for XQS and is provided as part of the XML Query Service. Do not confuse this with the Oracle XQuery implementation that is part of Oracle XML DB in the Oracle Database product, although the two have common origins and common features.

XMLItem Type

Oracle's XQuery implementation uses the type `oracle.xml.xqxp.datamodel.XMLItem` to represent an individual item in the result XML sequence of a query.

Predefined Namespaces and Prefixes

The following table lists predefined namespaces and prefixes for use with XQuery and XQS.

Table 8–1 Predefined Namespaces and Prefixes

Prefix	Namespace	Description
ora	http://xmlns.oracle.com/xdb	Oracle XML DB namespace
local	http://www.w3.org/2003/11/xpath-local-functions	XPath local function declaration namespace

Table 8–1 (Cont.) Predefined Namespaces and Prefixes

Prefix	Namespace	Description
fn	http://www.w3.org/2003/11/xpath-functions	XPath function namespace
xdt	http://www.w3.org/2003/11/xpath-datatypes	XPath datatype namespace
xml	http://www.w3.org/XML/1998/namespace	XML namespace
xs	http://www.w3.org/2001/XMLSchema	XML Schema namespace
xsi	http://www.w3.org/2001/XMLSchema-instance	XML Schema instance namespace

You can use these prefixes in XQuery expressions without first declaring them in the XQuery-expression prolog. You can redefine any of them except `xml` in the prolog. All of these prefixes except `ora` are predefined in the XQuery standard.

Oracle XQuery Extension Functions

The Java version of Oracle XQuery includes support for `ora` extension functions first introduced in the database version, including the following:

- `ora:contains` lets you restrict a structural search with a full-text predicate.
- `ora:matches` lets you use a regular expression to match text in a string.
- `ora:replace` lets you use a regular expression to replace text in a string.
- `ora:sqrt` lets you return the square root of a number.
- `ora:view` lets you query existing database tables or views inside an XQuery expression, as if they were XML documents. In effect, this function creates XML views over the relational data.

For more information about these functions, refer to the XQuery chapter of the *Oracle XML DB Developer's Guide*.

Note: At the time of this release, the W3C XQuery working group had not yet published the XQuery recommendation. Oracle will continue to track the evolution of the XQuery standard, until such time as it becomes a recommendation. During this period, in order to follow the evolution of the XQuery standard, Oracle may be forced to release updates to the XQuery implementation which are not backwards compatible with previous releases or patch sets. During this period Oracle does not guarantee any backward compatibility between database releases or patch sets with respect to our XQuery implementation. After the XQuery standard becomes a recommendation, Oracle will produce a release or patch set that includes an implementation of the XQuery recommendation. From that point on, standard Oracle policies with respect to backwards compatibility will apply to the Oracle XQuery implementation. See `http://www.w3.org` for the latest information on the status of XQuery.

Implementation Choices Specified in the XQuery Standard

Implicit time zone support: In XQuery in XQS, the implicit time zone is always assumed to be `Z`.

Implementation Departures from the XQuery Standard

Boundary condition differences, for +0 and -0: The XQuery standard distinguishes positive zero from negative zero, but XQuery in XQS does not. Both are presented as 0, and they are treated equally.

XQuery Optional Features

There is currently no support for the following optional XQuery features defined by the W3C:

- Schema Import Feature
- Schema Validation Feature
- Module Feature

In addition to these defined optional features, the W3C specification allows an implementation to provide implementation-defined pragmas and extensions. These include the following:

- Pragmas
- Must-understand extensions
- Static-typing extensions

The Oracle implementation does not require any such pragmas or extensions.

Support for XQuery Functions and Operators

XQS supports all of the XQuery functions and operators included in the latest specification with the following exceptions, for which there is no support:

- XQuery regular-expression functions.

  Use the Oracle extensions for regular-expression operations, instead
- Functions `fn: trace`, `fn:id`, `fn:idref`, `fn:codepoint-equal`, `fn:prefix-from-QName`, `fn:doc-available`, and `fn:collection()` (`fn:collection()` without any argument)

XQuery Functions doc and collection

XQuery built-in functions `fn:doc` and `fn:collection` are essentially implementation-defined. XQS supports these functions. Function `doc` retrieves a file from the local file system that is targeted by its URI argument; it must be a file of well-formed XML data. Function `collection` is similar, but works on a folder (each file in the folder must contain well-formed XML data).

Comparing XQS with the XQuery API for Java

The XQuery API for Java (XQJ) is an evolving standard for executing XQuery expressions from Java applications. (Refer to JSR-225.) It is a low-level programmatic interface comparable to JDBC. Because it is not yet a standard, however, XQJ is not yet supported in the 10.1.3 implementation of the Oracle Application Server.

XQS supplies a higher-level API for similar underlying functionality. You can execute XQuery expressions from Java without having to write code to connect to the data source, create XQuery expression objects, and so on.

Why Use XQS?

XQS is versatile. It is useful for Web services as well as Java applications, and for accessing non-XML data as well as XML data.

XQS also relieves you of coding steps such as using the low-level XQJ interface to execute an XQuery expression. The XQS model of data source access through non-programmatic configuration allows faster and more convenient development. Also, registration of the library for the external functions that XQS creates is handled automatically, and namespace assignment is flexible and within your control.

As well as being used to retrieve data, XQS can serve as a convenient mechanism for transforming XML data or joining data from multiple sources.

In addition, XQS offers the following advantages and conveniences:

- The XQS configuration model allows you to create queries that are independent of the particular data source or XML document you access, making your application more portable. You need only alter your XQS configuration for a different source or document location.
- Use of the "XQS views" feature allows more convenient hierarchical queries, as well as allowing queries to be shared, reused, and centrally maintained.
- Through the use of performance optimizations in cooperation with other Oracle Application Server and Oracle Database components, a query executed through XQS is expected to perform at least as well as, and typically better than, the hand-coded XQJ equivalent. XQS itself offers additional performance enhancements, including configurable caching for source data and results, and efficiencies in handling large data sets.
- XQS simplifies migration between the middle tier and database tier. If a data source outgrows middle-tier capacity, you can typically migrate data to Oracle XDB without modifying or redeploying your middle-tier application. Move your data to the database and update your configuration for that data source as appropriate. Other middle-tier data sources can remain on the middle tier.

Also see "Introduction to XQS Performance and Optimization Features" on page 8-12.

Requirements, Limitations, and Special Notes for the Current Release

Be aware of the following for the XQS 10.1.3 implementation:

- XQS uses the OC4J Java Object Cache for its caching. It requires that the cache be running (automatically starting it if necessary), and requires the presence of the Java Object Cache configuration file. See "Configuring XQS Caching" for additional information about caching.
- Due primarily to limitations of the current XQuery specification, XQS reads from data sources but cannot write to data sources. You can query data, but not update, insert, or delete data.
- Within the set of nodes (as defined by the XML data model) returned by a function call, each node has a unique identity. But if a function is called more than once in a query, XQS does not guarantee the uniqueness of node identities across the different result sets. Because of this, some queries based on node identity may produce nondeterministic results. As an example, consider the following queries, Q1 and Q2:

```
(: Q1 :)
declare namespace xqs="http://xmlns.oracle.com/ias/xqs";
declare function xqs:a() external;
```

```
let $x := xqs:a()
return $x/b is $x/b
```

Q1 will always return `true`, given that the node identity test is on nodes from the same function call.

```
(: Q2 :)
declare namespace xqs="http://xmlns.oracle.com/ias/xqs";
declare function xqs:a() external;
xqs:a()/b is xqs:a()/b
```

Q2 may return `true` or `false`, depending on certain performance optimization settings.

Overview of XQS Features and Functionality

This section introduces important XQS features. Further details are provided later in the chapter.

- XQS Data Source Support
- Introduction to XQS Configuration and Configuration Files
- Introduction to XQS Client Interfaces
- Introduction to OC4JPackager
- Security for XQS Applications
- Introduction to XQS Performance and Optimization Features
- Introduction to XQS Error Handling
- Summary of the Main Steps in Using XQS

XQS Data Source Support

As noted previously, XQS supports several different kinds of data sources. This section covers the following related topics:

- Supported Categories of Data Sources
- Data Source Access Through XQuery Functions
- What Do Data Source Function Objects Do?
- Overview of Preparing Data Sources

Supported Categories of Data Sources

In XQS, data sources requiring a variety of access protocols can be described through WSDL documents because of XQS support for the popular Apache Web Services Invocation Framework (WSIF), an extension of the Web service mechanism behind SOAP and HTTP to include any custom invocation protocols. Because of this mechanism, XQS supports WSDL-based sources other than SOAP-based Web services.

In all, XQS supports the following categories of data sources, with special features to access various kinds of sources through a WSDL document.

- Document: Access files or, more generally, any URL-based input streams that contain XML documents. This is similar to the functionality of the built-in XQuery function `fn:doc()`. XQS supports the Oracle Data Definition Description Language (D3L) translator plug-ins to convert non-XML data to XML, so non-XML files are supported as well.

- XQS view: Run a previously stored query (similar in concept to database stored procedures). XQS locates the query, binds any external variables, and executes the query.
- WSDL source with SOAP binding: Access a resource through a Web service. You must provide a WSDL document, with a SOAP binding, to describe the Web service operations and the data source.
- WSDL source with SQL binding: Access tables, views, PL/SQL stored procedures, or Java stored procedures in a relational database. This is accomplished through the XQS WSIF provider for SQL, a binding that is defined by Oracle. See "Preparing to Use a Database Source (WSDL Source with SQL Binding)" on page 8-19 for information. You must provide a WSDL document with a SQL binding. XQS converts a relational result set to an XML sequence by invoking the Oracle XML-SQL Utility. XQS gets connection information for the appropriate data source, as specified in the WSDL, from the OC4J `data-sources.xml` configuration file.

  The XQS `fetchSize` attribute for WSDL sources with SQL binding recommends a number of rows for JDBC to fetch from the database in one round trip. This attribute is translated into a call to the `setFetchSize` method of `java.sql.PreparedStatement`. The `setFetchSize` parameter in JDBC (and, therefore, the `fetchSize` attribute in XQS) is only a hint: it is not binding.
- WSDL source with Java or EJB binding: Access a resource through any custom Java class or EJB that returns XML data. This is accomplished through the WSIF provider for Java or WSIF provider for EJB. You must implement the class or EJB and provide a WSDL document with a Java or EJB binding, specifying the class name or EJB name and any argument types.

A WSDL document that you provide for any of the WSDL sources will describe a set of callable operations and provide specifics for connecting to the source, using the appropriate binding. Your XQS configuration will reference a name for each operation, and point to the WSDL for the description of the operation. The WSDL must have a working URL, which you specify in your XQS configuration. XQS will go to that URL to fetch the WSDL. Also note that XQS allows you to use a WSDL URL as an XQuery namespace, and to treat individual operations of a WSDL port as local names in that namespace.

Data Source Access Through XQuery Functions

Access of your data sources and execution of your operations are accomplished through external XQuery functions, referred to as "XQS functions" (introduced earlier), that XQS automatically creates and invokes for you based on your XQS configuration. You specify the desired name and namespace of the function in your configuration, along with other relevant items (input parameters, for example). When you use a data source in a query, you must declare the associated XQS function as an external function in your XQuery prolog.

A namespace corresponds to a function library containing function objects. Each function object corresponds to an operation. For a WSDL source with SOAP binding, for example, there is a function object for each operation in the WSDL document. For a WSDL source with SQL binding, each SQL query or stored procedure call maps to a function.

What Do Data Source Function Objects Do?

Each function object created by XQS is instantiated according to your XQS configuration, and performs the following basic tasks:

1. Accepts input arguments passed from the XQuery engine. Permissible input types are according to what is allowable in an XML sequence; namely, primitive Java types and XML nodes. XQS cannot perform Java object-to-XML mapping, but you can perform mapping prior to the query, such as through Oracle TopLink.
2. Invokes the query against the underlying data source. XQS maps the function name to the corresponding XQS configuration element, then finds the connection information for the data source and creates the connection.
3. Receives and packages results. XQS synchronously receives results from the data source, in XML form, then packages the results into an XML sequence.
4. Processes any errors (such as problems accessing a data source, for example, or type incompatibilities) and returns error information according to your error-handling configuration.

Overview of Preparing Data Sources

For most data sources you can use with XQS, there are necessary preparation steps. For example:

- To use a non-XML document source, you must prepare to use the conversion tool, Oracle D3L, that XQS supplies. Preparation includes providing a D3L schema file with instructions about the data format.
- To use an XQS view, you must consider the data input and return types, design the query, save it as an `.xq` file, and decide where to place it.
- To use a WSDL source with any of the supported bindings, you must provide (in some cases, create) an appropriate WSDL document.

See "How to Prepare to Use Your Data Sources" on page 8-15 for details.

Introduction to XQS Configuration and Configuration Files

As is true with OC4J and its other components as well, XQS has the concept of global configuration versus application-specific configuration. Global configuration is for XQS functions and data sources that are to be available to all applications running in the OC4J instance, as opposed to being available only to a particular application.

The application-specific XQS configuration file is `xqs-config.xml`, which XQS looks for in the `xqs-resources.jar` file (created by the OC4J packaging utility) at the top level of the application EAR file. Use it to specify information about XQS functions and data sources specific to your application.

The global XQS configuration file is `global-xqs-config.xml`, in the *`ORACLE_HOME`*`/j2ee/home/config` directory, which you can use to specify sources available to all applications.

When XQS looks for configuration for an XQS function, it first looks in the application-level file. If (and only if) it does not find it there, it will look in the global file. (In other words, if a function is configured in both files, it is ignored in the global file.)

The top-level elements are as follows:

- `<document-source>` to access data from a document, including either XML or non-XML files
- `<xqsview-source>` to use an XQS view
- `<wsdl-source>` for any data source involving a user-provided WSDL document

XQS reads information from the configuration file and uses that information to populate the XQS function objects that access the data sources.

You do not have to restart OC4J when you update configuration, but you must redeploy the application. (See the ***Oracle Containers for J2EE Deployment Guide*** for information.)

See "How to Configure Your XQS Functions" on page 8-24 for additional information.

Note: In the 10.1.3 implementation of the Oracle Application Server, there are not yet any pages for XQS configuration in the Oracle Enterprise Manager 10*g* Application Server Control Console. Although MBeans are available (summarized for reference in "Summary of XQS MBeans and Administration" on page 8-109), we recommend for the current release that you manage your XQS configuration directly in the `xqs-config.xml` file.

Introduction to XQS Client Interfaces

XQS supports the following client interfaces for implementing your XQuery functionality. See "How to Develop Your Application Code: Using the XQS Client Interfaces" on page 8-39 for usage information and examples for each of these interfaces.

Note: When you use any of the XQS client APIs for a query, the query execution defines the results, but does not necessarily put all the results into memory immediately. A set of results that is immediately and fully stored into memory is said to be *materialized*, whereas a set of results accessed one at a time (or several at a time, with batching) through an implicit cursor, using some sort of "next item" functionality, is said to be *nonmaterialized*. With nonmaterialized results, there is no guarantee as to when results are retrieved and written to memory. Any "next item" call may trigger the evaluation of the XQuery expression to produce the next result item.

As noted in the descriptions that follow, XQS client APIs allow you the choice of using *stateless* client objects, with materialized results, or using *stateful* client objects, with nonmaterialized results.

These concepts are discussed in further detail, and with strategic considerations, in "Stateful Versus Stateless Clients" on page 8-41.

- Java class client API: XQS provides a general-purpose Java class, `XQSFacade`, that you can use for a Java implementation of desired XQuery functionality. This class is based on the "facade" design pattern, shielding you from details and complexities of the underlying XQS functionality. Use the appropriate "execute" method to pass in an ad-hoc query or XQS view name and any bind parameters. In either case, use the "get next item" method to process the results. An `XQSFacade` instance returns an XML result sequence directly to the client, meaning it is up to the client whether to retrieve and process items from the sequence incrementally (a stateful approach), or to retrieve and process all items from the sequence at once (a stateless approach).

Note: The `XQSFacade` class is also used behind the scenes for the EJB client API and JSP tag library.

- EJB client API: The EJB client API provides a way to access your application through session beans, either remotely (through RMI) or locally. XQS supports the use of either stateful or stateless session beans. Stateless beans minimize calls to the EJB, while stateful beans are preferable if memory usage is a concern. In either case, you provide the ad-hoc query or XQS view name, along with any bind parameters, in the appropriate "execute" method. For a stateful session bean, you use the "get next item" method to process the results. For a stateless session bean, the sequence is always materialized and is returned by the "execute" method.
- JSP tag library: The JSP tag library provides a way to access your application through HTTP. XQS provides JSP tags for either stateful or stateless access. You provide the ad-hoc query or XQS view name using an "execute" or "executeCursor" tag. Which tag to use depends on whether you want stateful or stateless access. There is a nested "parameter" tag for specifying bind parameters. For stateful access, you use the "next" tag associated with the "executeCursor" tag to process the results. For stateless access, the sequence is always materialized and is returned by the "execute" tag. Results can be returned as a JSP output stream, a DOM document, or an array of Java `Object` instances (according to attribute settings of the "execute" or "executeCursor" tag).
- XQS view Web service: When you use an XQS view, you can optionally have XQS add an operation to a WSDL document that it creates, to expose the view as a Web service operation. (This is accomplished as part of the configuration for an XQS view.) You can then implement a client for this as you would for any other Web service.

Note: Also be aware of the XQS `QueryParameter` class. To use queries with external binds for the Java client API or EJB client API, you must create an array of `QueryParameter` objects for the bind values. See "XQS QueryParameter Class Reference" on page 8-75 for reference information. (This class is also used behind the scenes for the JSP client interface.)

Introduction to OC4JPackager

OC4JPackager, provided with XQS, is a command-line Java tool that packages XQS-related files with a J2EE or Web application to allow the application to use XQS functionality.

In a typical scenario, you would write a Web application that calls XQS through one of the XQS client APIs (the `XQSFacade` class, EJB client API, or JSP tag library). You would also create an XQS configuration file, `xqs-config.xml`, that points to the relevant data sources and XQS views, and choose an XQS repository—the directory where your views are located. Then you would bundle your application, XQS configuration file, and XQS repository into an EAR file, which you then deploy.

Given instructions through its command-line parameters, OC4JPackager will complete the step of bundling everything for you. Specifically, it does the following:

1. Bundles `xqs-config.xml` and all XQS repository files into a file called `xqs-resources.jar`.

2. Opens each archive associated with your application (such as WAR files and EJB JAR files) and modifies the `Class-Path` attribute in the manifest (`MANIFEST.MF`) to include `xqs-resources.jar`.
3. Bundles an EAR archive that consists of all your application archive files plus `xqs-resources.jar`.
4. Creates or modifies (as applicable) the `orion-application.xml` file in the EAR archive to add `xqs-resources.jar` to the list of libraries accessible by all components of the EAR.

When you run OC4JPackager, you specify a directory that contains one of the following:

- An existing EAR file, which OC4JPackager will unbundle and rebuild
- An existing set of J2EE modules such as WAR and EJB JAR files, which OC4JPackager will bundle into a new EAR.

For an XQS view that you want to expose as a Web service operation, OC4JPackager performs additional tasks, delegating invocation of the Web service operation to the XQS view. The operation is added to a WSDL document that is automatically generated. For more information see "OC4JPackager Additional Output to Expose XQS Views as Web Service Operations" on page 8-62.

For more information about OC4JPackager in general, see "How to Use OC4JPackager to Package Your XQS Application" on page 8-58 and "OC4JPackager Reference" on page 8-106.

Security for XQS Applications

XQS itself does not add any layers of security. In this way, it is essentially like any J2EE application. Security is provided by OC4J through standard Web and J2EE security features that you must use, as appropriate for the type of XQS client you are using and the type of data source you are accessing.

For example, if you are using a WSDL source with a SQL binding, use data source security features. If you are using the XQS EJB client interface, use standard EJB security. If you are using the XQS JSP tag library, use standard authentication for an HTTP connection. And so on.

Introduction to XQS Performance and Optimization Features

XQS offers the following features to improve performance:

- Caching: XQS can cache the results of XQS view execution, or cache XML data from data sources for use in future queries. This improves performance, particularly for situations where data sources, such as external Web services, may require significant time to access. Caching is also important when accessing asynchronous sources. You can specify expiration and invalidation settings for cached data, or optionally disable caching entirely, through your XQS configuration.
- Handling of large data sets: XQS has features to minimize the chance of running out of system memory when you access very large data sets, such as dozens of megabytes or more. In such situations, XQS will materialize the results partially, as the data is needed, staying within a fixed amount of memory usage (a "working unit" approach). You have the option of returning data items one by one.
- Scalability: XQS efficiently streams inputs to the XQuery engine, allows sharing and reuse of cached resources (such as parsed WSDL documents), and can use

non-materialized result sequences as appropriate for scalability. You can control these behaviors through your XQS configuration.

See "Using XQS Performance Features" on page 8-65 for more information.

Note: While XQS works in close conjunction with the Oracle XQuery engine and XDK to optimize data flow, XQS does not optimize the queries.

Introduction to XQS Error Handling

An XQS application may encounter problems related to XQS external functions and the underlying data sources during execution, such as problems converting input parameters to types expected by the data source, or problems converting data returned by the data source to XML. In addition, the data source may be unavailable or may return an error. The default behavior of XQS is to raise an XQuery dynamic error in these situations, which stops execution of a query, with no results being returned.

Alternatively, an XQS configuration attribute (`onError`) allows you to choose less severe behaviors, allowing XQS to continue so you can obtain additional information about any errors that occurred. Three error-handling modes are available:

- `dynamicError` (default)
- `emptySequence`
- `errorMessage`

With `emptySequence` or `errorMessage` mode, if an error is encountered during execution of an XQS external function for which this mode has been set, the error does not terminate query execution, and will be retained by XQS for later retrieval if desired. XQS returns an empty XML sequence (for `emptySequence` mode) or a one-item XML sequence consisting of an error message (for `errorMessage` mode). In `errorMessage` mode, you can preconfigure a fixed error message in your XQS configuration, or you can have the message be whatever is returned by the data source.

Information about "suppressed" errors from XQS external functions is returned by XQS in an iterator of `XQSError` objects.

XQS applies special handling only to errors inside an XQS function. A regular XQuery error, such as a syntax error or type mismatch, terminates query execution regardless of the XQS error mode.

You configure the XQS error mode for each XQS function individually. The error mode for one function does not have any affect on the behavior of other XQS functions in the same query.

See "Using XQS Error Handling Modes and APIs" on page 8-69 for more information.

Summary of the Main Steps in Using XQS

The following are the key steps in using XQS. Links are provided to the sections that cover each step in detail.

1. Enable XQS as an OC4J extension.

   For an instance of OC4J that Oracle Process Management and Notification (OPMN) manages, you can enable XQS by adding the following Java system property to the *`ORACLE_HOME`*`/opmn/conf/opmn.xml` configuration:

```
-Doracle.hooks=oracle.xds.XDSExtension
```

After configuring `opmn.xml`, you can use Oracle Enterprise Manager (OEM) or the OPMN tool to start or stop XQS with the OC4J instance. See "How to Enable XQS As an OC4J Extension" on page 8-14.

For a standalone instance of OC4J, pass following Java system property to the OC4J container:

```
-Doracle.hooks=oracle.xds.XDSExtension
```

You can pass this property on the command line, as follows:

```
java -jar oc4j.jar -Doracle.hooks=oracle.xds.XDSExtension
```

Or you can pass it by using the Ant task startoc4j.

See the *Oracle Application Server Installation Guide* for more details regarding various OC4J starting options. XQS users need to pass the additional Doracle.hooks parameter, as the preceding text describes.

2. Prepare your data sources as necessary. For a document source, for example, this would include any setup to convert from non-XML to XML. For XQS views, this includes defining and saving the queries. Some access of WSDL-based sources, such as databases, also requires special setup. See "How to Prepare to Use Your Data Sources" on page 8-15.
3. Configure XQS. Use the `xqs-config.xml` (or `global-xqs-config.xml`) file to specify and configure data sources to be accessed and queries to be executed, and to create mappings to the XQS functions that represent the data sources for the queries. See "How to Configure Your XQS Functions" on page 8-24.
4. Design your queries. See "How to Design Your Queries" on page 8-36.
5. Develop your application. This primarily involves using one of the APIs that XQS provides to execute your XQuery expressions, and processing the XML results that are returned. See "How to Develop Your Application Code: Using the XQS Client Interfaces" on page 8-39.
6. Package and deploy your application. Use the packager supplied with XQS. See "How to Use OC4JPackager to Package Your XQS Application" on page 8-58.

Note: This is a simplification. Typically, you cannot actually go through the steps in a modular, sequential manner, especially with multiple sources that may include XQS views. The process is iterative. For example, you could start by preparing a database source and then design a query for it. Then you may want to persist the query as an XQS view—in other words, prepare an XQS view source. Then you may design another query that uses the view.

How to Enable XQS As an OC4J Extension

For an OPMN-managed instance of OC4J, you can edit the `*ORACLE_HOME*/opmn/conf/opmn.xml` configuration to enable XQS as an OC4J extension. Add the following Java system property to the `start-parameters` category in the `OC4J` module of the IAS component:

```
-Doracle.hooks=oracle.xds.XDSExtension
```

For example:

```
<ias-component id="OC4J">
<process-type id="home" module-id="OC4J" status="enabled">
<module-data>
<category id="start-parameters">
<data id="java-options" value="-server
-Djava.security.policy=$ORACLE_HOME/j2ee/home/config/java2.policy
-Djava.awt.headless=true -Dhttp.webdir.enable=false
-Doracle.hooks=oracle.xds.XDSExtension/>
</category>
<category id="stop-parameters">
<data id="java-options"
value="-Djava.security.policy=$ORACLE_HOME/j2ee/home/config/java2.policy
-Djava.awt.headless=true -Dhttp.webdir.enable=false"/>
</category>
</module-data>
<start timeout="600" retry="2"/>
<stop timeout="120"/>
<restart timeout="720" retry="2"/>
<port id="default-web-site" range="12501-12600" protocol="ajp"/>
<port id="rmi" range="12401-12500"/>
<port id="jms" range="12601-12700"/>
<process-set id="default_group" numprocs="1"/>
</process-type>
</ias-component>
```

After you configure `opmn.xml`, you can start or stop an OC4J instance, including XQS, with EM or the OPMN tool.

For a standalone instance of OC4J, pass following Java system property to the OC4J container:

```
-Doracle.hooks=oracle.xds.XDSExtension
```

You can pass this property on the command line, as follows:

```
java -jar oc4j.jar -Doracle.hooks=oracle.xds.XDSExtension
```

Or you can pass it by using the Ant task startoc4j.

How to Prepare to Use Your Data Sources

For most data sources you want to use with XQS, there are necessary preparation steps. This section covers the following topics:

- Preparing to Use a Non-XML Document Source
- Preparing to Use an XQS View
- Preparing to Use a WSDL Source with SOAP Binding
- Preparing to Use a Database Source (WSDL Source with SQL Binding)
- Preparing to Use a Custom Class or EJB (WSDL Source with Java or EJB Binding)

Preparing to Use a Non-XML Document Source

Some documents do not use XML as their native message format, instead using native formats such as structured records of bytes and characters. Examples of this are Excel comma-separated-values (CSV) files. For non-XML data to be understandable for use with XQS and XQuery, it must follow a predefined, structured set of rules so that XQS can use an appropriate conversion tool. Data in such a structured format can be processed so that it can be retrieved and transformed into an XML format for your application.

For use of non-XML documents, the XQS 10.1.3 implementation supports the Oracle Data Definition Description Language (D3L). Prepare to use a non-XML document source with XQS by taking the following steps:

1. Ensure that the non-XML data is compatible with the D3L conversion mechanism.
2. Provide a schema file that gives instructions to D3L about the data format, so D3L can parse and convert the data.
3. Configure XQS to use D3L, specifying the name and location of the schema file.

The following sections provide an overview of D3L and its usage:

- What is D3L?
- D3L Schema Files
- Configuring XQS to Use D3L

Note: See the *Oracle Application Server Integration InterConnect User's Guide* for details about D3L features.

What is D3L?

D3L describes the structure that must be followed by the native, non-XML format of a document to allow processing by certain Oracle middle-tier components—in general, by OracleAS Integration InterConnect; for purposes of this document, by XQS.

D3L supplies the following:

- An XML-based data description language that describes the format of native files, such as the record layout of binary, string, structured, and sequence data
- A translation engine that uses the instructions from a D3L schema file to translate the native format file contents

D3L schema files must comply with the syntax defined by the D3L document type definition (DTD). You specify the D3L schema file to use through your XQS configuration.

Important: To use D3L, the number of fields in the underlying native format data must be fixed and known. D3L is not suitable for arbitrarily structured data (such as regular XML), name-value pair data, or conditional data structures that require token look-aheads to parse.

D3L Schema Files

A D3L schema file describes data types, formats, and delimiters so that the D3L translation engine can parse the data and convert it to XML. The schema file must conform to the specifications of `d3l.dtd`.

Refer to the *Oracle Application Server Integration InterConnect User's Guide* for details about D3L schema files and how to create them, but here is a fragment to give you a general idea of their appearance:

```
<?xml version="1.0" encoding="US-ASCII"?>
```

```
<!DOCTYPE message SYSTEM "d3l.dtd">
<message name="replyFlight" type="BookingReplyType" object="Booking"
         header="D3L-Header" value="replyOptions">
   <unsigned4 id="u4" />
   <unsigned2 id="u2" />
   <struct id="DateTimeRecord">
      <field name="DateInfo">
         <date format="MMDDYY">
            <pfxstring id="datstr" length="u4" />
         </date>
      </field>
      <field name="TimeHour"><limstring delimiter="*" /></field>
      <field name="TimeMinute"><limstring delimiter="*" /></field>
   </struct>
...
</message>
```

Configuring XQS to Use D3L

To have XQS use the D3L conversion mechanism for a non-XML document source:

1. Use the `<XMLTranslate>` subelement of `<document-source>` to direct XQS to use D3L. (In the future, other tools may be supported as well, and specified in the same way.)
2. Use the `<schema-file>` subelement of `<XMLTranslate>` to specify the schema file for D3L to use in parsing the data.

Here is the XQS configuration to use a D3L schema file named `PersonalInfoD3L.xml`:

```
<document-source ... >
   ...
   <XMLTranslate method="D3L">
      <schema-file>http://host:port/xqs/PersonalInfoD3L.xml</schema-file>
   </XMLTranslate>
   ...
</document-source>
```

Notes:

- There is reference information for these elements under "XQS Configuration File Reference" on page 8-86.
- Also see "Configuring an XQS Function That Accesses a Document Source" on page 8-24.

Preparing to Use an XQS View

As noted earlier, an XQS view is a query that is stored for future use. XQS treats the view itself as a source, and you configure it through an `<xqsview-source>` element. To prepare to use a query as an XQS view, do the following:

1. Consider any input parameters that the query will require. You can specify a parameter for an XQS view by declaring it as an external variable in the XQuery prolog.
2. Design the query, including external variable declarations for any input parameters, and save it as an `.xq` file. (Also see "How to Design Your Queries" on page 8-36.)

3. Consider where you will place the `.xq` file. Your location for `.xq` files is referred to as the XQS repository. You can specify a location through your XQS configuration (using the `<repository>` element), or you can use a location you specify through the `-repository` option when you run OC4JPackager.
4. Consider whether to expose the XQS view as a Web service operation. Do this through the `WSDLvisibility` attribute of the `<xqsview-source>` element in your XQS configuration. This results in the view being included as an operation in a Web service that XQS adds to your application. If you do expose the view as a Web service operation, be aware of the XML output type—we advise that you reflect that type in the `<output-element>` subelement of `<xqsview-source>` in your configuration. (Also see "Using an XQS View Exposed as a Web Service Operation" on page 8-58 for additional information.)

Following is an XQuery expression that we will use for an XQS view. It accepts the external variable `loc` and passes it through to a function `readFile` that reads purchase order data from a file. You can use any desired `.xq` file name to save it, and specify that name in your XQS configuration. You will also have to specify a desired name for the XQS function that XQS will implement to execute the view. The file name and function name are distinct, but you can use the function name as the base name of the file by default if you want.

```
declare variable $loc external;
declare namespace xqs = "http://xmlns.oracle.com/ias/xqs";
declare function xqs:readFile($l as xs:string) external;
for $po in xqs:readFile($loc)//po return $po//total
```

In your XQS configuration, under the `<xqsview-source>` element, you will typically use at least the following:

- The `WSDLvisibility` attribute if you want to expose the view as a Web service operation, and then the `<output-element>` subelement if you do expose the view
- The `<function-name>` subelement to specify the desired name of the XQS function for the view
- The `<input-parameters>` subelement and its subelements to specify external variables that correspond to input parameters of the XQS function (`loc` in the preceding example)
- The `<queryName>` subelement to specify the name of the `.xq` file containing the view (unless you use a base file name that matches the XQS function name)
- The `<repository>` subelement to specify the location of the `.xq` file (not necessary if the file is in the location specified through the `-repository` option when you run OC4JPackager)

Notes:

- There is reference information for these elements under "XQS Configuration File Reference" on page 8-86.
- For a continuation of the above example, see "Configuring an XQS Function That Uses an XQS View" on page 8-27.

Preparing to Use a WSDL Source with SOAP Binding

To access a data source through a Web service operation, you must provide a WSDL document for the Web service and specify the location of the WSDL in your XQS configuration. One XQS view function corresponds to one Web service operation that XQS will implement.

Examine the WSDL. You will use a `<wsdl-source>` element and its subelements in the XQS configuration file to configure the XQS function, with configuration settings (some optional) corresponding to WSDL entries as follows:

- The WSDL operation corresponds to an `<operation>` element in the XQS configuration.
- For WSDL input messages, each input message part corresponds to a `<part>` subelement of `<input-parameters>`.
- The applicable WSDL service corresponds to a `<service>` element.
- The applicable WSDL port corresponds to a `<port>` element.
- The applicable WSDL port type corresponds to a `<portType>` element.

In addition, consider how the WSDL document may be accessed by XQS, and specify this location using the `<wsdlURL>` element.

Notes:

- There is reference information for these elements under "XQS Configuration File Reference" on page 8-86.
- For an example of a WSDL document and a corresponding XQS configuration fragment, see "Configuring an XQS Function That Accesses a WSDL Source" on page 8-32.

Preparing to Use a Database Source (WSDL Source with SQL Binding)

The XQS WSIF provider for SQL allows you to use XQuery on data fetched from a relational database that is accessed through SQL. To use this feature, you must provide a WSDL document that specifies SQL binding and contains complete information about any SQL operations (queries or stored procedure calls) to perform.

As an initial summary, be aware of the following for the WSDL document:

- For connection information, the WSDL refers to a JNDI name, and XQS looks in the OC4J `data-sources.xml` configuration file for the data source associated with that JNDI name.
- The WSDL specifies the SQL statement for each operation. Each WSDL operation is for a single SQL statement (and corresponds to a single XQS function, as noted earlier).
- The XQS `fetchSize` attribute for WSDL sources with SQL binding recommends a number of rows for JDBC to fetch from the database in one round trip. This attribute is translated into a call to the `setFetchSize` method of `java.sql.PreparedStatement`. The `setFetchSize` parameter in JDBC (and, therefore, the `fetchSize` attribute in XQS) is only a hint, not a binding.
- Each WSDL operation must use a predefined message type, `SQLOutputMessage`, for the output message. This is defined in the Oracle `xqs.wsdl` document, which you can import into your WSDL as discussed below. The output message contains one part, named `result`, corresponding to the root of the XML document containing the query results. The `result` type is defined in the XQS types namespace:

  ```
  http://xmlns.oracle.com/ias/xqs/types/
  ```

- Each WSDL operation must use a predefined message type, `SQLFaultMessage`, for fault messages. This is also defined in `xqs.wsdl`. The fault message contains two parts: `code` for the error code and `message` for the error message. These are also defined in the XQS types namespace.
- Each operation can specify an XML transform to use for non-XML data. Currently only the Oracle XML-SQL Utility (XSU) is supported.

Specifically, you must use the XQS SQL extensions in your WSDL as follows (and as shown in the example afterward).

WSDL Definitions and Imports

1. Declare the namespaces and associated prefixes for XQS, XQS types, and the XQS SQL extensions. (By convention, in discussion that follows, we use the "sql:" prefix for XQS SQL extensions.)
2. Import `xqs.wsdl` to define `SqlOutputMessage` and `SqlFaultMessage`. The `xqs.wsdl` document is available with the `XQSDemo` application, so that you can place it in the same directory as your WSDL document. Alternatively, you can import it from the following location:

```
<import namespace="http://xmlns.oracle.com/ias/xqs/"
      location="http://www.oracle.com/technology/tech/xml/xqs/xqs.wsdl" />
```

Input Message Declarations

1. If your SQL query uses bind variables, you must specify an input message with a part for each bind variable, where the part types match the required bind variable types.

WSDL Port Type Declarations

For each operation:

1. Use `message="xqs:SqlOutputMessage"` for the `<output>` element.
2. Use `message="xqs:SqlFaultMessage"` for the `<fault>` element.

WSDL Binding Declarations

In the `<binding>` element:

1. Use an empty `<sql:binding/>` subelement.
2. For each operation, use a `<sql:operation>` subelement with `XMLtransform="XSU-Client"` and element contents specifying the SQL statement to execute.
3. For each operation input, use a `<sql:input>` subelement to specify the names of the bind parameters. This is a comma-separated list of previously defined message part names, in the order in which they are to be bound to the SQL statement.

WSDL Service Declarations

1. Use a `<sql:address>` element, setting the `data-source-location` attribute to indicate the JNDI location of the data source, as configured in the OC4J `data-sources.xml` file.

Sample WSDL with SQL Binding

```
<?xml version="1.0" ?>
<definitions targetNamespace="http://xqstest.sql.scott/"
```

```
             xmlns:scott="http://xqstest.sql.scott/"
             xmlns:xs="http://www.w3.org/2001/XMLSchema"
             xmlns:xqs="http://xmlns.oracle.com/ias/xqs/"
             xmlns:xqstypes="http://xmlns.oracle.com/ias/xqs/types/"
             xmlns:sql="http://xmlns.oracle.com/ias/xqs/sql/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
  <!-- import definitions of XQS SqlOutputMessage and SqlFaultMessage-->
  <import namespace="http://xmlns.oracle.com/ias/xqs/" location="xqs.wsdl" />
  <!-- input messages -->
  <message name="EmpByNameAndDeptInput">
    <part name="ename" type="xs:string"/>
    <part name="deptno" type="xs:int"/>
  </message>
  <message name="EmpByNameInput">
    <part name="ename" type="xs:string"/>
  </message>
  <!-- port type declarations -->
  <portType name="ScottSqlSource">
    <operation name="SelectEmpByName">
      <input name="SqlInput" message="scott:EmpByNameInput"/>
      <output name="SqlResult" message="xqs:SqlOutputMessage"/>
      <fault name="SqlFault" message="xqs:SqlFaultMessage"/>
    </operation>
    <operation name="SelectEmpByNameAndDept">
      <input name="SqlInput" message="scott:EmpByNameAndDeptInput"/>
      <output name="SqlResult" message="xqs:SqlOutputMessage"/>
      <fault name="SqlFault" message="xqs:SqlFaultMessage"/>
    </operation>
  </portType>
  <!-- binding declarations -->
  <binding name="SQLBinding" type="scott:ScottSqlSource">
    <sql:binding/>
    <operation name="SelectEmpByName">
      <sql:operation XMLtransform="XSU-client">
         select * from emp where ename=:1
      </sql:operation>
      <input name="SqlInput">
         <sql:input> ename </sql:input>
      </input>
      <output name="SqlResult"/>
      <fault name="SqlFault"/>
    </operation>
    <operation name="SelectEmpByNameAndDept">
      <sql:operation XMLtransform="XSU-client">
         select * from emp where ename=:1 and deptno=:2
      </sql:operation>
      <input name="SqlInput">
         <sql:input> ename,deptno </sql:input>
      </input>
      <output name="SqlResult"/>
      <fault name="SqlFault"/>
    </operation>
  </binding>
  <!-- service declaration -->
  <service name="ScottSQLService">
    <port name="ScottSQLPort" binding="scott:SQLBinding">
      <sql:address  data-source-location="jdbc/OracleCoreDS" />
    </port>
  </service>
</definitions>
```

xqs.wsdl

For reference, here is the Oracle `xqs.wsdl` document, imported by the preceding sample WSDL.

```
<?xml version="1.0" ?>
<definitions targetNamespace="http://xmlns.oracle.com/ias/xqs/"
             xmlns:xqstypes="http://xmlns.oracle.com/ias/xqs/types/"
             xmlns:xs="http://www.w3.org/2001/XMLSchema"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
  <!-- XQS type defs -->
  <types>
   <xs:schema targetNamespace="http://xmlns.oracle.com/ias/xqs/types/"
              xmlns:xs="http://www.w3.org/1999/XMLSchema">
     <!-- Result of all SQL functions is an XML Node
          representing document root,
          with optional XML type for elements-rows -->
     <xs:complexType name="SQLXMLDocument">
         <xs:attribute name="rowType" type="xs:QName" />
         <xs:sequence>
             <xs:element name="root" type="xs:anyType"
                         minOccurs="0" maxOccurs="1"/>
         </xs:sequence>
     </xs:complexType>
   </xs:schema>
  </types>
  <!-- all XQS sql functions should use this message type for output -->
  <message name="SqlOutputMessage">
    <part name="result" type="xqstypes:SQLXMLDocument"/>
  </message>
  <!-- all XQS sql functions should use this message type for error (fault) -->
  <message name="SqlFaultMessage">
    <part name="code" type="xs:int"/>
    <part name="message" type="xs:string"/>
  </message>
</definitions>
```

Considerations for the XQS Configuration

For general configuration considerations for a WSDL source, see "Preparing to Use a WSDL Source with SOAP Binding" on page 8-18.

In particular, each `<port>` element in your XQS configuration for a WSDL source must refer to the appropriate SQL binding. For example, consider the following XQS configuration:

```
<port namespace="http://xqstest.sql.scott/">
   ScottSQLPort
</port>
```

Corresponding to the following WSDL port definition:

```
<service name="ScottSQLService">
   <port name="ScottSQLPort" binding="scott:SQLBinding">
      <sql:address data-source-location="jdbc/OracleCoreDS" />
   </port>
</service>
```

- Each XQS function name in your XQS configuration is bound to a single SQL statement in the WSDL.
- You must specify a `<port>` element that refers to the correct SQL binding.

Notes:

- There is reference information for these elements under "XQS Configuration File Reference" on page 8-86.
- Also see "Configuring an XQS Function That Accesses a WSDL Source" on page 8-32.

Preparing to Use a Custom Class or EJB (WSDL Source with Java or EJB Binding)

You can access a data source through a custom Java class or EJB by using the WSIF provider for Java or the WSIF provider for EJB. XQS does nothing special in this area; this is open technology from the Apache Foundation. You must do the following:

- Implement a custom Java class or EJB that returns XML data that you can then query.
- Create a WSDL document with Java or EJB binding (as appropriate) to define the desired operations.
- Consider any XML-Java type mapping you will require.

See "Preparing to Use a WSDL Source with SOAP Binding" on page 8-18 for general WSDL considerations for your XQS configuration.

Refer to the *Oracle Application Server Web Services Developer's Guide* for information about how to implement the class or bean and the *Oracle Application Server Advanced Web Services Developer's Guide* for information on the WSIF providers for Java and EJB.

Note: Also see "Configuring an XQS Function That Accesses a WSDL Source" on page 8-32.

Here is a sample fragment from a WSDL document with Java binding. With this, you can invoke the Java method `readEntry()`, for example, by its operation name. Note that a WSDL `<format:typeMapping>` specification corresponds to a `<typeMap>` element in the XQS configuration.

```
< definitions>
  ...
  <binding name="JavaBinding" type="tns:AddressBook">
    <java:binding />
    <format:typeMapping encoding="Java" style="Java">
       <format:typeMap typeName="typens:address"
             formatType = "localjava.client.stub.addressbook.wsiftypes.Address" />
       <format:typeMap typeName="xsd:string"
             formatType="java.lang.String" />
    </format:typeMapping>
    <operation name="readEntry">
        <java:operation methodName="readEntry"
              parameterOrder = "name"
              methodType = "instance" />
        <input name="ReadEntryWholeNameRequest" />
    <operation name="readAllMatchingEntries">
     ...
    </operation>
  </binding>
  <service name="AddressBookService">
     <port name="JavaPort" binding="tns:JavaBinding">
        <java:address
```

```
          className = "localjava.service.AddressBookImpl" />
     </port>
  </service>
  ...
</definitions>
```

How to Configure Your XQS Functions

There must be an entry in the XQS configuration for each XQuery external function used in any XQuery expression in your application—in other words, for each XQS function that XQS will implement—that specifies details including the desired name of the XQS function that will be used in the query, the data source to access, and any input parameters. There are three basic configuration categories: for a document source, an XQS view, or a WSDL source. And there are three corresponding high-level configuration elements: `<document-source>`, `<xqsview-source>`, and `<wsdl-source>`.

Your configuration for any given XQS function can be either in the application-specific `xqs-config.xml` file, or, for the configuration to be available to any application running in the OC4J instance, in the `global-xqs-config.xml` file. In case of any duplication, the application-specific configuration file takes precedence. Duplicate or conflicting configuration in the global file would be ignored.

This section covers the following topics:

- Configuring an XQS Function That Accesses a Document Source
- Configuring an XQS Function That Uses an XQS View
- Configuring an XQS Function That Accesses a WSDL Source

See "Introduction to XQS Configuration and Configuration Files" on page 8-9 for related overview, including discussion of the local versus global XQS configuration file.

See "XQS Configuration File Reference" on page 8-86 for reference information about the configuration elements discussed here.

Configuring an XQS Function That Accesses a Document Source

This section discusses how to configure an XQS function that accesses a document source, concluding with examples. Elements mentioned here are subelements of the `<document-source>` element. For additional information, see the reference section "<document-source>" on page 8-88, which includes links to information about all the subelements.

Always Required

At a minimum, you must configure the following element for a document source:

- `<function-name>`: Through this element, specify the qualified name you want to use for the XQS function that XQS implements to access the document source. The element value is the local name, and there are attributes for the namespace (either `namespace` or `prefix`, as discussed in "<function-name>" on page 8-90). Your XQuery prolog must use the same name when declaring the XQS function as an XQuery external function.

  Here is a sample configuration:

  ```
  <document-source ... >
     <function-name namespace="http://xmlns.oracle.com/ias/xqs">
  ```

```
      myFileSource
   </function-name>
   ...
</document-source>
```

And here is a corresponding XQuery declaration and usage:

```
declare namespace xqs = "http://xmlns.oracle.com/ias/xqs";
declare function xqs:myFileSource() external;
for $po in xqs:myFileSource()//po
return $po
```

Optional or Sometimes Required

The following elements are sometimes necessary or appropriate for a document source:

- `<documentURL>`: Use this element to specify the URL of the document source. Here is an example:

  ```
  <document-source ... >
     ...
     <documentURL>
        http://host:port/xqsdemos/Repository/pos-2KB.xml
     </documentURL>
     ...
  </document-source>
  ```

  Important: For your specification of the document URL, be aware that if the document is on the local file system, then using `file://` protocol to specify the absolute path to the file, instead of using `http://` protocol, will give you faster data retrieval.

  If you provide a `<documentURL>` element, then you implicitly specify that your XQS function takes no arguments. Alternatively, you can omit `<documentURL>`, in which case you must pass the URL in to the XQS function at runtime, as in the following example. In this case, the function must be declared in XQuery to have one parameter for the URL.

  ```
  declare namespace xqs = "http://xmlns.oracle.com/ias/xqs";
  declare function xqs:myFileSource($bind as xs:string) external;
  for $po in xqs:myFileSource
           ("http://host:port/xqsdemos/Repository/pos-2KB.xml")//po
  return  $po
  ```

- `<output-element>`: You can use this subelement to specify the qualified name of the XML element or output type (either a simple type or a complex type) for data that is returned. This is not always required, but XQS can use the information for type checking. Here is an example:

  ```
  <document-source ... >
     ...
     <output-element namespace="http://xmlns.oracle.com/ias/xqs/CustomerDemo"
                  location="http://host:port/xqsdemos/Customers.xsd">
  type=Customers
     </output-element>
     ...
  </document-source>
  ```

For an <xqsview-source> element that defines <output-element>, the <output-element> subelement is required in every XQS function that defines an "errorMessage" error-handling option in the underlying query, or in a query that is nested via use of another <xqsview-source> function. For more information about this requirement, see "Using XQS Error Handling Modes and APIs" on page 8-69.

- `<XMLTranslate>`: For a non-XML document source, use this element to specify the conversion tool to use and to provide any information the tool needs. Currently only D3L is supported, which requires a D3L schema file to be specified through the `<schema-file>` subelement. See "Preparing to Use a Non-XML Document Source" on page 8-15 for information about D3L.

  Here is an example:

  ```
  <document-source ... >
     ...
     <XMLTranslate method="D3L">
        <schema-file>
           http://host:port/xqsdemos/paymentInfoD3L.xml
        </schema-file>
     </XMLTranslate>
     ...
  </document-source>
  ```

Performance and Error-Handling

You can set up error-handling and specify the use of performance features such as caching or special processing of large data sets. This would involve the `<document-source>` attributes `isCached`, `largeData`, and `onError`, and subelements `<cache-properties>` and `<error-message>`. Configuring these features is not discussed here. Refer to "Using XQS Error Handling Modes and APIs" on page 8-69 and "Using XQS Performance Features" on page 8-65.

Examples

These examples use default settings for caching, large data, and error handling, which is equivalent to `<document-source>` attribute settings of `isCached="false"`, `largeData="false"`, and `onError="dynamicError"`.

The following example is for an XML document source that resides in a fixed location:

```
<document-source>
   <function-name namespace="http://xmlns.oracle.com/ias/xqs">
      myFileSource
   </function-name>
   <documentURL>
      http://host:port/xqsdemos/Repository/pos-2KB.xml
   </documentURL>
</document-source>
```

The following example is for a document source in non-XML format, using the D3L conversion tool and `paymentInfoD3L.xml` D3L schema file:

```
<document-source>
   <function-name prefix="xqs">
      paymentStatusInfo
   </function-name>
   <documentURL>http://host:port/xqsdemos/paymentInfo.csv</documentURL>
   <XMLTranslate method="D3L">
      <schema-file>
         http://host:port/xqsdemos/paymentInfoD3L.xml
      </schema-file>
```

```
   </XMLTranslate>
</document-source>
```

Configuring an XQS Function That Uses an XQS View

This section discusses how to configure an XQS function that uses an XQS view, concluding with an example. Elements mentioned here are subelements of the `<xqsview-source>` element. For additional information, see the reference section "<xqsview-source>" on page 8-104, which includes links to information about all the subelements.

For preliminary steps, such as creating the XQS view that you will use, see "Preparing to Use an XQS View" on page 8-17.

WSDLvisibility Setting

You can use the `<xqsview-source>` attribute setting `WSDLvisibility="true"` to expose an XQS view as a Web service operation. XQS will add the operation to the WSDL that it generates. If the configuration is in the application-specific configuration file, `xqs-config.xml`, then the Web service operation is made part of the Web service of that application. A view that is configured in the global configuration file, `global-xqs-config.xml`, is potentially available to any application running in the OC4J instance. (To add an operation based on a global XQS view to the Web service of your application, you must point to the global configuration file when you run OC4JPackager. See "OC4JPackager Parameters" on page 8-106 for information about the `-globalXqsConfig` option.)

For more information about exposing an XQS view as a Web service, see "OC4JPackager Additional Output to Expose XQS Views as Web Service Operations" on page 8-62.

Always Required

For ensuing discussion, assume the following example has been saved as an XQS view in `mytotals.xq`. It uses a function that reads from a file, taking the input parameter `loc` (a string variable indicating a file location) and passing it through to the function, then returns an integer total:

```
declare variable $loc external;
declare namespace xqs = "http://xmlns.oracle.com/ias/xqs";
declare function xqs:readFile($l as xs:string) as xs:int external;
for $po in xqs:readFile($loc)//po return $po//total
```

At a minimum, you must configure the following elements to use an XQS view:

- `<function-name>`: Through this element, specify the qualified name you want to use for the XQS function that XQS implements to execute the view. The element value is the local name, and there are attributes for the namespace (either `namespace` or `prefix`, as discussed in "<function-name>" on page 8-90). If you use this function in a query (as opposed to using the view directly in an `executeView()` call), then you must declare the XQS function as an XQuery external function, using the function name specified in the `<function-name>` element.

  Here is a sample configuration:

  ```
  <xqsview-source ... >
     <function-name namespace="http://xmlns.oracle.com/ias/xqs">
        totals
     </function-name>
     ...
  ```

```
</xqsview-source>
```

- `<input-parameters>`: If an XQS view takes external variables, then the function implemented by XQS must take an input parameter for each external variable in the view. XQS will assign function argument values to external variables before executing the query. Use the `<input-parameters>` element to configure input parameters, with a `<part>` subelement for each parameter. Note that this element is always required for an XQS view; use an empty element if there are no input parameters:

  ```
  <xqsview-source ... >
     ...
     <input-parameters/>
     ...
  </xqsview-source>
  ```

  Following is an example with one input parameter being bound to an external string variable named `loc`. Type information, through a `<schema-type>` or `<xquery-sequence>` element, is required for any input parameter for an XQS view. Assume for the following example that the "`xs`" prefix has been set to correspond to the XMLSchema namespace. (See reference documentation later in this chapter for information about the `<input-parameters>`, `<part>`, `<schema-type>`, and `<xquery-sequence>` elements.)

  ```
  <xqsview-source ... >
     ...
     <input-parameters type-match="none" >
        <part position="1" name="loc">
           <schema-type prefix="xs">string</schema-type>
        </part>
     </input-parameters>
     ...
  </xqsview-source>
  ```

And here is an example of using the XQS view in a query, with the XQuery declaration and usage corresponding to the preceding `<function-name>` and `<input-parameters>` discussion:

```
declare namespace xqs = "http://xmlns.oracle.com/ias/xqs";
declare function totals($loc as xs:string) as xs:int external;
for $t in xqs:totals("C:\MyPurchaseOrders.xml") return
<outstanding_balance> fn:sum($t) </outstanding_balance>
```

Optional or Sometimes Required

The following elements are sometimes necessary or appropriate for an XQS view:

- `<output-element>`: Use this to specify the qualified name of the XML element or output type (either a simple type or a complex type) for data that is returned. For an XQS view with `WSDLvisibility="true"`, the `<output-element>` element is particularly advisable and its `location` attribute must point to the XML schema file containing the element or type definition. Here is an example:

  ```
  <xqsview-source WSDLvisibility="true">
     ...
     <output-element namespace="http://xmlns.oracle.com/ias/xqs/CustomerDemo"
                 location="http://host:port/xqsdemos/Customers.xsd">
     </output-element>
     ...
  </xqsview-source>
  ```

For an XQS view with `WSDLvisibility="false"`, the `<output-element>` element is still useful for type-checking. Assume the prefix "`xs`" has been set to correspond to the XMLSchema namespace:

```
<xqsview-source WSDLvisibility="false">
   ...
   <output-element prefix="xs">integer</output-element>
   ...
</xqsview-source>
```

- `<query-name>`: Use this to specify the name of the `.xq` file where you have saved the XQS view, with or without specifying the `.xq` extension (which XQS will append if necessary). The `<query-name>` element is optional. If you omit it, XQS will assume that the base name of the `.xq` file is the same as the XQS function name specified in the `<function-name>` element. Here is an example for an XQS view saved in `mytotals.xq`:

  ```
  <xqsview-source ... >
     ...
     <queryName>mytotals</queryName>
     ...
  </xqsview-source>
  ```

  Given the earlier example for the `<function-name>` element, if `<queryName>` were omitted, then XQS would look for a file named `totals.xq`.

- `<repository>`: Use this to specify where your XQS view repository is—the location of your `.xq` file. If you omit this element, XQS assumes the repository is as specified through the optional `-repository` option when you run OC4JPackager.

  Here is an example:

  ```
  <xqsview-source ... >
     ...
     <repository>META-INF/xqs/mydir</repository>
     ...
  </xqsview-source>
  ```

Considerations for Using <output-element >

When the `WSDLvisibility` attribute is set to `true`, XQS generates a Web service operation for the view and includes the definition of this operation in the WSDL document. The contents of `<output-element>` in `<xqsview-source>` determine the type of the output message for the WSDL operation, as follows:

- If <output-element> is omitted, the type of result in the WSDL operation output message will be declared as follows:

  ```
  <sequence> <any/> </sequence>
  ```

- If <output-element> is present, it should provide a namespace for the output element in the form of namespace or prefix attribute. The namespace results in the <import> element in the WSDL. For example:

  ```
  <output-element namespace="urn:PurchaseOrders"

  location="http://myhost:80/myapp/PurchaseOrders.xsd" />
  ```

  This code generates the following import element in the WSDL:

  ```
  <types>
         <schema...>
  ```

```
          <import namespace="urn:PurchaseOrders"
schemaLocation="http://myhost:80/myapp/PurchaseOrders.xsd" />
...
</schema>
</types>
```

Whenever possible, provide the location attribute because it allows a more complete `<import>` element in the WSDL.

- If `<output-element>` specifies a type attribute, it will be used in conjunction with a `namespace` or `prefix` attribute to create a qualified name for the type of the result element. The text value of `<output-element>` will be used as the name for the result element itself.

  For example:

```
<output-element namespace="urn:PurchaseOrders"

location="http://myhost:80/myapp/PurchaseOrders.xsd"
                                       type="POType" >
                             po
</output-element>
```

  This code results in an `<import>` element, as discussed in the preceding text, plus the following definition of the result type for the WSDL operation:

```
<complexType name="POswithRetTypeResultType">
...
<element name="result" nillable="true">
<complexType >
<sequence>
          <element name="po" xnlns:_ns1="urn:PurchaseOrders"
type="_ns1:POType" minOccurs="0" maxOccurs="unbounded" />
 </sequence>
...
```

- If `<output-element>` does not specify the `type` attribute, then the result element in the WSDL operation output message will be declared as a *reference* to an element in the imported schema. For example, consider the following configuration element:

```
<output-element namespace="urn:PurchaseOrders"

location="http://myhost:80/myapp/PurchaseOrders.xsd">
                             polist                        </output-element>
```

  Such an element generates a result definition in the WSDL like this:

```
<complexType name="POListResultType">
...
<element name="result" nillable="true">
<complexType >
<sequence>
             <element ref="_ns1:polist"   minOccurs="0" maxOccurs="unbounded"
/>

</sequence>
...
```

- If `<output-element>` specifies the `type` attribute but is empty (has no `text` element value), XQS will use a fixed name for the result element, `item`, but use the type name provided. For example, consider the following configuration:

```
<output-element namespace="urn:PurchaseOrders"
```

```
location="http://myhost:80/myapp/PurchaseOrders.xsd"
                                       type="POType" />
```

Such a configuration will generate a definition of the result element like this:

```
<complexType name="POsResultType">
...
<element name="result" nillable="true">
<complexType >
<sequence>
          <element name="item" xnlns:_ns1="urn:PurchaseOrders"
type="_ns1:POType" minOccurs="0" maxOccurs="unbounded" />
 </sequence>
```

Important: The type that defines the result element (through the `type` attribute in WSDL), or the element that defines the result element by reference (through the ref attribute in WSDL) must refer to a *top-level* type or element definition in the imported schema. In the previous examples, `POType` must be a top-level type definition in the imported schema for the namespace `urn:PurchaseOrders`; alternatively, `polist` (used to define an element by reference) must be defined at the top level of the imported schema at this location:

http://myhost:80/myapp/PurchaseOrders.xsd

Performance and Error-Handling

You can set up error-handling and specify the use of performance features such as caching or special processing of large data sets. This would involve the `<xqsview-source>` attributes `isCached`, `largeData`, and `onError`, and subelements `<cache-properties>` and `<error-message>`. Configuring these features is not discussed here. Refer to "Using XQS Error Handling Modes and APIs" on page 8-69 and "Using XQS Performance Features" on page 8-65.

Example

The example that follows puts together previous fragments to configure the XQS function `totals`, which takes a string variable `loc` (for a file location) and outputs an integer result. For convenience, here once again is the XQS view, `mytotals.xq`:

```
declare variable $loc external;
declare namespace xqs = "http://xmlns.oracle.com/ias/xqs";
declare function xqs:readFile($l as xs:string) external;
for $po in xqs:readFile($loc)//po return $po//total
```

This example uses default settings for caching, large data, and error handling, which is equivalent to `<xqsview-source>` attribute settings of `isCached="false"`, `largeData="false"`, and `onError="dynamicError"`. It does not expose the XQS view as a Web service.

```
<xqsview-source WSDLvisibility="false">
   <function-name namespace="http://xmlns.oracle.com/ias/xqs">
      totals
   </function-name>
   <input-parameters type-match="none" >
      <part position="1" name="loc">
         <schema-type prefix="xs">string</schema-type>
      </part>
   </input-parameters>
   <repository>META-INF/xqs/mydir</repository>
   <queryName>mytotals</queryName>
   <output-element prefix="xs">int</output-element>
</xqsview-source>
```

Configuring an XQS Function That Accesses a WSDL Source

This section discusses how to configure an XQS function that accesses a WSDL-based source, concluding with a partial sample WSDL document and corresponding configuration. Elements mentioned here are subelements of the `<wsdl-source>` element. For additional information, see the reference section "<wsdl-source>" on page 8-102, which includes links to information about all the subelements.

Depending on the type of WSDL source you are using, also see "Preparing to Use a WSDL Source with SOAP Binding" on page 8-18, "Preparing to Use a Database Source (WSDL Source with SQL Binding)" on page 8-19, or "Preparing to Use a Custom Class or EJB (WSDL Source with Java or EJB Binding)" on page 8-23.

Always Required

At a minimum, you must configure the following elements for a WSDL source:

- `<function-name>`: Through this element, specify the qualified name you want to use for the XQS function that XQS implements to access the WSDL source. The element value is the local name, and there are attributes for the namespace (either `namespace` or `prefix`, as discussed in "<function-name>" on page 8-90). Your XQuery prolog must use the same name when declaring the XQS function as an XQuery external function.

  Here is a sample configuration:

  ```
  <wsdl-source ... >
     <function-name namespace="http://xmlns.oracle.com/ias/xqs">
        getmySearchCachedPage
     </function-name>
     ...
  </wsdl-source>
  ```

- `<input-parameters>`: For every WSDL source, the function implemented by XQS must take an input parameter for each input part specified in the WSDL. Use the `<input-parameters>` element to configure input parameters, with a `<part>` subelement for each parameter. Note that this element is always required for a WSDL source; use an empty element if there is no input:

  ```
  <wsdl-source ... >
     ...
     <input-parameters/>
     ...
  </wsdl-source>
  ```

  Following is an example with two input parameters, external string variables named `key` and `url`. Type information, through a `<schema-type>` element, is optional for input parameters for a WSDL source; however, it is useful so that XQS can perform type-checking during invocation of the Web service. In the example, assume the "`xs`" prefix has been set to correspond to the XMLSchema namespace. (See reference documentation later in this chapter for information about the `<input-parameters>`, `<part>`, and `<schema-type>` elements.)

  ```
  <wsdl-source ... >
     ...
  ```

```
   <input-parameters>
      <part position="1" name="key" >
         <schema-type prefix="xs">string</schema-type>
      </part>
      <part position="2" name="url" >
         <schema-type prefix="xs">string</schema-type>
      </part>
   </input-parameters>
   ...
</wsdl-source>
```

- `<wsdlURL>`: Specify a URL that instructs XQS where to find the WSDL document. For example:

  ```
  <wsdl-source ... >
     ...
     <wsdlURL>http://api.mySearch.com/mySearch.wsdl</wsdlURL>
     ...
  </wsdl-source>
  ```

- `<operation>`: Specify the Web service operation to execute, corresponding to an operation name in the WSDL. For example:

  ```
  <wsdl-source ... >
     ...
     <operation>doGetCachedPage</operation>
     ...
  </wsdl-source>
  ```

- `<port>`: Specify the applicable Web service port, corresponding to a port name in the WSDL. For example:

  ```
  <wsdl-source ... >
     ...
     <port namespace="urn:mySearch">mySearchPort</port>
     ...
  </wsdl-source>
  ```

Here is a sample function declaration corresponding to the `<function-name>` and `<input-parameters>` examples above:

```
declare namespace xqs = "http://xmlns.oracle.com/ias/xqs";
declare function xqs:getmySearchCachedPage ($key as xs:string, $url as xs:string)
        as xs:base64Binary external;
```

Optional or Sometimes Required

The following elements are sometimes necessary or appropriate for a WSDL source.

- `<service>`: Use this to specify the applicable service, corresponding to a service name in the WSDL. This is not required, however, if the WSDL defines only one service. Here is a sample service configuration:

  ```
  <wsdl-source ... >
     ...
     <service namespace="urn:mySearch">mySearchService</service>
     ...
  </wsdl-source>
  ```

- `<portType>`: Use this to specify the applicable port type, corresponding to a port type name in the WSDL. This is not required, however, if the port in the WSDL supports only one binding (and, therefore, only one port type). Here is a sample port type configuration:

```
<wsdl-source ... >
   ...
   <portType namespace="urn:mySearch">mySearchPort</portType>
   ...
</wsdl-source>
```

- `<output-element>`: You can optionally use this to specify the qualified name of the XML output element or type (either a simple type or a complex type) for data that is returned. This is not required, but XQS can use the information for type-checking. Here is an example:

```
<wsdl-source ... >
   ...
   <output-element namespace="http://xmlns.oracle.com/ias/xqs/CustomerDemo"
                location="http://host:port/xqsdemos/Customers.xsd">
type="Customers>
   </output-element>
   ...
</wsdl-source>
```

- `<typeMap>`: For a WSDL source with Java or EJB binding, you can use this element to map XML types to Java types, using the `<mapping>` subelement and its `<xmlType>` subelement as shown. (See the reference documentation later in this chapter for more information about the `<typeMap>`, `<mapping>`, and `<xmlType>` elements.)

```
<wsdl-source ... >
   ...
   <typeMap>
      <mapping typeClass="org.w3c.dom.Node">
         <xmlType prefix="myeis">Customer</xmlType>
      </mapping>
      ...
   </typeMap>
   ...
</wsdl-source>
```

(A `<typeMap>` element would be ignored for a WSDL source with a SQL or SOAP binding.)

Performance and Error-Handling

You can set up error-handling and specify the use of caching. This would involve the `<wsdl-source>` attributes `isCached` and `onError`, and subelements `<cache-properties>` and `<error-message>`. Configuring these features is not discussed here. Refer to "Using XQS Error Handling Modes and APIs" on page 8-69 and "Configuring XQS Caching" on page 8-66.

Example

This example puts together some of the previous fragments to configure an XQS function for a Web service operation defined in `mySearch.wsdl`, a portion of which follows.

The example uses default settings for caching and error handling, which is equivalent to `<wsdl-source>` attribute settings of `isCached="false"` and `onError="dynamicError"`.

```
<wsdl-source>
   <function-name namespace="http://xmlns.oracle.com/ias/xqs">
      getmySearchCachedPage
   </function-name>
   <wsdlURL>http://api.mySearch.com/mySearch.wsdl</wsdlURL>
   <operation>doGetCachedPage</operation>
   <service namespace="urn:mySearch">mySearchService</service>
   <port namespace="urn:mySearch">mySearchPort</port>
   <input-parameters>
      <part position="1" name="key" >
         <schema-type prefix="xs">string</schema-type>
      </part>
      <part position="2" name="url" >
         <schema-type prefix="xs">string</schema-type>
      </part>
   </input-parameters>
</wsdl-source>
```

mySearch.wsdl

Here are relevant fragments of `mySearch.wsdl`, relating to the preceding configuration. Highlighted WSDL portions correspond to configuration elements.

```
<?xml version="1.0" ?>

<definitions name="mySearch" targetNamespace="urn:mySearch"
   xmlns:typens="urn:mySearch"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns="http://schemas.xmlsoap.org/wsdl/">
   ...
   <message name="doGetCachedPage">
      <part name="key" type="xs:string" />
      <part name="url" type="xs:string" />
   </message>
   <message name="doGetCachedPageResponse">
      <part name="return" type="xs:base64Binary" />
   </message>
   ...
   <portType name="mySearchPort">
      <operation name="doGetCachedPage">
         <input message="typens:doGetCachedPage" />
         <output message="typens:doGetCachedPageResponse" />
      </operation>
      ...
   </portType>
   ...
   <binding name="mySearchBinding" type="typens:mySearchPort">
      <soap:binding style="rpc"
                  transport="http://schemas.xmlsoap.org/soap/http" />
      <operation name="doGetCachedPage">
         <soap:operation soapAction="urn:mySearchAction" />
         <input>
            <soap:body use="encoded" namespace="urn:mySearch"
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
         </input>
         <output>
            <soap:body use="encoded" namespace="urn:mySearch"
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
         </output>
      </operation>
      ...
   </binding>
   ...
   <service name="mySearchService">
      <port name="mySearchPort" binding="typens:mySearchBinding">
         <soap:address location="http://api.mySearch.com/search/beta2" />
      </port>
   </service>
</definitions>
```

How to Design Your Queries

This chapter assumes developers are already familiar with the XQuery language and the basics of how to design an effective and efficient query, but this section will summarize key points to consider, including special considerations for working with XQS, and includes query examples. The following topics are covered:

- Query Considerations
- Query Examples
- Type-Checking for Input Parameters

Notes:

- Oracle fully supports XQuery 1.0, so you can use any syntax that XQuery supports.
- You can use any third-party tool to help you develop your queries, including the Oracle JDeveloper Query Builder, but currently no tools, including JDeveloper, provide any special support for XQS.

Query Considerations

As a first step, of course, you must consider your data source, including how to access it and the types of data it contains. "How to Prepare to Use Your Data Sources" on page 8-15 already discussed any preliminary steps you must take for certain kinds of data sources.

Then you must consider aspects of the query itself, including the following:

- Will your query require input parameters? If so, do you want type-checking? (See "Type-Checking for Input Parameters" on page 8-38.)
- What are the data types of any input parameters and the query return value?
- Will you want to transform the data, outputting it into another XML structure?
- Will you want to use an ad-hoc query, or save the query as an XQS view (`.xq` file)?
- Are there tuning and performance considerations, and will you want to use XQS performance features? XQS can cache source data, and also has a special mode for processing large volumes of data. (See "Using XQS Performance Features" on page 8-65 for details.)

As discussed previously, XQS implements an external XQuery function to access the data source. Your XQuery prolog must include a declaration for the function that XQS creates, using a function name you choose, and must reference the appropriate namespace. The function name you declare must match the name you specify in the `<function-name>` element when you configure the data source (as described under "How to Configure Your XQS Functions" on page 8-24).

Query Examples

Here is a simple query that retrieves data from an XML document, the location of which is passed in to the XQS function at runtime:

```
declare namespace xqs = "http://xmlns.oracle.com/ias/xqs";
declare function xqs:get_poSQ($bind as xs:string) external;
for $po in xqs:get_poSQ("http://host:port/xqsdemos/Repository/pos-2KB.xml")
return  $po
```

XQS implements the function according to your XQS configuration. The function name (`get_poSQ` in this example) is of your choosing. The function takes as input a string with the name and location of the document source, `pos-2KB.xml`, and returns purchase order data from the file.

Now here is an example that passes in parameter values through the XQuery code. The XQS function is `example`, implemented by XQS according to your configuration.

```
declare namespace xqs = "http://xmlns.oracle.com/ias/xqs";
declare function xqs:example($i as xs:int, $d as xs:duration,
                             $h as xs:hexBinary, $bin as xs:base64Binary,
                             $t as xs:boolean) external;

for $result in xqs:example(xs:int(1),
                           xs:duration("P1Y2MT2H"),
                           xs:hexBinary("0FB7"),
                           xs:base64Binary("vYrfOJ39673//-BDiIIGHSPM=+"),
                           xs:boolean("true"))
return $result;
```

Following is a more complicated example for a query that, given the name of a customer, searches a payment record for orders by that customer. It returns information about the customer and the customer's orders. There are two data sources involved: 1) customer information is in a data source accessed through a custom WSIF extension "myeis"; and 2) payment information is in an Excel spreadsheet (non-XML document source). To access the data source for customer information, the XQS function `customerInfo` is used, taking a string with the customer name as input. That function accesses the customer information data source and yields information for the specified customer, including a customer key. Then the payment information Excel file is accessed, using the XQS function `paymentStatusInfo`, and order information is returned for the customer whose key matches the key from the customer information data source. Note the XQuery code includes an XML transformation of the results.

```
declare namespace xqs = "http://xmlns.oracle.com/ias/xqs";
(: returns payment info for all customers:)
declare function xqs:paymentStatusInfo() external;

(: returns customer info given customer name :)
declare function xqs:customerInfo ($name as xs:string) external;

(: customer name passed in to query :)
declare variable $custName external;

let $custInfo := xqs:customerInfo($custName)
for $custOrderInfo in xqs:paymentStatusInfo()/excel/Row[CustomerKey eq
$custInfo/key]
return
   <result>
      <MYEIS_RESULT>
         <Row>
            <Name> { $custInfo/name } </Name>
            <Company> { $custInfo/company} </Company>
            <Address> { $custInfo/address} </Address>
            <City> { $custInfo/city} </City>
            <State> { $custInfo/state} </State>
            <Zip> { $custInfo/zip} </Zip>
         </Row>
      </MYEIS_RESULT>
      <EXCEL_RESULT>
         <Row>
            <OrderID>{$custOrderInfo/OrderId}</OrderID>
            <Amount>{$custOrderInfo/Amount}</Amount>
            <PaymentStatus>{$custOrderInfo/PaymentStatus}</PaymentStatus>
         </Row>
      </EXCEL_RESULT>
   </result>
```

Type-Checking for Input Parameters

For a WSDL source or XQS view, if you want XQuery to perform additional type-checking steps before sending input arguments to the underlying data source, then take the following steps:

1. For each input parameter, which is specified in your configuration for the applicable XQS function through a `<part>` subelement under the `<input-parameters>` element, specify the parameter type. For XQS view sources, this would be through one of the following, as appropriate:

   - A `<schema-type>` subelement of `<part>`
   - For an input sequence, an `<xquery-sequence>` subelement of `<part>`, with the `<itemType>` subelement of `<xquery-sequence>`

   For WSDL sources, which do not support a sequence as input, only `<schema-type>` is relevant.

   (See the reference documentation later in this chapter for details about all these elements.)

2. Include the parameter types with the function declaration in the XQuery prolog.
3. If input parameters are of user-defined types, import the schema or files containing the type definitions.

Here is an example. First, the XQS configuration:

```
<wsdl-source isCached="false">
   <function-name namespace="http://xmlns.oracle.com/ias/xqs">
      SplitRatio
   </function-name>
   ...
   <input-parameters>
      <part position="1" name="parameters">
         <schema-type namespace="http://www.xignite.com/services/">
            GetSplitRatio
         </schema-type>
      </part>
   </input-parameters>
</wsdl-source>
```

Then, the corresponding XQuery expression:

```
import schema namespace xignite="http://www.xignite.com/services/"
       at "http://www.xignite.com/xSecurity.asmx?wsdl"
declare namespace xqs="http://xmlns.oracle.com/ias/xqs";
declare function xqs:SplitRatio($params as xignite:GetSplitRatio)
        as xignite:GetSplitRatioResponse external;

let $in := <xignite:GetSplitRatio> .....</xignite:GetSplitRatio>

let $y := xqs:SplitRatio($in) return <split>$y//xignite:Ratio</split>
```

At execution time, XQuery will ensure that the argument passed into `xqs:SplitRatio` is of the type `xignite:GetSplitRatio` from the imported schema.

How to Develop Your Application Code: Using the XQS Client Interfaces

This section discusses the steps involved in using the client APIs that XQS provides (first discussed in "Introduction to XQS Client Interfaces" on page 8-10), and offers some examples. The focus is on the Java class client API, the EJB client API, and the JSP tag library. For Web service clients, the particulars of what XQS does to expose an XQS view as a Web service operation are left to the packaging discussion—see "OC4JPackager Additional Output to Expose XQS Views as Web Service Operations" on page 8-62. Otherwise, XQS generates a regular WSDL with SOAP 1.1 binding, and we assume readers are knowledgeable about creating Web service clients.

Note: XQS does not generate Java code for a Web service client. It is a responsibility of the user to invoke the Web service dynamically or generate the client invocation code based on the WSDL.

The following topics are covered here:

- Supported Types for Query Parameters
- General Coding Steps in Using XQS Client APIs
- Stateful Versus Stateless Clients
- Using the Java Class Client API
- Using the EJB Client API
- Using the JSP Tag Library
- Using an XQS View Exposed as a Web Service Operation

Notes:

- We assume that before you use any of the XQS client interfaces, you have prepared and configured your data source as described in "How to Prepare to Use Your Data Sources" on page 8-15 and "How to Configure Your XQS Functions" on page 8-24. We also assume you have completed any additional configuration, such as the `ejb-jar.xml` file if you use an EJB, or `web.xml` file if you use a Web module.
- Your choice of which client API to use is independent of the type of data source you use. These are entirely separate considerations.

Supported Types for Query Parameters

The XQS `oracle.xds.client.QueryParameter` class is for use with the XQS Java client API and EJB client API for arrays of bind parameters for queries. It is also used behind the scenes with the XQS JSP tag library. (See "XQS QueryParameter Class Reference" on page 8-75 for additional information.) Table 8–2 shows the correspondence between XML types that XQS supports, and Java types that you use to pass input values through instances of the `QueryParameter` class.

Table 8–2 XQS Type Support for Bind Parameters

Supported XML Type	Corresponding Java Type for QueryParameter Class
boolean	boolean
string	java.lang.String
int	int
integer	int, long, java.math.BigInteger
long	long
float	float
double	double
decimal	java.math.BigDecimal
base64Binary	java.lang.String
hexBinary	java.lang.String
anyURI	java.net.URI
dateTime	boolean, java.util.GregorianCalendar
duration	java.lang.String (lexical representation of duration)
anyType, user-defined XML types	org.w3c.dom.Node

General Coding Steps in Using XQS Client APIs

There are several basic coding steps in using the XQS client APIs:

1. Create your query. To use an ad-hoc query, this may consist of hard-coding a query or writing code to accept a query from user input. To use an XQS view, this consists of saving the query in an `.xq` file. Also see "How to Design Your Queries" on page 8-36.

2. Obtain an XQS client object, such as by creating the local interface to use the XQS EJB client API, by creating an `XQSFacade` instance to use the Java class client API, or by including the XQS JSP tag library in a JSP page.
3. Specify any input parameters. For the Java class API or EJB API, create an array of type `oracle.xds.client.QueryParameter`. (See "XQS QueryParameter Class Reference" on page 8-75.) For the JSP tag library, use the `param` subtag of `execute` or `executeCursor`. (See "XQS param Tag" on page 8-82.) Also see the preceding section, "Supported Types for Query Parameters".
4. Execute the query.
5. Read and process the results. For a stateless client, XQS either returns the results all at once in a `java.util.Vector` instance or, if you are using the JSP tags, returns a `java.util.ArrayList` instance and an XML document. For a stateful client, you retrieve the results item by item, either using the Oracle XQuery type `oracle.xml.xqxp.datamodel.XMLItem`, or, if you employ the JSP tags, using `java.lang.Object` instances and nodes of an XML document. For the XQS EJB API and JSP tag library, you have the option of using either a stateless or stateful access pattern.

   The Java class API (`XQSFacade`) uses a stateful access pattern. See "Using the Java Class Client API" on page 8-42 for information about using the `XQSFacade` class.
6. Optionally retrieve and process errors. If you configure XQS to continue even if errors are encountered during execution of the XQS function for the query, you can retrieve information about the errors. See "Using XQS Error Handling Modes and APIs" on page 8-69.
7. When using a stateful client, close the client object to free all resources associated with the query.

Note: This discussion does not include security or performance considerations. See "Security for XQS Applications" on page 8-12 and "Using XQS Performance Features" on page 8-65 for information on those topics.

Stateful Versus Stateless Clients

The XQS client APIs return query results as an XML sequence—one or more XML items, where each item may be an XML document, XML node, or primitive value. It is important to realize that the query execution defines the result, but does not necessarily materialize all result items into process memory immediately. An XML sequence, like a result set from a relational query, implicitly maintains a "cursor" that is initially positioned in front of the first item in the sequence and may be advanced by a "next" operation. If an XML sequence is accessed through the implicit cursor, through repeated calls to a "next" operation, then any of the "next" calls may trigger the actual evaluation of the XQuery expression in order to produce the next item requested. The exact moment when XQuery evaluates the expression and whether XQuery evaluates it incrementally or all at once depend on the specific XQuery expression and the level of optimization of the XQuery implementation. As an alternative to using this sort of cursor access, though, the XQS client APIs also offer reading all query results immediately into process memory in a single step.

Each of the single-step and cursor modes of execution has advantages and disadvantages. For most situations, single-step execution is preferable, because associated resources (such as connections to the underlying data sources, and internal

XQuery resources dedicated to the expression) are freed immediately after the evaluation is completed. You also do not have the potential performance impact of repeated trips to the data source. However, single-step execution is not feasible when the total size of items in the result sequence may be too large, such that materializing the whole sequence at one time would run out of process memory. In this case, cursor mode is the only option, as long as there is nothing (such as aggregate expressions, for example) preventing the query from providing the results item-by-item.

To allow the choice between the single-step and cursor modes, XQS provides two types of client APIs: *stateless* for single-step mode, or *stateful* for cursor mode. (In this discussion, "state" refers to the internal state of the XQuery engine.) These work as follows:

- When an XQuery expression is executed through a stateless API, the entire result sequence is materialized at once, and all internal state and resources are freed immediately afterward. The only requirement for stateless execution is that the entire result sequence must fit into process memory.
- By contrast, executing an XQuery expression through a stateful API results in placement of the "current" position in front of the first item in the result sequence. To obtain each item, starting with the first item, the client must invoke a "next" operation. The current position and all the internal resources necessary to evaluate the next item constitute the state of the query. This state is freed only after the last item in the sequence is returned, or if the query is explicitly closed. It is important to note, however, that to take advantage of the reduced memory requirements of stateful execution, the client code must release each result item once it has been processed.

The XQS EJB client API and JSP client API each has features for stateless execution and features for stateful execution. The XQS Java class (`XQSFacade`) client API provides stateful access. Users can simulate stateless access by immediately getting all result items and freeing the client object.

Using the Java Class Client API

Refer to the "General Coding Steps in Using XQS Client APIs" on page 8-40. Do as follows to apply these steps using the XQS Java class client API (after creating your query):

1. Create an `XQSFacade` instance. Method calls in the following steps are called from this instance.
2. Create a `QueryParameter` array for any input parameters.
3. Use the `execute()` method to execute an ad-hoc query, or the `executeView()` method to execute an XQS view, passing in the query and `QueryParameter` array (or `null` if there are no input parameters). For `executeView()`, also pass in the namespace of the XQS function for the view.
4. If you configured any data sources used in the query with the `emptySequence` or `errorMessage` error mode, optionally use the `getErrors()` method to retrieve any errors encountered during execution of the query. This returns an iterator over a collection of `XQSError` objects. See "Using XQS Error Handling Modes and APIs" on page 8-69 for information about error configuration and processing.
5. Repeatedly use the `getNextItem()` method to get the results back item by item. Each item is returned in an `XMLItem` instance. Process these items as desired.
6. Use the `close()` method to free all resources associated with the query.

These steps are shown in examples that follow. Also see "XQSFacade Class Reference" on page 8-77 for reference information.

Example 1: XQSFacade API with an Ad-Hoc Query

This example shows general use of the `XQSFacade` API. The query returns bind parameters in a sequence, showing how to bind and how to retrieve data. No configuration is necessary, because the query does not use any functions.

For processing query results, the code uses the `XMLItem` class (in package `oracle.xml.xqxp.datamodel`).

```
import oracle.xml.xqxp.datamodel.XMLItem;
import oracle.xml.parser.v2.XMLDocument;
import oracle.xml.parser.v2.XMLElement;
import oracle.xds.client.XQSFacade;
import oracle.xds.client.QueryParameter;
import oracle.xds.client.XQSError;
import oracle.xml.xqxp.functions.builtIns.FNUtil;
import java.util.Vector;

public class XQSFacadeTest {
public static void main(String[] args) throws Exception {
 try{
          //get XQSFacade
          XQSFacade facade = new XQSFacade();
          String xquery = "declare variable $bind1 external;\n"+
                "declare variable $bind2 external;\n"+
                "declare variable $bind3 external;\n"+
                "declare variable $bind4 external;\n"+
                "declare variable $bind5 external;\n"+
                "declare variable $bind6 external;\n"+
                "declare variable $bind7 external;\n"+
                "declare variable $bind8 external;\n"+
           "declare variable $bind9 external;\n"+
           "declare variable $bind10 external;\n"+
           "declare variable $bind11 external;\n"+
           "declare variable $bind12 external;\n"+
           "declare variable $bind13 external;\n"+
           "declare variable $bind14 external;\n"+
           "declare variable $bind15 external;\n"+
           "declare variable $bind16 external;\n"+

                " ("+
                "$bind1,$bind2,$bind3,$bind4,$bind5,$bind6,$bind7,\n"+
                "$bind8,$bind9,$bind10,$bind11,$bind12,$bind13,$bind14,$bind15,$b
ind16)";
          QueryParameter params[] = new QueryParameter[16];
          params[0] = new QueryParameter("bind1");
          params[0].setString("test");
          params[1] = new QueryParameter("bind2");
          params[1].setInteger(new BigInteger("100"));
          params[2] = new QueryParameter("bind3");
          params[2].setBoolean(true);
          params[3] = new QueryParameter("bind4");
          params[3].setFloat(-1);
          params[4] = new QueryParameter("bind5");
          params[4].setDuration("P1Y2M3DT10H30M");
          params[5] = new QueryParameter("bind6");
          params[5].setDouble(-11.0);
```

```
            TimeZone LAtz = new SimpleTimeZone(-28800000,
            "America/Los_Angeles",
                  Calendar.APRIL, 1, -Calendar.SUNDAY,
                  7200000,
                  Calendar.OCTOBER, -1, Calendar.SUNDAY,
                  7200000,
                  3600000);
       GregorianCalendar cal = new GregorianCalendar(LAtz);

       params[6] = new QueryParameter("bind7");
       params[6].setDateTime(cal,true);

       URI uri = new URI("http://www.test.com");
       params[7] = new QueryParameter("bind8");
       params[7].setAnyURI(uri);
       params[8] = new QueryParameter("bind9");
       params[8].setInt(-7);
       params[9] = new QueryParameter("bind10");
       params[9].setLong(50l);
       params[10] = new QueryParameter("bind11");
       params[10].setDecimal(new BigDecimal(999999));
       params[11] = new QueryParameter("bind12");
       XMLDocument document = new XMLDocument();
       Element elem = document.createElementNS("http://client.xqs.oracle/",
"tns:result");
       document.appendChild(elem);
       params[11].setNode(elem);
       params[12] = new QueryParameter("bind13");
       params[12].setBase64Binary("vYrfOJ39673//-BDiIIGHSPM=+");
       params[13] = new QueryParameter("bind14");
       params[13].setHexBinary("0FB7");
       params[14] = new QueryParameter("bind15");
       params[14].setDayTimeDuration(60.5);
       params[15] = new QueryParameter("bind16");
       params[15].setYearMonthDuration(13);
       facade.execute(xquery, params);
       // lookup functions
       XMLItem item = facade.getNextItem();

       while(item != null) {
         if (item.instanceOfType(XMLItem.XMLITEM_STRING)) {
            System.out.println("string item value: expected: \”test\”, actual:",
item.getString());
         }
         else if (item.instanceOfType(XMLItem.XMLITEM_INT)) {
            System.out.println("int item value: expected: \”-7\”, actual: ",
item.getInt());
         }
         else if (item.instanceOfType(XMLItem.XMLITEM_LONG)) {
            System.out.println("long item value: expected: \”501\”, actual: ",
item.getInt());
         }
         else if(item.instanceOfType(XMLItem.XMLITEM_INTEGER)) {
            if(item.intFormat())
              System.out.println("integer item value: expected: \”100\”, actual: "
, item.getInt());
            else
              System.out.println("integer item value: expected: \”100\”, actual:
", item.getDecimal().intValue());
```

```
         }
         else if(item.instanceOfType(XMLItem.XMLITEM_BOOLEAN)) {
            System.out.println("boolean item value: expected: \"true\", actual: ",
item.getBoolean());
         }
         else if(item.instanceOfType(XMLItem.XMLITEM_FLOAT)) {
            System.out.println("float item value: expected: \"-1\", actual:
",(float)item.getDouble());
         }
         else if (item.instanceOfType(XMLItem.XMLITEM_XDT_DAYTIMEDURATION)) {
            System.out.println("dayTimeDuration item value: expected: \"60.5
seconds\", actual: ", item.getDayTimeDuration());
         }
         else if (item.instanceOfType(XMLItem.XMLITEM_XDT_YEARMONTHDURATION)) {
            System.out.println("yearMonthDuration item value: expected: \"13
months\", actual: ",  item.getYearMonthDuration());
         }
         else if (item.instanceOfType(XMLItem.XMLITEM_DURATION)) {
            System.out.println("duration item value: expected: \" P1Y2M3DT10H30M
\", actual: ", item.getLexicalValue());
         }
         else if(item.instanceOfType(XMLItem.XMLITEM_DOUBLE)) {
            System.out.println("double item value: expected: \" -11.0\", actual:
", item.getDouble());
         }
         else if(item.instanceOfType(XMLItem.XMLITEM_DATETIME)) {
            System.out.println("dateTime item value: expected Timezone: \"Pacific
(Latz) \", actual: ",
                                                item.getCalendar().getTimeZone());
         }
         else if (item.instanceOfType(XMLItem.XMLITEM_ANYURI)) {
            System.out.println("anyURI item value: expected:
\"http://www.test.com\", actual: ", item.getString());
         }
         else if (item.instanceOfType(XMLItem.XMLITEM_DECIMAL)) {
            System.out.println("decimal item value: expected: \"999999 \", actual:
", item.getDecimal().intValue());
         }
         else if(item.instanceOfType(XMLItem.XMLITEM_NODE)) {
            XMLElement node = (XMLElement)item.getNode();
            //when obtained from document function, it returns XMLDocument
            if(node instanceof XMLDocument)
               node = (XMLElement)node.getFirstChild();
            System.out.println("node item value: expected namespace: \"
http://client.xqs.oracle \", actual: ",node.getNamespaceURI());
            System.out.println("node item value: expected local name: \"result\",
actual: ",node.getLocalName());
         }
         else if (item.instanceOfType(XMLItem.XMLITEM_HEXBINARY)) {
            System.out.println("hexBinary item value: expected: \"0FB7\", actual:
", item.getString());
         }
         else if (item.instanceOfType(XMLItem.XMLITEM_BASE64BINARY)) {
            System.out.println("base64Binary item value: expected: \"
vYrfOJ39673//-BDiIIGHSPM=+ \", actual: ",
                                                                item.getString());
         }
         else {
           System.out.println("item type not supported: "+item.getLexicalValue());
        }
```

```
            item = facade.getNextItem();
              }
        }catch(Exception ex){
              ex.printStackTrace();
              fail(ex.getMessage());
              assertTrue("xquery execution failed", false);
        }
        facade.close();
      }
    }
```

Example 2: XQSFacade API with an Ad-Hoc Query

This section has a Java class to show the steps of using the `XQSFacade` API in your client code, along with related XQS configuration. The example has an ad-hoc query that uses an XQS view source, which in turn uses a document source.

Query This example uses the following ad-hoc query:

```
declare namespace xqs="http://xmlns.oracle.com/ias/xqs";
declare function xqs:get_poSQ($bind as xs:string) external;
for $po in xqs:get_poSQ("http://localhost:8888/myrepository/pos-2KB.xml")
    return $po
```

Configuration The query in turn uses the XQS function `get_poSQ`, which corresponds to an XQS view source that is configured as shown immediately below. By default, if the `.xq` file name and location are not specified in the configuration (through the `<queryName>` and `<repository>` elements), the file name is assumed to match the function name, and the location is assumed to be as specified through the `-repository` option when you run OC4JPackager.

```
<xqsview-source>
   <function-name prefix="xqs">get_poSQ</function-name>
   <input-parameters>
      <part position="1" name="bind" >
         <schema-type prefix="xs">string</schema-type>
      </part>
   </input-parameters>
</xqsview-source>
```

The XQS view `get_poSQ`, defined in the file `get_poSQ.xq`, uses a document source and is defined as follows:

```
declare namespace xqs = "http://xmlns.oracle.com/ias/xqs";
declare variable $bind external;
declare function xqs:genericFile($bind as xs:string) external;
for $po in xqs:genericFile($bind)//po
   return $po
```

And the document source `genericFile` is configured as follows:

```
<document-source>
   <function-name namespace="http://xmlns.oracle.com/ias/xqs">
      genericFile
   </function-name>
</document-source>
```

The `genericFile` function takes the document URL as input, so there is no `<documentURL>` element.

Java Code The code follows—a complete class that does the following:

- An `XQSFacade` instance is created.
- The ad-hoc query, which uses the XQS view `get_poSQ`, is defined. The query is stored in the string `xqueryStr` after being pieced together in a string buffer `xqueryBuf`. The URL for the backend document source is taken by the `get_poSQ` function as input.
- The query is executed. There are no input parameters for the query, so `null` is passed in to the `execute()` method where the `QueryParameter` array would go. (The next example, "Example 3: XQSFacade API with an XQS View" on page 8-48, shows an input parameter.)
- No error processing is shown here, but if you have appropriate configuration of the source (error mode `errorMessage` or `emptySequence`), you could retrieve and process error objects. (The next example also shows error processing. Or see "Using XQS Error Handling Modes and APIs" on page 8-69 for additional information.)
- A `while` loop goes through the cursor item by item. Each item is a node, so is placed into an XML document, `doc`. From there, results can be displayed or further processed as desired (from that point, the processing is not XQS-specific).
- A `close()` call closes the cursor and frees associated resources.

```
import oracle.xml.xqxp.datamodel.XMLItem;
import oracle.xml.parser.v2.XMLDocument;
import oracle.xml.parser.v2.XMLElement;
import oracle.xml.parser.v2.XMLNode;
import oracle.xds.client.XQSFacade;

public class XQSFacadeTest {

    public static void main(String[] args) throws Exception{
        XQSFacade bean = new XQSFacade();
        StringBuffer xqueryBuf = new StringBuffer();
        xqueryBuf.append
             ("declare namespace xqs=\"http://xmlns.oracle.com/ias/xqs\";\n");
        xqueryBuf.append
             ("declare function xqs:get_poSQ($bind as xs:string) external;\n");
        xqueryBuf.append("for $po in xqs:get_poSQ
                     (\"http://localhost:8888/myrepository/pos-2KB.xml\") \n");
        xqueryBuf.append("return $po\n");

        String xqueryStr = xqueryBuf.toString();

        //no parameters to be bound, so pass null
        bean.execute(xqueryStr, null);

        XMLItem item = bean.getNextItem();

        XMLDocument doc = new XMLDocument();
        XMLElement rootElem = (XMLElement)doc.createElement("QueryResult");
        doc.appendChild(rootElem);

        while(item != null) {
            /* the test is actually superfluous - all items are expected to nodes
               for Purchase Order, here we show a standard treatment of items that
               are XML nodes, including entire documents
               (where node type is DOCUMENT_NODE)
            */
```

```
                if(item.getItemType().isNode()) {
                       rootElem.appendChild(doc.importNode(
                          item.getNode().getNodeType()==
                                             XMLNode.DOCUMENT_NODE?(XMLNode)
                          ((XMLDocument)item.getNode()).getDocumentElement():
                          (XMLNode)item.getNode(), true));
                }
                item = bean.getNextItem();
           }
           bean.close();
        }
    }
```

Example 3: XQSFacade API with an XQS View

This section has an example using the `XQSFacade` API to execute an XQS view directly through an `executeView()` call.

Configuration Here is the configuration for an XQS view named `BasicFileWS`:

```
<xqsview-source WSDLvisibility="true" isCached="false" onError="errorMessage">
   <function-name  prefix="xqs">BasicFileWS</function-name>
   <input-parameters type-match="none" >
      <part position="1" name="custName">
         <schema-type prefix="xs">string</schema-type>
      </part>
   </input-parameters>
   <output-element namespace="http://xmlns.oracle.com/ias/xqs/CustomerDemo"
                location="http://host:port/xqsdemos/MyTypes.xsd"
type="CustomerOrdersType">
   </output-element>
</xqsview-source>
```

There is no `<queryName>` element, so the `.xq` file name is assumed to be `BasicFileWS.xq` (matching the specified function name). The view takes a string parameter, `custName`, as input.

Here is the definition of the view `BasicFileWS.xq` (a query discussed earlier, in "Query Examples" on page 8-37), including the variable to take the customer name as input. This view uses two XQS functions, for data sources whose configurations are shown immediately following.

```
declare namespace xqs = "http://xmlns.oracle.com/ias/xqs" ;
(: returns payment info for all customers:)
declare function xqs:paymentStatusInfo () external;

(: returns customer info given customer name :)
declare function xqs:customerInfo ($name as xs:string) external;

(: customer name passed in to query :)
declare variable $custName external;

let $custInfo := xqs:customerInfo($custName)
for $custOrderInfo in xqs:paymentStatusInfo()/excel/Row[CustomerKey eq
$custInfo/key]
return
   <result>
      <MYEIS_RESULT>
         <Row>
            <Name> { $custInfo/name } </Name>
            <Company> { $custInfo/company} </Company>
            <Address> { $custInfo/address} </Address>
            <City> { $custInfo/city} </City>
            <State> { $custInfo/state} </State>
            <Zip> { $custInfo/zip} </Zip>
         </Row>
      </MYEIS_RESULT>
      <EXCEL_RESULT>
         <Row>
            <OrderID>{$custOrderInfo/OrderId}</OrderID>
            <Amount>{$custOrderInfo/Amount}</Amount>
            <PaymentStatus>{$custOrderInfo/PaymentStatus}</PaymentStatus>
         </Row>
      </EXCEL_RESULT>
   </result>
```

Here is the configuration for the document source associated with the function `paymentStatusInfo`. This is a non-XML document, so D3L is used for conversion. (See "Preparing to Use a Non-XML Document Source" on page 8-15.)

```
<document-source isCached="false">
   <function-name prefix="xqs">paymentStatusInfo</function-name>
   <documentURL>http://host:port/xqsdemos/paymentInfo.csv</documentURL>
   <XMLTranslate method="D3L">
      <schema-file>
         http://host:port/xqsdemos/paymentInfoD3L.xml
      </schema-file>
   </XMLTranslate>
</document-source>
```

And here is the configuration for the WSDL source associated with the function `customerInfo`. This includes some information for Java-to-XML type mapping.

```
<wsdl-source isCached="false">
   <function-name  namespace="http://xmlns.oracle.com/ias/xqs">
      customerInfo
   </function-name>
   <wsdlURL>http://host:port/xqsdemos/CustomerInfo.wsdl</wsdlURL>
   <operation>getCustomer</operation>
   <service prefix="myeis">CustomerInfoMYEISService</service>
   <port prefix="myeis">CustomerInfo</port>
   <input-parameters>
      <part position="1" name="name">
         <schema-type prefix="xs">string</schema-type>
      </part>
   </input-parameters>
   <typeMap>
      <mapping typeClass="org.w3c.dom.Node">
         <xmlType prefix="myeis">
            Customer
         </xmlType>
      </mapping>
   </typeMap>
</wsdl-source>
```

Java Code This section shows a complete class with Java code for executing the view `BasicFileWS`, accomplishing the following:

- Obtain an `XQSFacade` instance.
- Set up the `QueryParameter` array to input the customer name.

- Execute the view. The view name passed to the `executeView()` method must match the function name associated with the view in your configuration. Note this method also takes the namespace of the function (also indicated in your configuration). In this example, the view is declared in the XQS namespace.
- Process errors. Methods of the `XQSError` class are used to print out information about any errors encountered during the query. This assumes appropriate XQS error configuration to use the `emptySequence` or `errorMessage` mode; otherwise, an error will terminate the query and there will be no useful information in the errors iterator. In this example, as shown earlier in the `BasicFileWS` configuration, `errorMessage` mode is used. (See "Using XQS Error Handling Modes and APIs" on page 8-69 and "XQSError Class Reference" on page 8-85 for additional information.)
- Process results. Query results are obtained as instances of class `XMLItem` (in package `oracle.xml.xqxp.datamodel`). Type constants and type-specific accessors of the `XMLItem` class are used to retrieve values.
- Close the cursor to free resources associated with the query

Note: There is no need to declare the XQS function, `BasicFileWS`, when it is executed directly in an `executeView()` call.

```
import oracle.xml.xqxp.datamodel.XMLItem;
import oracle.xml.parser.v2.XMLDocument;
import oracle.xml.parser.v2.XMLElement;
import oracle.xml.parser.v2.XMLNode;
import oracle.xds.client.XQSFacade;
import oracle.xds.client.XQSError;
import oracle.xds.client.QueryParameter;

public class XQSFacadeViewTest {

    public static void main(String[] args) throws Exception{
        XQSFacade facade = new XQSFacade();

        String xqueryView = "BasicFileWS";
        QueryParameter param = new QueryParameter(null, "custName");
        param.setString("John Ford");
        QueryParameter[] params = new QueryParameter[1];
        params[0] = param;

        facade.executeView(xqueryView, "http://xmlns.oracle.com/ias/xqs", params);

        java.util.ListIterator errors = facade.getErrors();
        while(errors.hasNext()) {
            XQSError error = (XQSError)errors.next();
            System.out.println
               ("The function which gives error is: " + error.getFunctionQName());

            System.out.println("The error type is: "+error.getErrorType()+ " which
is - "+XQSError.typeNames(error.getErrorType()));
            System.out.println("The error code is:" + error.getErrorCode());
            System.out.println("The error message is:" + error.getErrorMessage());
        }

        XMLItem item = facade.getNextItem();
        StringBuffer resultBuf = new StringBuffer();
```

```
XMLDocument doc = new XMLDocument();
XMLElement rootElem = (XMLElement)doc.createElement("QueryResult");
doc.appendChild(rootElem);

while(item != null) {
if(item.instanceOfType(XMLItem.XMLITEM_STRING)) {
   resultBuf.append(item.getString());
}
else if(item.instanceOfType(XMLItem.XMLITEM_INT)) {
   resultBuf.append(item.getInt());
}
else if(item.instanceOfType(XMLItem.XMLITEM_LONG)) {
   resultBuf.append(item.getInt());
}
else if(item.instanceOfType(XMLItem.XMLITEM INTEGER)) {
   if(item.intFormat())
       resultBuf.append(item.getInt());
   else
       resultBuf.append(item.getDecimal().intValue());
}
else if(item.instanceOfType(XMLItem.XMLITEM_BOOLEAN)) {
   resultBuf.append(item.getBoolean());
}
else if(item.instanceOfType(XMLItem.XMLITEM_FLOAT)) {
   resultBuf.append((float)item.getDouble());
}
else if(item.instanceOfType(XMLItem.XMLITEM_DOUBLE)) {
   resultBuf.append(item.getDouble());
}
else if(item.instanceOfType(XMLItem.XMLITEM_XDT_DAYTIMEDURATION)) {
   resultBuf.append(item.getDayTimeDuration());
}
else if(item.instanceOfType(XMLItem.XMLITEM_YEARMONTHDURATION)) {
   resultBuf.append(item.getYearMonthDuration());
}
else if(item.instanceOfType(XMLItem.XMLITEM_DURATION)) {
   resultBuf.append(item.getLexicalValue());
}
else if(item.instanceOfType(XMLItem.XMLITEM_DATETIME)) {
   resultBuf.append(item.getCalendar().getTime());
}
else if(item.instanceOfType(XMLItem.XMLITEM_ANYURI)) {
   resultBuf.append(item.getString());
}
else if(item.instanceOfType(XMLItem.XMLITEM_DECIMAL)) {
   resultBuf.append(item.getDecimal().intValue());
}
else if(item.instanceOfType(XMLItem.XMLITEM_NODE)) {
   resultBuf.append(item.getNode().getNodeName());
}
else if(item.instanceOfType(XMLItem.XMLITEM_HEXBINARY)) {
   resultBuf.append(item.getString());
}
else if(item.instanceOfType(XMLItem.XMLITEM_BASE64BINARY)) {
   resultBuf.append(item.getString());
}


item = facade.getNextItem();
```

```
            }
            facade.close();
        }
    }
```

Using the EJB Client API

Refer to the "General Coding Steps in Using XQS Client APIs" on page 8-40. For the XQS EJB client API, there are differences in the details of these steps depending on whether you choose a stateful session or a stateless session, as follows (after creating your query):

1. Complete appropriate steps to create a bean instance, such as by executing the lookup, obtaining the EJB local home interface, and creating the local interface. Method calls in the following steps are called from this instance. See "EJB Clients for Stateful Versus Stateless Sessions" immediately below for how to create a stateful bean versus a stateless bean.
2. Create a `QueryParameter` array for any input parameters.
3. Use the `execute()` method to execute an ad-hoc query, or the `executeView()` method to execute an XQS view, passing in the query and `QueryParameter` array (or `null` if there are no input parameters). For `executeView()`, also pass in the namespace of the XQS function for the view.
4. For a stateful bean, repeatedly use the `getNextItem()` method to get the results back item by item. Each item is returned in an `XMLItem` instance. Process these items as desired.

   For a stateless bean, the query results are fully materialized as a vector, `java.util.Vector`, containing `XMLItem` instances. Process this vector of items as desired.
5. If you configured any of the data sources used in the query with the `emptySequence` or `errorMessage` error mode, optionally use the `getErrors()` method to retrieve any errors encountered within those data sources during execution of the query. This returns an iterator over a collection of `XQSError` objects. See "Using XQS Error Handling Modes and APIs" on page 8-69 for information about error configuration and processing.
6. For a stateful bean, use the `close()` method to free all resources associated with the query. This is not applicable for a stateless bean.

These steps are shown in examples that follow. Also see "XQS EJB Client API Reference" on page 8-78 for reference information.

EJB Clients for Stateful Versus Stateless Sessions

See "Stateful Versus Stateless Clients" on page 8-41 for information about when to use a stateful session and when to use a stateless one.

Obtaining a stateful EJB versus a stateless EJB is determined by your JNDI lookup and which XQS package you use. The following example will obtain a stateful bean:

```
//Look up and create the EJB to execute the query.
InitialContext ic = new InitialContext();
//Use Local client.
oracle.xds.client.ejb.stateful.XQSClientLocalHome home =
   (oracle.xds.client.ejb.stateful.XQSClientLocalHome)ic.lookup
                                     ("java:comp/env/XQSClientStatefulLocal");
oracle.xds.client.ejb.stateful.XQSClientLocal bean = home.create();
```

And this example will obtain a stateless bean:

```
//Look up and create the EJB to execute the query.
InitialContext ic = new InitialContext();
//Use Local client.
oracle.xds.client.ejb.stateless.XQSClientLocalHome home =
   (oracle.xds.client.ejb.stateless.XQSClientLocalHome)ic.lookup
                                      ("java:comp/env/XQSClientStatelessLocal");
oracle.xds.client.ejb.stateless.XQSClientLocal bean = home.create();
```

As noted earlier, when you execute a query (ad-hoc or view) in a stateless EJB session, results are fully materialized in a vector containing `XMLItem` instances, which you then process as desired. For a stateful session, after executing the query, you retrieve items one by one using the `getNextItem()` method, which returns items of type `XMLItem`.

Use of the EJB Client API in Stateful Sessions

The XQS EJB stateful client API is the same as the `XQSFacade` API. Beyond the code shown in the preceding section to create the bean, code examples for a stateful EJB client would look like what is shown for `XQSFacade` in "Example 2: XQSFacade API with an Ad-Hoc Query" on page 8-46 and "Example 3: XQSFacade API with an XQS View" on page 8-48. You loop to repeatedly use the `getNextItem()` method to retrieve and process items one by one, as `XMLItem` instances.

Example: EJB Client API with an XQS View in a Stateless Session

This stateless EJB example reuses the `BasicFileWS` XQS view used in "Example 3: XQSFacade API with an XQS View" on page 8-48, and repeats some of that code. Refer to that section to see the view and its configuration.

In addition to the JNDI lookup and creation of the stateless bean, the key difference between this stateless EJB example and an `XQSFacade` example is in the result processing. Results for a stateless session are fully materialized into a vector containing `XMLItem` instances. You then retrieve items one by one from there, casting them as `XMLItem`, as opposed to there being a `getNextItem()` method in the bean to retrieve `XMLItem` instances. The basic nature of the processing remains the same, however.

```
...
//lookup and create the EJB to execute the xquery
InitialContext ic = new InitialContext();

oracle.xds.client.ejb.stateless.XQSClientLocalHome home =
                 (oracle.xds.client.ejb.stateless.XQSClientLocalHome)
                 ic.lookup("java:comp/env/XQSClientStatelessLocal");
oracle.xds.client.ejb.stateless.XQSClientLocal bean = home.create();

String xqueryView = "BasicFileWS";
QueryParameter param = new QueryParameter("custName");
param.setString("John Ford");
QueryParameter[] params = new QueryParameter[1];
params[0] = param;

java.util.Vector  result =
     bean.executeView(xqueryView,"http://xmlns.oracle.com/ias/xqs",params);

java.util.Enumeration resultEnum = result.elements();
while(resultEnum!=null && resultEnum.hasMoreElements()) {
    XMLItem item = (XMLItem)resultEnum.nextElement();
```

```
        if(item.instanceOfType(XMLItem.XMLITEM_STRING)) {
           resultBuf.append(item.getString());
        }
        else if(item.instanceOfType(XMLItem.XMLITEM_INT)) {
           resultBuf.append(item.getInt());
        }
        else if(item.instanceOfType(XMLItem.XMLITEM_LONG)) {
           resultBuf.append(item.getInt());
        }
        else if(item.instanceOfType(XMLItem.XMLITEM_INTEGER)) {
           if(item.intFormat())
               resultBuf.append(item.getInt());
           else
               resultBuf.append(item.getDecimal().intValue());
        }
        else if(item.instanceOfType(XMLItem.XMLITEM_BOOLEAN)) {
           resultBuf.append(item.getBoolean());
        }
        else if(item.instanceOfType(XMLItem.XMLITEM_FLOAT)) {
           resultBuf.append((float)item.getDouble());
        }
        else if(item.instanceOfType(XMLItem.XMLITEM_DOUBLE)) {
           resultBuf.append(item.getDouble());
        }
        else if(item.instanceOfType(XMLItem.XMLITEM_XDT_DAYTIMEDURATION)) {
           resultBuf.append(item.getDayTimeDuration());
        }
        else if(item.instanceOfType(XMLItem.XMLITEM_YEARMONTHDURATION)) {
           resultBuf.append(item.getYearMonthDuration());
        }
        else if(item.instanceOfType(XMLItem.XMLITEM_DURATION)) {
           resultBuf.append(item.getLexicalValue());
        }
        else if(item.instanceOfType(XMLItem.XMLITEM_DATETIME)) {
           resultBuf.append(item.getCalendar().getTime());
        }
        else if(item.instanceOfType(XMLItem.XMLITEM_ANYURI)) {
           resultBuf.append(item.getString());
        }
        else if(item.instanceOfType(XMLItem.XMLITEM_DECIMAL)) {
           resultBuf.append(item.getDecimal().intValue());
        }
        else if(item.instanceOfType(XMLItem.XMLITEM_NODE)) {
           resultBuf.append(item.getNode().getNodeName());
        }
        else if(item.instanceOfType(XMLItem.XMLITEM_HEXBINARY)) {
           resultBuf.append(item.getString());
        }
        else if(item.instanceOfType(XMLItem.XMLITEM_BASE64BINARY)) {
           resultBuf.append(item.getString());
        }
// Here is where you might use a bean.getErrors() call and process errors.
// (Assuming appropriate configuration.)
}
...
```

Using the JSP Tag Library

Refer to the "General Coding Steps in Using XQS Client APIs" on page 8-40. For the XQS JSP tag library, there are differences in the details of these steps depending on whether you choose a stateful or stateless access mode, as follows (after creating your query):

1. Import the XQS JSP tag library through a `taglib` statement, specifying the TLD URI and desired prefix.
2. Use an `executeCursor` tag to execute a query in a stateful mode, or an `execute` tag to execute a query in a stateless mode. With either an `executeCursor` or `execute` tag:
   - Use the `xqueryString` attribute to specify an ad-hoc query, or the `xqsViewName` and `namespace` attributes to specify an XQS view.
   - Define output variables for results of the query: to an XML document (by using the `toXMLDoc` attribute) and to a `java.util.ArrayList` instance of results (by using the `resultItems` attribute). Also, you can optionally output to the JSP output stream (by using the `toWriter` attribute). Results will be represented by Java object types according to Table 8–2, " XQS Type Support for Bind Parameters" on page 8-40.
   - If you configured any of the data sources used in the query with the `emptySequence` or `errorMessage` error mode, optionally use the `errors` attribute to process any errors encountered within those data sources during execution of the query. This will give you an iterator over a collection of `XQSError` objects. See "Using XQS Error Handling Modes and APIs" on page 8-69 for information about error configuration and processing.
3. In either an `executeCursor` or `execute` tag, use a `param` subtag for each input parameter. Use the `param` tag `localName`, `namespace`, `value`, and `type` attributes as appropriate.
4. With the `executeCursor` tag, use the related `next` tag to retrieve results item by item or in batches and place them into the output vehicles.

   With the `execute` tag, results are fully materialized into the output vehicles.
5. Loop through one or more output vehicles to process the results. This is not XQS-specific, as the available output vehicles—JSP output stream, XML document, and list of Java objects—are standard. Be aware that results of each execution of `next` must be processed before `next` is executed again, because results are overwritten.
6. With the `executeCursor` tag, use the `close` tag to free all resources associated with the query. This is not applicable for the `execute` tag.

These steps are shown in examples that follow. Also see "XQS JSP Tag Library Reference" on page 8-79 for reference information.

Note: You can also use standard JSTL tags in a JSP page to transform results as desired.

JSP Tags for Stateful Versus Stateless Access

See "Stateful Versus Stateless Clients" on page 8-41 for information about when to use stateful access and when to use stateless access.

In a JSP page employing XQS, whether you use stateful access or stateless access is determined by which query tag you use. The `executeCursor` tag uses stateful access, while the `execute` tag uses stateless access.

As noted earlier, when you execute a query (ad-hoc or view) in a stateless JSP access pattern, results are fully materialized into the output vehicles you choose through your attribute settings—an XML DOM document and an `Object[]` array, and optionally the JSP output stream—and then you process those results as desired. For a stateful access pattern, after executing the query, you retrieve and process items one by one using the `next` tag.

Example: JSP Tags with an XQS View in a Stateful Access Pattern

This stateful JSP example reuses the `BasicFileWS` XQS view used in "Example 3: XQSFacade API with an XQS View" on page 8-48. Refer to that section to see the view and its configuration.

The code shown below does the following:

- The `page` directives import required classes.
- A `taglib` directive specifies the TLD URI of the tag library, and the desired tag prefix to use.
- The `executeCursor` tag executes the view, resulting in an open cursor ready for fetching. The `namespace` attribute specifies the namespace where the underlying XQSView function is defined.
- The `param` subtag of `executeCursor` specifies the input parameter (customer name).
- The `next` tag, associated with the `executeCursor` tag through the `cursorId` setting, loops through the results to populate the output vehicles (XML document and result items `ArrayList` instance), using the `itemsFetched` attribute each time through as a test of whether any data remains to be processed. The `next` tag must reference the cursor ID (`mycursor`) specified in the `executeCursor` tag.
- The `close` tag closes the cursor and frees resources associated with it. This tag must also reference the cursor ID (`mycursor`) specified in the `executeCursor` tag.

Notes:

- Error processing is not shown here. Assuming appropriate error configuration (`emptySequence` or `errorMessage` mode in the data source configuration), you could process the `myerrors` iterator. See "Using XQS Error Handling Modes and APIs" on page 8-69.
- Processing of the output vehicles is not shown here; there is nothing specific to XQS about processing an XML document or array of result objects.

```
<%@ page import="oracle.xml.parser.v2.XMLElement" %>
<%@ page import="oracle.xml.parser.v2.XMLDocument" %>

<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/xquerytag.tld"
           prefix="xq"%>
<%
String xqueryView = "BasicFileWS";
int items;
%>
<xq:executeCursor
                xqsViewName="<%=xqueryView%>"
                namespace="http://xmlns.oracle.com/ias/xqs"
                resultItems="myresults"
                toXMLDoc="mydoc"
                cursorId="mycursor"
                errors="myerrors" >
   <xq:param
              localName="custName"
              value="John Ford"
              type="String"/>
</xq:executeCursor>
<%
do{
// Populate the XML document and result items array.
%>
   <xq:next
                cursorId="mycursor"
                itemsFetched="fetched" />
<%
  items=fetched.intValue();
 }while(items>0)
// Here is where you might process the "myerrors" error iterator.
// (Assuming appropriate configuration.)

// Retrieve results from XML document mydoc and/or result items array myresults.
// (This processing is not XQS-specific.)
%>
<xq:close cursorId="mycursor" />
```

Example: JSP Tags with an Ad-Hoc Query in a Stateless Access Pattern

This example uses the XQS JSP `execute` tag to execute an ad-hoc query, and a stateless access pattern is used. The intent is primarily to compare and contrast the following, compared to the preceding example using stateful access:

- Use an ad-hoc query instead of a view. See "Example 2: XQSFacade API with an Ad-Hoc Query" on page 8-46, which uses the same query, for relevant configuration. The query is pieced together in a string buffer, then written from there to a string.
- Use the `execute` tag, for stateless access, instead of the `executeCursor` tag, for stateful access. Note that there is no `next` subtag for an `execute` tag, and the `close` tag is not required.

```
<%@ page import="oracle.xml.xqxp.datamodel.XMLItem" %>
<%@ page import="oracle.xml.parser.v2.XMLElement" %>
<%@ page import="oracle.xml.parser.v2.XMLDocument" %>

<%@ taglib uri="http://xmlns.oracle.com/j2ee/jsp/tld/ojsp/xquerytag.tld"
           prefix="xq"%>
<%
StringBuffer xqueryBuf = new StringBuffer();
xqueryBuf.append("declare namespace xqs=\"http://xmlns.oracle.com/ias/xqs\";\n");
xqueryBuf.append("declare function xqs:get_poSQ($bind as xs:string) external;\n");
xqueryBuf.append("for $po in
xqs:get_poSQ(\"http://localhost:8888/myrepository/pos-2KB.xml\") \n");
xqueryBuf.append("return $po\n");

String xqueryStr = xqueryBuf.toString();
%>
<xq:execute
                xqueryString="<%=xqueryStr%>"
```

```
                 toXMLDoc="mydoc"
                 resultItems="myresults"
                 errors="myerrors" />
<%
// Here is where you might process the "myerrors" error iterator.
// (Assuming appropriate configuration.)
%>
<%
// Retrieve the results from either the XML document mydoc or result items array
// myresults. (This processing is not XQS-specific.)
%>
```

Using an XQS View Exposed as a Web Service Operation

XQS can expose an XQS view as a Web service operation, with the Web service being implemented through a servlet that is generated automatically. This and related details are described in "OC4JPackager Additional Output to Expose XQS Views as Web Service Operations" on page 8-62.

Note that you must provide the Web service client yourself.

How to Use OC4JPackager to Package Your XQS Application

This section describes how to use OC4JPackager to bundle XQS-related files with your application. See "Introduction to OC4JPackager" on page 8-11 for an overview.

OC4JPackager produces an EAR file which you can deploy to OC4J as you would any other EAR file. See the ***Oracle Containers for J2EE Deployment Guide*** for general information.

The following topics are covered here:

- Steps in Using OC4JPackager
- Running OC4JPackager on the Command Line
- Running OC4JPackager Through Ant
- OC4JPackager Basic Output
- OC4JPackager Additional Output to Expose XQS Views as Web Service Operations

Steps in Using OC4JPackager

This section covers the basic preparatory and execution steps when you have a J2EE application and want to run OC4JPackager to enable it to use XQS features. See "OC4JPackager Reference" on page 8-106 for general information about the OC4JPackager parameters and Java properties mentioned here.

Note: OC4JPackager does not support directory or file names with spaces.

Preparing to Run OC4JPackager

Complete these steps to prepare to run OC4JPackager:

1. Create JAR files for your application components as usual—WAR files for Web modules, EJB JAR files for EJB modules, and so on, all with the applicable standard J2EE configuration files (such as `web.xml` and `ejb-jar.xml`). You then

have the option of bundling them into an EAR file for OC4JPackager to access, or leaving them and any other relevant files (such as `application.xml`) in a directory structure reflecting the structure and contents of an EAR file.

2. Choose a desired directory for your application, and place the EAR file or structured EAR contents (as applicable) in that directory.

   You will specify that directory to OC4JPackager through the `-appArchives` parameter (required).

3. Choose a desired directory as your XQS repository, where any XQS view (`.xq`) files are located, and place the XQS view (`.xq`) files there.

   You can specify that directory to OC4J Packager through the `-repository` parameter.

4. Assuming you have completed your XQS configuration (see "How to Configure Your XQS Functions" on page 8-24), choose a desired directory for the `xqs-config.xml` file and place the file in that directory.

   You will specify that directory to OC4JPackager through the `-xqsConfig` parameter (required).

5. If you would like to expose XQS view sources in the global configuration as Web services through your application, choose a desired directory for the `global-xqs-config.xml` file and place the file in that directory.

   You will specify that directory to OC4JPackager through the `-globalXqsConfig` parameter.

Running OC4JPackager: Required and Optional Parameters and Properties

Specify the following, as applicable, when you run OC4JPackager. See the next section, "Running OC4JPackager on the Command Line", for examples.

Program Parameters

1. Use the required `-appArchives` parameter to specify the directory where your application is located (where the EAR file is, or the top-level directory of an EAR structure).

2. Use the optional `-repository` parameter to specify the directory for your XQS repository.

3. Use the required `-xqsConfig` parameter to specify the full path of your application `xqs-config.xml` file, including the file name.

4. If the global XQS configuration file includes configuration for XQS views and you would like to expose XQS view sources in the global configuration as Web services through your application, use the `-globalXqsConfig` parameter to specify the full path of the `global-xqs-config.xml` file, including the file name.

5. Use the required `-name` and `-output` parameters to specify the desired name and output directory of the EAR file that OC4JPackager will produce.

6. Use applicable flags to have OC4JPackager include files for XQS features that your application uses:

   - Use the `-jsp` flag if you use the XQS JSP tag library.
   - Use the `-sf` flag if you use the XQS stateful EJB client API.
   - Use the `-sl` flag if you use the XQS stateless EJB client API.

   (The `XQSFacade` class is always included.)

Java VM Properties

1. Use the required Java property `java.home` to specify your Java home directory, from which OC4JPackager will run the `java` command for its tasks.
2. Use the optional Java property `xds.packager.work.dir` if you want to have OC4JPackager use a particular working directory (where it unbundles and bundles EAR files as needed). The default is the directory specified by the `user.home` Java property,.
3. Use the optional Java property `java.util.logging.properties.file` to specify a logging properties file where you can specify Java logging settings, including the OC4JPackager logging level. Logging properties are defined by standard J2SE logging, as described in the following guide:

   `http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/`

4. Use the optional Java property `oracle.home` if you want to indicate your Oracle home directory. You must do this to use OC4JPackager flags `-jsp`, `-sf`, and `-sl` (described in "OC4JPackager Parameters" on page 8-106) and related JAR files.

Running OC4JPackager on the Command Line

You can run OC4JPackager from a command line by using the `java` command to execute `OC4JPackager.jar` (where `%` is the command prompt):

```
% java -jar OC4JPackager.jar ...
```

`OC4JPackager.jar` is in the `xds/tools` directory of your XQS distribution.

Here is an example (where `$ORACLE_HOME` has been set as the home directory of your Oracle installation, and "\" indicates line wrap). This is for an application that uses the XQS JSP tag library and stateful EJB client API, and has no relevant configuration in the XQS global configuration file. Note that you can specify paths that are either relative to your current directory, or absolute.

```
% java -jar OC4JPackager.jar \
       -Djava.home=/home/myjavainstall \
       -Dxds.packager.work.dir=$ORACLE_HOME/temp \
       -Doracle.home=$ORACLE_HOME \
       -Djava.util.logging.properties.file=myfile.properties \
       -appArchives $ORACLE_HOME/myapp \
       -xqsConfig $ORACLE_HOME/xds/myconfig/xqs-config.xml \
       -repository $ORACLE_HOME/xds/repository \
       -output $ORACLE_HOME/xds/mystaging -name myxqsapp.ear \
       -jsp -sf
```

Alternatively, you can run OC4JPackager through Ant, as described in the next section.

Also see "OC4JPackager Basic Output" on page 8-61.

Running OC4JPackager Through Ant

As an alternative to running OC4JPackager from a command line, as described in the preceding section, you can run it as part of a build process using Ant, through the standard Ant `java` task. Refer to the Apache Web site `http://ant.apache.org/manual/` for general information about Ant. (In addition, if you are interested, see the ***Oracle Containers for J2EE Deployment Guide*** for information about OC4J-specific Ant tasks.)

The following example from an Ant build file executes OC4JPackager, and corresponds to the command-line example shown in the preceding section:

```
<java jar="OC4JPackager.jar" fork="true" failonerror="true">
   <jvmarg value="-Doracle.home=${ORACLE_HOME}"/>
   <jvmarg value="-Djava.home=/home/myjavainstall"/>
   <jvmarg value="-Dxds.packager.work.dir=${ORACLE_HOME}/temp"/>
   <jvmarg value="-Djava.util.logging.properties.file=myfile.properties"/>
   <arg line="-appArchives ${ORACLE_HOME}/myapp"/>
   <arg line="-xqsConfig ${ORACLE_HOME}/xds/myconfig/xqs-config.xml"/>
   <arg line="-repository ${ORACLE_HOME}/xds/repository"/>
   <arg line="-output ${ORACLE_HOME}/xds/mystaging" />
   <arg line="-name myxqsapp.ear"/>
   <arg line="-jsp"/>
   <arg line="-sf"/>
</java>
```

Important: In the `<arg>` elements to set OC4JPackager parameters, use the `line` attribute, as shown, as opposed to the `value` attribute.

OC4JPackager Basic Output

The EAR file produced by OC4JPackager has the standard structure, with at least one additional component—a file named `xqs-resources.jar` that contains the XQS configuration file and a directory with the XQS repository files (`.xq` files for XQS views).

Note: The EAR file will contain additional components if you have XQS views that you are exposing as Web service operations. See the next section, "OC4JPackager Additional Output to Expose XQS Views as Web Service Operations".

For an application consisting of a Web module `mywebapp` and EJB module `myejb`, here are the contents of the EAR file that OC4JPackager would produce:

```
META-INF/
   application.xml
   orion-application.xml
   data-sources.xml
mywebapp.war
myejb.jar
xqs-resources.jar
```

Aside from `xqs-resources.jar`, your application must already include the `application.xml` file, plus optional `orion-application.xml` and `data-sources.xml` files, if needed, WAR files for any Web modules, EJB JAR files for any EJB modules, and so on.

The structure of `xqs-resources.jar` is as follows:

```
   APP-REPOSITORY/
      filename.xq
      filename.xq
      filename.xq
      ...
   xqs-config.xml
   xds-application.properties
```

The file `xds-application.properties` is for internal use by XQS.

OC4JPackager Additional Output to Expose XQS Views as Web Service Operations

For any XQS view with a setting of `WSDLvisibility="true"` in the XQS configuration, XQS exposes the view as a Web service operation. (You must have a Web service client to invoke the operation, as for any Web service.)

OC4JPackager creates an implementation of the Web service and a WSDL document that defines an operation for each XQS view to be exposed. OC4JPackager then produces a WAR file with contents including the following:

- The WSDL document
- Web module configuration file `web.xml`, to configure the Web service servlet
- Web services configuration file `oracle-webservices.xml`

OC4JPackager also updates the `application.xml` file for your application, to make an entry for the additional Web module.

The name of the additional Web module, representing the XQS View Web service, as well as the context root and the WAR file correspond to the name of your application, such as the following for an application `myapp`:

- `/myapp-WS`—the context root of the Web services module
- `myapp-WS-web.war`

The generated WSDL file is always names `xqsview-WS.wsdl`.

The URL pattern for the servlet representing the Web service (and leading to the Oracle Web service test page) is always `/xqs-ws`.

Thus, the URL for the WSDL of the XQS view Web service is `http://`*`host`*`:`*`port`*`/myapp-WS/xqs-ws?WSDL`.

Note: The additional WAR file does *not* contain client invocation code for Web services. It is intended to be part of your application deployment on the server.

In the WSDL document, the `wsdl:operation` will have the same name as the corresponding XQS function name defined in the XQS configuration (in the `<function-name>` subelement of `<xqsview-source>`). Each `wsdl:input` message type maps to the XML types corresponding to the input parameter types defined in the XQS configuration (under the `<input-parameters>` subelement of `<xqsview-source>`). The `wsdl:output` message type maps to the XML element or type corresponding to the output element defined in the XQS configuration (in the `<output-element>` subelement of `<xqsview-source>`). These can be either simple types such as `string` or `int`, or complex types defined by users. The bindings will be SOAP doc/wrapped bindings.

The following sections together provide an example:

- Example: Configuration to Expose a View as a Web Service Operation
- Example: EAR File for a View Exposed as a Web Service Operation
- Example: WAR File for a View Exposed as a Web Service Operation
- Example: WSDL document for a View Exposed as a Web Service Operation

Example: Configuration to Expose a View as a Web Service Operation

The following configuration, seen earlier in this chapter, exposes the view `BasicFileWS` as a Web service operation, through the setting `WSDLvisibility="true"`.

```
<xqsview-source WSDLvisibility="true"  isCached="false">
   <function-name  prefix="xqs">BasicFileWS</function-name>
   <input-parameters type-match="none" >
      <part position="1" name="custName">
         <schema-type prefix="xs">string</schema-type>
      </part>
   </input-parameters>
   <output-element namespace="http://xmlns.oracle.com/ias/xqs/CustomerDemo"
                location="http://host:port/xqsdemos/MyTypes.xsd"
type=CustomerOrdersType>customerOrders </output-element>
</xqsview-source>
```

Assume, for ensuing discussion, that this is part of an application called `myapp`.

Example: EAR File for a View Exposed as a Web Service Operation

Assume an application, `myapp`, comprises `mywebapp.war` and `myejb.jar` and exposes one or more XQS views as Web service operations. Here are the relevant contents of the application EAR file after we run OC4JPackager:

```
META-INF/
   application.xml
mywebapp.war
myejb.jar
myapp-WS-web.war
xqs-resources.jar
```

The WAR file `myapp-WS-web.war` is for the automatically created servlet implementation of the Web service operations for any exposed views. See the next section for its contents.

Example: WAR File for a View Exposed as a Web Service Operation

Here are the contents of `myapp-WS-web.war`, created for a servlet implementation of the Web service operations for any exposed views:

```
WEB-INF/
   web.xml
   oracle-webservices.xml
   wsdl/
      xqsview-WS.wsdl
```

In particular, note the WSDL document. Sample fragments of the WSDL are shown in the next section, "Example: WSDL document for a View Exposed as a Web Service Operation".

Example: WSDL document for a View Exposed as a Web Service Operation

This section shows fragments of a WSDL document generated through use of OC4JPackager for an XQS view being exposed as a Web service. This example focuses on fragments relating to a view named `BasicFileWS`.

```
...
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:__xqscws="xqs-client-ws"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xqs0="http://xmlns.oracle.com/ias/xqs" xmlns:tns="xqs-
    client-ws_myapp-WS" name="myapp-WS" targetNamespace="xqs-
    client-ws_myapp-WS">
...
<types>
...
<schema targetNamespace="http://xmlns.oracle.com/ias/xqs"
xmlns:xqs1="http://xmlns.oracle.com/ias/xqs/CustomerDemo"
xmlns:tns="http://xmlns.oracle.com/ias/xqs"
...
>
<import namespace="http://xmlns.oracle.com/ias/xqs/CustomerDemo"
schemaLocation="http://host:port/xqsdemos/MyTypes.xsd"/>

<complexType name="BasicFileWSType">
      <sequence>
          <element name="custName" type="string"
              nillable="true" />
      </sequence>
   </complexType>
   <complexType name="BasicFileWSResultType">
       <sequence>
         <element name="return" nillable="true">
           <complexType>
             <sequence>
               <element name="result" nillable="true">
                 <complexType>
                   <sequence>
                     <element name="customerOrders"
type="xqs1:CustomerOrdersType"
minOccurs="0" maxOccurs="unbounded" />
                    </sequence>
                  </complexType>
               </element>
             <element name="errors" nillable="true" minOccurs="0">
               <complexType>
                 <sequence>
                    <element name="xqserror"
                      type="__xqscws:XQSErrorType"
                      maxOccurs="unbounded" />
                  </sequence>
                </complexType>
              </element>
            </sequence>
          </complexType>
        </element>
      </sequence>
    </complexType>
<element name="BasicFileWS" type="tns:BasicFileWSType" />
<element name="BasicFileWSResult" type="tns:BasicFileWSResultType" />

…
</schema>
…
</types>
…
<message name="BasicFileWSRequest">
 <part name="parameters" element="xqs0:BasicFileWS" />
</message>
<message name="BasicFileWSResponse">
 <part name="return" element="xqs0:BasicFileWSResult" />
```

```
</message>
…
<portType name="XQSViewWebServices">
   <operation name="BasicFileWS">
      <input message="tns:BasicFileWSRequest" />
      <output message="tns:BasicFileWSResponse" />
    </operation>
    …
    </portType>
      <binding name="HttpSoap11Binding"  type="tns:XQSViewWebServices">
          <soap:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http" />
        <operation name="BasicFileWS">
           <soap:operation />
          <input>
              <soap:body use="literal" parts="parameters" />
           </input>
          <output>
              <soap:body use="literal" parts="return" />
           </output>
         </operation>
         …
    </binding>
   <service name="XQSView-WS">
       <port name="HttpSoap11" binding="tns:HttpSoap11Binding">
           <soap:address xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
             location="http://host:port/myapp-WS/xqs-ws" />
       </port>
   </service>
</definitions>
```

Using XQS Performance Features

This section discusses performance features that XQS offers. The following topics are covered:

- Performance Considerations for Using the XQS Stateless or Stateful Client APIs
- Configuring XQS Caching
- Configuring XQS Document or View Sources for Large Data
- XQS XPath Optimization for WSDL Sources with SQL Binding

Performance Considerations for Using the XQS Stateless or Stateful Client APIs

As noted previously, the XQS client APIs give you the choice of stateless query execution, where results are fully materialized into process memory, or stateful query execution, where results are accessed incrementally through a cursor. Each approach has advantages and disadvantages. For most situations, stateless execution is preferable because associated resource are freed immediately after the evaluation is completed. However, stateless execution is not feasible when the total size of items in the result sequence may be too large, such that materializing the whole sequence at one time would run out of process memory. In this case, cursor mode is the only option, as long as there is nothing (such as aggregate expressions, for example) preventing the query from providing the results item-by-item.

See "Stateful Versus Stateless Clients" on page 8-41 for additional information.

Configuring XQS Caching

This section covers the details of how to configure the cache when using XQS, and discusses related considerations:

- Configure XQS Cache Settings
- XQS Caching Strategies
- Caching and Nondeterministic Results

Important: The XQS 10.1.3 implementation uses the OC4J Java Object Cache for its caching. The Java Object Cache must be running (and will be started automatically if necessary), and its configuration file, `ORACLE_HOME/j2ee/home/config/javacache.xml`, must be present. There is generally no need for you to update this file directly, but you have that option. See Chapter 7, "Java Object Cache," for more information.

Be aware that any configuration of the Java Object Cache applies across the OC4J instance, not just to your XQS application.

Configure XQS Cache Settings

XQS caching is configured individually for each XQS function (essentially, for each data source that you access). Use the steps that follow to employ caching for any particular XQS function.

Strategies to consider are discussed in the next section, "XQS Caching Strategies". Complete documentation for all the elements and attributes mentioned here is under "XQS Configuration File Reference" on page 8-86.

1. Enable caching through the `isCached` attribute of the `<document-source>`, `<xqsview-source>`, or `<wsdl-source>` element. For example:

```
<document-source isCached="true">
   ...
</document-source>
```

Important: If `isCached` has a value of "`false`" (the default), all other cache settings for the XQS function are ignored.

---

2. Use the `<cache-properties>` subelement of `<document-source>`, `<xqsview-source>`, or `<wsdl-source>`. The `time-to-live` attribute takes an integer value to determine how long, in seconds, cache entries are maintained before expiring. For example:

```
   <cache-properties time-to-live="600">
      ...
   </cache-properties>
```

3. Use the `<in-memory>` subelement of `<cache-properties>` to specify which cache mode to use, through the `useSpool` and `useDiskCache` attributes. For plain memory-based caching, set `useSpool` and `useDiskCache` both to "`false`". For memory-based caching with the capability of spooling cached data to disk as necessary in case of any memory shortage, set `useSpool` to "`true`" and `useDiskCache` to "`false`". For disk-based caching, set `useDiskCache` to "`true`" and `useSpool` to "`false`". (See "XQS Caching Strategies", immediately

below, for related considerations.) The following example uses memory-based caching, but with spooling if necessary:

```
<in-memory useSpool="true" useDiskCache="false"/>
```

Here is the complete cache configuration example:

```
<document-source isCached="true">
   ...
   <cache-properties time-to-live="600">
      <in-memory useSpool="true" useDiskCache="false"/>
   </cache-properties>
   ...
</document-source>
```

XQS Caching Strategies

Note the following considerations for XQS caching:

- In situations where data is expensive in any way (in time, in terms of any other possible access costs, or in terms of application overhead), you should probably cache your sources. You should also consider the possible impact of caching on the OC4J instance system memory, but memory effects can be alleviated through disk-based caching, or memory-based caching with spooling. (These are discussed immediately below.)
- In order to cache query results, define a query as an XQS view. A view, being a type of XQS source, can be cached (meaning its results are cached).
- Expiration, or time-to-live, of cached objects is at your discretion. The nature of the data, and how old it can be without losing its usefulness, determines how long it should be cached before expiring.
- In defining your `<in-memory>` settings, consider whether to use plain memory-based caching (`useSpool="false"` and `useDiskCache="false"`); memory-based caching with spooling to disk, in case that is necessary due to memory shortage (`useSpool="true"` and `useDiskCache="false"`); or disk-based caching (`useSpool="false"` and `useDiskCache="true"`). See "<in-memory>" on page 8-91 for additional information. Note the following:
  - Plain memory-based caching has the fastest performance but the greatest risk of loss of cached data, due either to memory shortage or server failure.
  - Memory-based caching with spooling is faster than disk-based caching (unless memory issues result in cached data always being spooled), but risks the loss of cached data in case of server failure.
  - Disk-based caching has the slowest performance, but the greatest safety for cached data. There are no memory concerns, and cached data is recovered in the event of server failure.

  Balance the criticality of cached data versus the importance of execution speed.

Caching and Nondeterministic Results

Be aware that the issue of XQS functions and node identity, described in "Requirements, Limitations, and Special Notes for the Current Release" on page 8-6, may be relevant in some unusual or atypical situations, resulting in differing query results depending on whether caching is used. Generally this is not an issue, however, as the great majority of queries depend on equality between values, not between nodes.

Configuring XQS Document or View Sources for Large Data

There are situations where data sets are so large that memory limitations are encountered regardless of system capacity or the use of typical memory conservation. A solution for this is to use a "working unit" approach behind the scenes, where a constant amount of memory is used regardless of the total amount of data being retrieved. XQS is cooperating with dependent technologies in the Oracle XDK and middle-tier XQuery to provide such a scalable solution whenever you are running XQS.

There are also situations, however, where data sets are large enough that even the "working unit" approach is insufficient. In these circumstances, for document sources and XQS view sources, you can provide a hint to XQS that the data source may take up sizable memory resources, and that XQS should use memory-saving internal optimizations. Use the `largeData` flag—an attribute of the `<document-source>` and `<xqsview-source>` elements—as in the following example:

```
<document-source largeData="true">
   ...
</document-source>
```

If you have trouble loading a large document source or XQS view source into memory, enabling this flag may allow you to proceed.

Important:

- The `largeData` flag does not apply to WSDL sources.
- With `largeData="true"`, you may experience degradation in speed. Use this setting only when necessary. For this reason, the default value is "`false`".

XQS XPath Optimization for WSDL Sources with SQL Binding

For WSDL sources with SQL binding, XQS has special functionality to analyze each XQuery expression and try to choose the most efficient way to execute it. In particular, if you have expressions that apply an XPath path to the results of the function call, it is best to apply the path directly at the point of the function call.

For example, assume `mySQLData()` is for a WSDL source with SQL binding, executing a relational query and returning results as XML. First consider the following XQuery expression:

```
...
for $i in mySQLData(param) return
<c>
$i/a/b/c
</c>
```

By comparison, now consider this XQuery expression:

```
...
for $i in mySQLData(param)/a/b/c return
<c>
$i
</c>
```

The second XQuery expression is equivalent to the first, but may execute faster and bring in less data from the database to the OC4J server space, if the XPath expression selects only a small subtree of the XML result returned by `mySQLData()`. This is because in the second case, the XPath expression "`/a/b/c`" would be executed by the XQuery engine in the database, and therefore only the "`c`" sub-tree would be brought into the XQS application running on OC4J. XQS applies this optimization to a certain subset of XPath expressions, such that the correctness of the result is assured.

Note: This optimization is supported only with Oracle Database and a version of 10.2 or higher.

Using XQS Error Handling Modes and APIs

As noted in "Introduction to XQS Error Handling" on page 8-13, you can configure XQS functions to continue even if they encounter errors (such as data source unavailability or data conversion problems). If you do configure XQS functions to continue, you can then retrieve the errors, which are returned as an iterator over `oracle.xds.client.XQSError` objects, and obtain information about the errors.

XQS applies special handling only to errors inside an XQS function. A regular XQuery error, such as a syntax error or type mismatch, terminates query execution regardless of the XQS error mode.

You configure the XQS error mode for each XQS function individually. The error mode for one function does not have any effect on the behavior of other XQS functions in the same query.

The following sections discuss how to configure XQS functions to continue and return error information, and what methods are available to retrieve that information:

- Configuring XQS Function Error Handling
- Retrieving XQS Error Objects
- Obtaining Information from XQS Error Objects
- Example: Error Retrieval and Processing

Configuring XQS Function Error Handling

For each data source you configure in `xqs-config.xml` (or `global-xqs-config.xml`), you can specify XQS error-handling through the `onError` attribute of the source element—`<document-source>`, `<xqsview-source>`, or `<wsdl-source>`. The following settings are supported.

- `dynamicError` (default): With this setting, XQS raises an XQuery dynamic error and the query is terminated if any problem is encountered during evaluation of the XQS function. In this circumstance, no error objects are returned; you cannot retrieve any information beyond the exception message itself.
- `emptySequence`: With this setting, in the event of an error generated by the XQS function, the function returns an empty XML sequence as its result, and the query execution can continue. You can retrieve error objects and obtain information from them as discussed in the sections that follow.
- `errorMessage`: With this setting, in the event of an error generated by the XQS function, the function returns a one-item XML sequence for the function result. The item in the sequence represents an error message—either a preconfigured message or whatever message the data source returns, depending on your configuration. You can also retrieve error objects and obtain information from them as discussed in the sections that follow.

So there are possibly two steps in configuring your XQS error-handling for a particular data source:

1. Specify the desired setting of the `onError` attribute in the `<document-source>`, `<xqsview-source>`, or `<wsdl-source>` element.
2. If you are using `errorMessage` mode, optionally use the `<error-message>` subelement of `<document-source>`, `<xqsview-source>`, or `<wsdl-source>`. This will preconfigure an error message for XQS to place in the one-item XML sequence that is returned for the query result.

Here is an example for a source that uses `emptySequence` mode:

```
<document-source onError="emptySequence">
   ...
</document-source>
```

And here is an example that uses `errorMessage` mode with a preconfigured message:

```
<wsdl-source onError="errorMessage">
   ...
   <error-message>No information available</error-message>
   ...
</wsdl-source>
```

In `errorMessage` mode, if you do not specify a preconfigured message (in other words, if you do not use an `<error-message>` element), then XQS will take whatever message is returned by the data source, and put that message in the one-item XML sequence that is returned as the function result. Occasionally it will be impossible to obtain any meaningful error message from a data source with poor error-reporting facilities. In that case, if you did not provide a preconfigured message, the error message returned will be empty.

The single item in the sequence returned can be of one of two types: `xs:string` or `xs:node`.

- If the item is a string, the string is simply the error message, either preconfigured or taken from the actual error that occurred.
- If the item is an XML node, it will be called `xqserror` and its text value will be the error message. In addition, the error node will have three attributes: `functionName`, `type` and `code`. The XQS error node looks like this:

```
<xqserror functionName="XQS function name" type="XQSERR_XXX" code="XQS 0NNN" >
   error message
</xqserror>
```

XQS chooses a more appropriate type for the error item based on the type information it has about the function:

- If the XQS function is a Document function, the error item will be of type `xs:node`. This is because a Document function is always expected to return an XML node.
- If the XQS function is configured with `<output-element>`, and the type or element name is not a primitive XML Schema type, XQS assumes that the function is an XML element type and creates the error item as an XML node.
- If `<output-element>` specifies a primitive XML Schema type (even if it is not `xs:string`), or if `<output-element>` is not provided, then XQS creates the error item as a string.

Notes:

- If you configure an XQS function with the `errorMessage` error mode and you do not provide `<output-element>`, or you provide `<output-type>` and it is not a user-defined complex type or `xs:string`, do not specify a return type for the function in the XQuery prolog of the queries that use this function. Declaring a return type for external XQuery functions is optional. If you specify the return type under the preceding circumstances, XQuery enforces this return type at runtime. If an error occurs during execution of the XQS function, XQS will return the error message as a string, and this will conflict with the return type declared in the query prolog. If you do not declare the return type, the type match will not be enforced.
- For a situation in which an XQS view uses other XQS functions and sources, error-handling configuration of the view and of the underlying functions are relevant. Configuration of any underlying function determines how the function behaves when an error is encountered—whether it terminates with a dynamic error, or continues and returns a predetermined value (either an empty sequence or an error message). Configuration of the view itself determines whether to stop or to continue the query associated with the view when an error is encountered, including but not limited to errors raised by any underlying functions.
- For an <xqsview-source> element that defines <output-element>, check if any XQS function in the underlying query, or in a query nested via use of another <xqsview-source> function, defines an "errorMessage" error-handling option. Every such nested <xqsview-source> function with the onError="errorMessage" option must provide an appropriate <output-element> definition. If this requirement is not met and an error occurs in the nested XQS view function, XQS will create a return value of the type xs:string for that function. This would violate the declared type of the outermost XQS view function and create additional XQuery errors due to a type mismatch.

  Other error-handling options are not subject to this requirement.
- When an error occurs in the default `dynamicError` mode, XQS clients propagate the exception as follows: an EJB will return an EJB exception; an `XQSFacade` instance will throw an XQS exception wrapping the XQuery exception; a JSP tag will throw a JSP tag exception which can be handled through the standard JSP `errorPage` attribute.
- In addition to creating error objects (discussed in "Retrieving XQS Error Objects" below), XQS reports errors to OC4J log files according to your overall OC4J logging configuration. You can search for XQS error, warning, or information messages in the relevant log file (such as `*ORACLE_HOME*/j2ee/home/log/oc4j/log.xml`). See ***Oracle Containers for J2EE Configuration and Administration Guide*** for general information about OC4J logging.

Retrieving XQS Error Objects

Assuming an XQS function is configured to use the `emptySequence` or `errorMessage` error-handling mode, you can use features of the XQS client APIs to retrieve an iterator of error objects, from which you can obtain information about any problems that occurred during the query. Here is a quick summary:

- If you use the `XQSFacade` class or either of the XQS EJB APIs (stateful or stateless), use the following method to retrieve an iterator containing error objects:

  ```
  java.util.ListIterator getErrors()
  ```

- If you use the XQS JSP tag library (stateful or stateless), specify the desired variable name as the value of the `errors` attribute of the `execute` or `executeCursor` tag. The JSP tag implementation creates the `ListIterator` variable and populates it with an iterator containing error objects.

In either case, you will have an iterator containing `oracle.xds.client.XQSError` objects. Here is an example:

```
...
XQSFacade bean = new XQSFacade();
...
bean.executeView(viewname, namespace, queryparams);
...
ListIterator errorIt = bean.getErrors();
XQSError error = null;
while(errorIt.hasNext()) {
   error = (XQSError)errorIt.next();
   (Process errors)
}
...
bean.close();
...
```

And here is a JSP example:

```
XQSError error=null;
...
<xq:executeCursor ... errors="errorIt" ... >
   ...
</xq:executeCursor>
...
<xq:next ... />
while(errorIt.hasNext()) {
   error = (XQSError)errorIt.next();
   (Process errors)
}
```

See the next sections, "Obtaining Information from XQS Error Objects" and "Example: Error Retrieval and Processing", for information and examples for obtaining error information.

Obtaining Information from XQS Error Objects

The preceding section, "Retrieving XQS Error Objects", describes how to obtain a `java.util.ListIterator` instance consisting of `oracle.xds.client.XQSError` objects.

You can process the iterator instance to obtain the individual `XQSError` objects. The `ListIterator` type includes the following methods:

- `Object next()`: This returns the next element in the iterator. You can cast each element to the `XQSError` type.
- `boolean hasNext()`: This returns `true` if there are elements remaining.

You can then use the individual `XQSError` instances to obtain information about each error that occurred during a query.

Notes:

- XQS returns `XQSError` objects only if you configure one or more XQS functions used in your query with a nondefault `emptySequence` or errorMessage mode. If no errors occurred, `ListIterator` will be empty.
- The `XQSError` Java object should not be confused with the error item of type `xs:node`, which XQS sometimes constructs as a return value for an XQS function in the case of an error, as described in "Configuring XQS Function Error Handling". on page 8-69. `XQSError` is a Java object, as opposed to an XML node. `XQSError` objects are returned to the *client* of XQS, whereas the XML error node is returned to the XQuery execution. Finally, `XQSError` objects are returned even if the error mode is `emptySequence`, whereas an XML error node is constructed only when the error mode is `errorMessage`.

This section summarizes the `XQSError` methods for obtaining error information, and also summarizes XQS error types. For complete information, see "XQSError Class Reference" on page 8-85.

With each `XQSError` instance, you can do the following. Assume an `XQSError` instance, `xqserr`, for these examples:

1. Obtain the qualified name of the XQS function that returned the error:

   ```
   String funcname = xqserr.getFunctionQName();
   ```

2. Obtain the error type (see immediately below for types):

   ```
   int errortype = xqserr.getErrorType();
   ```

3. Obtain string representation of the error type:

   ```
   String typeStr= XQSError.typeNames(xqserr.getErrorType()
   ```
4. Obtain the error message:

   ```
   String errormessage = xqserr.getErrorMessage();
   ```

5. For data sources that provide error codes separately from error messages (such as `ORA-xxxxx` codes from an Oracle database), obtain the error code:

   ```
   String errorcode = xqserr.getErrorCode();
   ```

6. Optionally use utility methods to combine the function name, error type, error message, and error code into a single string, an XML string (a string that includes appropriate XML markup), or an XML node:

   ```
   String errorstring = xqserr.toString();
   ...
   String errorxmlstring = xqserr.toXMLString();
   ...
   ```

```
org.w3c.dom.node errornode = xqserr.toXMLNode();
```

Here is a summary of the integer constants for error types:

- `XQSERR_MISSING_SRC`: Data source not found.
- `XQSERR_SRC_ACCESS`: Data source found but returns an access error (such as invalid login).
- `XQSERR_SRC_ERROR`: Data source found and accessed but returns some other error.
- `XQSERR_PARAM`: Problems encountered in passing bind parameters to the source, or receiving output parameters from the source.
- `XQSERR_SRC_CONFIG`: Problems encountered in processing XQS configuration.
- XQSERR_INTERNAL:XQS encountered unexpected internal error.

See "XQSError Class Reference" on page 8-85 for additional information about these constants.

Example: Error Retrieval and Processing

This section shows a simple error-processing example with corresponding configuration.

Configuration To use `XQSError` objects in an XQS application (as discussed earlier, in "Configuring XQS Function Error Handling" on page 8-69), the data source must be configured for error mode `emptySequence` or `errorMessage`. The following example uses `errorMessage` mode:

```
<xqsview-source ... onError="errorMessage">
   <function-name prefix="xqs">BasicFileWS</function-name>
<output-element namespace="myNS">myElement</output-element>
   ...
</xqsview-source>
```

Because no `<error-message>` element is used, the one-item XML result sequence that is returned will forward whatever message comes from the data source, rather than containing a predefined message. If the XQS view function, in fact, encounters an error, XQS will construct an `<xqserror>` XML node and return it as the function result. The error message will be the text value of the `<xqserror>` node. XQS will choose the XML node representation because the XQS view configuration specifies a user-defined element in its `<output-element>` configuration.

Java Code Following is a sample code fragment that executes a view and then does the following error processing:

- Calls the `getErrors()` method of the `XQSFacade` class to retrieve a `java.util.ListIterator` instance of `XQSError` objects.
- Uses the `ListIterator` class `hasNext()` method to determine how many error objects to iterate through.
- For each error, prints the qualified name of the XQS function that produced the error, then prints the error type, error message, and error code.

```
...
XQSFacade bean = new XQSFacade();
String xqueryView = "BasicFileWS";
QueryParameter param = new QueryParameter("custName");
```

```
param.setString("John Ford");
QueryParameter[] params = new QueryParameter[1];
params[0] = param;
bean.executeView(xqueryView, null, params);
while(bean.getNextItem() !=null) {
}
ListIterator errors = bean.getErrors();
while(errors.hasNext()) {
   XQSError error = (XQSError)errors.next();
   System.out.println("The function which gives error is: " +
                       error.getFunctionQName());
   System.out.println("The error type is:"  + error.getErrorType() +
", which means " +XQSError.typeNames(error.getErrorType()));

   System.out.println("The error message is:" + error.getErrorMessage());
   System.out.println("The error code is:" + error.getErrorCode());
}
...
```

XQS Client APIs Reference

This section provides reference documentation for XQS public classes and user interfaces:

- XQS QueryParameter Class Reference
- XQSFacade Class Reference
- XQS EJB Client API Reference
- XQS JSP Tag Library Reference
- XQSError Class Reference

XQS QueryParameter Class Reference

This section documents the constructor and methods of the `oracle.xds.client.QueryParameter` class. A `QueryParameter` array is used for binding external variables for a query when using the XQS general-purpose Java API or the EJB API.

"Example 3: XQSFacade API with an XQS View" on page 8-48 includes an example of using `QueryParameter` for an input parameter.

QueryParameter Constructors

The `QueryParameter` class supplies the following constructors:

- `QueryParameter(String namespace, String localName)`
- `QueryParameter(String localName)`

  When you construct a `QueryParameter` instance, you specify the name of the external XQuery variable to be bound, and this must match the name from the corresponding external variable declaration in the XQuery prolog. XQuery variables use XML names, so the `QueryParameter` constructor can take two string values—one for the namespace and one for the local name. If you do not specify a namespace, then `null` is assigned to the namespace in the qualified name.

QueryParameter Methods

The `QueryParameter` class supplies the following methods, each to set a value of an external XQuery variable. These methods reflect XML types supported by XQS, and their Java equivalents; also see "Supported Types for Query Parameters" on page 8-40.

- `void setBase64Binary(String value)`

  Use this to bind a base 64 binary value, represented by a Java string (for example, "vYrfOJ39673//-BDiIIGHSPM=+"), and corresponding to the XML `base64Binary` type.

- `void setBoolean(boolean value)`

  Use this to bind a `boolean` value, corresponding to the XML `boolean` type.

-
  ```
  void setDateTime(java.util.GregorianCalendar value
                   boolean isTimeZoneSet)
  ```

  Use this to bind a `GregorianCalendar` value representing a particular point in time, corresponding to the XML `dateTime` type. Set `isTimeZone` to `true` if the calendar object `TimeZone` property has been set. (See `http://java.sun.com/j2se/1.4.2/docs/api/` for information about using `GregorianCalendar` objects.)

- `void setDecimal(java.math.BigDecimal value)`

  Use this to bind a `BigDecimal` value, corresponding to the XML `decimal` type.

- `void setDouble(double value)`

  Use this to bind a `double` value, corresponding to the XML `double` type.

- `void setDuration(String value)`

  Use this to bind a `String` value representing a duration, corresponding to the XML `duration` type. See `http://www.w3.org/TR/xmlschema-2/#duration` for information about lexical (string) representations of durations.

- `void setFloat(float value)`

  Use this to bind a floating point value, corresponding to the XML `float` type.

- `void setHexBinary(String value)`

  Use this to bind a hexadecimal binary value, represented by a Java string (for example, "0FB7"), and corresponding to the XML `hexBinary` type.

- `void setInt(int value)`

  Use this to bind an integer value, corresponding to the XML `int` type.

- `void setInteger(int value)`
- `void setInteger(long value)`
- `void setInteger(java.math.BigInteger value)`

  Use one of the `setInteger()` methods to bind an integer value, corresponding to the XML `integer` type. Choose the appropriate signature based on the value representation or magnitude.

- `void setLong(long value)`

  Use this to bind a long integer value, corresponding to the XML `long` type.

- `void setString(String value)`

Use this to bind a Java string value, corresponding to the XML `string` type.

- `void setAnyURI(java.net.URI value)`

  Use this to bind a `URI` value, corresponding to the XML `anyURI` type.

- `void setNode(org.w3c.dom.Node value)`

  Use this to bind an XML node, corresponding to the XML `anyType` type or to any user-defined XML type.

> **Note:** The `QueryParameter` class has no getter methods corresponding to these setter methods.

XQSFacade Class Reference

This section documents the constructor and methods of the `oracle.xds.client.XQSFacade` class, a general-purpose Java client API that XQS supplies. See "Using the Java Class Client API" on page 8-42 for usage instructions and an example.

XQSFacade Constructor

The `XQSFacade` constructor has no arguments:

- `XQSFacade()`

XQSFacade Methods

The `XQSFacade` class supplies the following methods:

- `void execute(String xquery, QueryParameter[] params)`

  Use this method to execute an ad-hoc query. Pass in a string containing the query, and a `QueryParameter` array for any bind parameters. (See "XQS QueryParameter Class Reference" on page 8-75.) The array can be null if there are no parameters to pass.

- `void executeView(String viewName, String namespace, QueryParameter[] params)`

  Use this method to execute an XQS view directly. Pass in a view name by specifying the local name and namespace (as strings), and pass in a `QueryParameter` array for any bind parameters (or null). The view name and namespace together define a qualified name and must match the corresponding function name and namespace in your configuration, according to the `<function-name>` subelement of `<xqsview-source>`, and its `namespace` or `prefix` attribute.

- `oracle.xml.xqxp.datamodel.XMLItem getNextItem()`

  After using `execute()` or `executeView()`, use `getNextItem()` to obtain the next item in the result sequence, as an Oracle `XMLItem` instance. The first `getNextItem()` call returns the first item in the sequence. If you call this method after you have already retrieved the last item, it returns `null`.

> **Important:** A call to the `getNextItem()` method triggers the evaluation of the item by XQuery. Any resources required when executing the query (such as a database connection or file handle) are not freed until after the last `getNextItem()` call.

- `java.util.ListIterator getErrors()`

  If you configure any of the XQS functions used in your query to continue even if errors are encountered (as discussed in "Configuring XQS Function Error Handling" on page 8-69), you can use `getErrors()` to retrieve information about any errors that may have been encountered during execution of XQS functions. This method returns an iterator over a collection of `XQSError` objects, from which you can retrieve the specifics of each error. (See "XQSError Class Reference" on page 8-85.) If there were no errors, the iterator will point to an empty collection.

- `void close()`

  Use this to free all resources associated with the query. It is good practice to call `close()` after retrieving all query results.

XQS EJB Client API Reference

This section documents EJB client methods supported by XQS, through the `XQSClientRemote` and `XQSClientHome` interfaces for remote EJBs, and the `XQSClientLocal` and `XQSClientLocalHome` interfaces for local EJBs. There are versions of these interfaces for either stateful or stateless session beans.

See "Using the EJB Client API" on page 8-52 for usage instructions and examples.

Stateful EJB Client Methods

The following Oracle-specific methods are available through EJB interfaces in package `oracle.xds.client.ejb.stateful`, for stateful session beans, and have the same functionality as the methods of the same name discussed in "XQSFacade Class Reference" on page 8-77:

-
  ```
  void execute(String xquery, QueryParameter[] params)
  throws EJBException
  ```

-
  ```
  void executeView(String viewName, String namespace,
  QueryParameter[] params)
  throws EJBException
  ```

-
  ```
  oracle.xml.xqxp.datamodel.XMLItem getNextItem()
  throws EJBException
  ```

- `java.util.ListIterator getErrors()`

- `void close()`

Stateless EJB Client Methods

The following Oracle-specific methods are available through EJB interfaces in package `oracle.xds.client.ejb.stateless`, for stateless session beans:

-
  ```
  java.util.Vector execute(String xquery, QueryParameter[]
  params)
  throws EJBException
  ```

  Use this method to execute an ad-hoc query. Pass in a string containing the query, and a `QueryParameter` array for any bind parameters. (See "XQS QueryParameter Class Reference" on page 8-75.) The array can be null if there are no parameters to pass. Results are fully materialized, returned in a `Vector` instance.

- `java.util.Vector executeView(String viewName, String namespace, QueryParameter[] params) throws EJBException`

  Use this method to execute an XQS view. Pass in a view name by specifying the local name and namespace (as strings), and pass in a `QueryParameter` array for any bind parameters (or null). The view name and namespace together define a qualified name and must match the corresponding function name and namespace in your configuration, according to the `<function-name>` subelement of `<xqsview-source>`, and its `namespace` or `prefix` attribute. Results are fully materialized, returned in a `Vector` instance.

- `java.util.ListIterator getErrors()`

  This method has the same functionality as the method of the same name discussed in "XQSFacade Class Reference" on page 8-77.

XQS JSP Tag Library Reference

XQS supplies custom JSP tags for either stateful or stateless access to XQuery.

See "Using the JSP Tag Library" on page 8-54 for usage instructions and examples.

See the *Oracle Containers for J2EE Support for JavaServer Pages Developer's Guide* for general information about using JSP tag libraries.

Notes: The prefix "xq:" is used in the tag syntax here. This is by convention but is not required. You can specify any desired prefix in your `taglib` directive.

For tag syntax, note the following:

- *Italic* indicates where you specify a value or string.
- Optional attributes are enclosed in square brackets: `[...]`
- Default values of optional attributes are indicated in **bold**.
- Choices in attribute values are separated by vertical bars: `|`
- Except where noted, you can use JSP runtime expressions to set tag attribute values: "`<%=` *jspExpression* `%>`"

JSP Tags for Stateful Access

Use these tags when you want a stateful access to XQuery.

XQS executeCursor Tag Use this tag to prepare and execute the query. It has the following syntax. (The `param` subtag, used for any bind parameters, is described next.)

```
<xq:executeCursor
  [ xqueryString = "query" ]
  [ xqsViewName = "viewname" ]
  [ namespace = "namespace" ]
    cursorId = "cursorname"
  [ maxItems = "maxnumber" ]
  [ toWriter = "true" | "false" ]
    toXMLDoc = "docname"
    resultItems = "arrayname"
    errors = "errorvarname" >
```

```
   <xq:param ... />
   <xq:param ... />
   ...

</xq:executeCursor>
```

Important:

- For query input, you must use either the `xqueryString` attribute or the `xqsViewName` and `namespace` attributes.
- You must provide variable names for the output attributes `toXMLDoc`, `resultItems`, and `errors`.

Use attributes of the `executeCursor` tag as follows:

- `xqueryString`: When using an ad-hoc query, use this attribute to specify the complete XQuery syntax of the query. It is usually easiest to define the query in a string variable and specify the name of the variable in a JSP expression:

  ```
  String myquerystring = "...";
  ...
  <xq:executeCursor ... xqueryString = "<%=myquerystring%>" ... >
     ...
  </xq:executeCursor>
  ```

- `xqsViewName`: When using an XQS view, use this attribute to specify the view. Specifically, this is the name of the corresponding XQS function name in your configuration.
- `namespace`: If you use `xqsViewName` to specify a view, you must also use `namespace` to specify the namespace of the XQS function for the view, according to your XQS configuration.
- `cursorId` (required): Use this tag to specify the name of a variable for the cursor, for later use. (The variable will be declared by the JSP container.) You will reference the specified variable name when you use a `next` tag to retrieve results from the cursor, and when you use the `close` tag to close the cursor. Here is an example:

  ```
  <xq:executeCursor ... cursorId="mycursor" ... >
     ...
  </xq:executeCursor>
  ...
  <xq:next cursorId="mycursor" />
  ...
  <xq:close cursorId="mycursor" />
  ```

- `maxItems`: Use this if you want to specify a maximum number of items to receive from the query, such as `maxItems="10000"`. Items beyond that point will be discarded, and the `next` tag will stop returning values after `maxItem` items have been received.
- `toWriter`: Set this to "`true`" to output query results to the JSP output stream. The writing occurs every time the `next` tag is executed.
- `toXMLDoc` (required): Provide the name for a variable to be used by a related `next` tag to output query results to an XML DOM document. The document can be used after the `executeCursor` tag. A variable of type

org.w3c.dom.Document will be declared by the JSP container. Here is an example:

```
<xq:executeCursor cursorId="mycursor" toXMLDoc="mydoc" ... >
   ...
</xq:executeCursor>

<xq:next cursorId="mycursor" />

Element root = mydoc.getDocumentElement();
...
```

After using the `executeCursor` tag, each time you execute the `next` tag you will receive a batch of result nodes collected in the document `mydoc`.

Important:

- The `toXMLDoc` document collects only results that are XML nodes. Results that are primitive types are ignored for the XML document, but collected through the `resultItems` array.
- The `toXMLDoc` document holds only the batch of items from one execution of the `next` tag. Each subsequent execution of `next` overwrites the items with a new set of items.
- You cannot use a runtime expression for the `toXMLDoc` attribute.

Note: The specified name will also become the name of the root element of the document.

- `resultItems` (required): Provide the name for a variable to be used by a related `next` tag to output query results to an array list. A variable of type `java.util.ArrayList` will be declared by the JSP container. The `ArrayList` instance can be used after the `executeCursor` tag. Here is an example:

```
<xq:executeCursor ... resultItems="myobjects" ... >
   ...
</xq:executeCursor>

Node node = (Node)myobjects.get(0);
...
```

After using the `executeCursor` tag, each time you execute the `next` tag you can access a result item from the `myobjects` variable.

Important:

- The `resultItems` variable holds only the batch of items from one execution of the `next` tag. Each subsequent execution of `next` overwrites the items with a new set of items.
- You cannot use a runtime expression for the `resultItems` attribute.

- `errors` (required): Provide the name of a variable to be used in retrieving errors that result from execution of the XQS query. A variable of type `java.util.ListIterator` will be declared by the JSP container for an iterator over a collection of `XQSError` objects (one object per error). The iterator can be used after the `executeCursor` tag. Here is an example:

```
<xq:executeCursor cursorId="mycursor" errors="myerroriter" ... >
   ...
</xq:executeCursor>
<xq:next cursorId="mycursor" />

XQSError error = (XQSError)myerroriterator.next();
...
```

See "XQSError Class Reference" on page 8-85 for how to use `XQSError` objects.

Important: You cannot use a runtime expression for the `errors` attribute.

XQS param Tag Use this subtag of `executeCursor` and `execute` to specify any external bind parameters for the query (one `param` tag per parameter). It has the following syntax:

```
<xq:param
    localName = "localvarname"
  [ namespace = "namespace" ]
    value = "bindvalue"
    type = "bindparamtype"
/>
```

Use attributes of the `param` tag as follows:

- `localName` (required): Use this to specify the local part of the external variable name.
- `namespace`: Use this to specify the namespace part of the external variable name. Because it is permissible to use a local variable name without a namespace, this attribute is optional.
- `value` (required): Use this for the value to be bound, which will presumably be a runtime value using a JSP runtime expression:

  ```
  <xq:param ... value = "<%=jspExpression%>"... />
  ```

  Note: The type of the bind value must be a match for the `type` attribute setting, as noted in Table 8–3 below.

- `type` (required): Use this to specify the data type of the bind parameter, indicating an XML type for which the XQS `QueryParameter` class supports a corresponding Java type. Supported settings, and the type or types for which they are appropriate, are shown in Table 8–3. (Also see "Supported Types for Query Parameters" on page 8-40.)

Table 8–3 Correspondence of Java Types to type Attribute Settings

type Attribute Setting	Matching Java Type(s)
boolean	java.lang.Boolean
string	java.lang.String

Table 8–3 (Cont.) Correspondence of Java Types to type Attribute Settings

type Attribute Setting	Matching Java Type(s)
int	java.lang.Integer
integer	java.lang.Integer, java.lang.Long, java.math.BigInteger
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double
decimal	java.math.BigDecimal
base64Binary	java.lang.String (Representation of the value.)
hexBinary	java.lang.String (Representation of the value.)
anyURI	java.net.URI
dateTime	java.util.GregorianCalendar (With the assumption that the time zone for the calendar value has been initialized appropriately.)
duration	java.lang.String (Lexical representation of the duration, discussed at `http://www.w3.org/TR/xmlschema-2/#duration`.)
node	java.lang.String (Text representation of the node, to be parsed by an XML parser.)

XQS next Tag Use this tag to process results, item by item or batch by batch, from the `executeCursor` tag, referencing the cursor ID name that you specified in that tag. The `next` tag has the following syntax:

```
<xq:next
    cursorId = "cursorname"
  [ batchSize= "numitems" ]
  [ itemsFetched = "intvarname" ]
/>
```

Use attributes of the `next` tag as follows:

- `cursorId` (required): Use this attribute to reference the cursor name for the query, as specified in the `executeCursor` tag. The cursor must still be open (there has not yet been a `close` tag specifying the same cursor name).
- `batchSize`: Use this attribute if you want XQS to use batch mode for the query. Specify an integer value for how many items you want fetched from the cursor each time the `next` tag is executed. The default value is "`1`" (no batching).
- `itemsFetched`: Specify the name of a `java.lang.Integer` variable for a value that indicates how many items were fetched from the cursor in the current `next` tag execution. The variable will be declared by the JSP container.

Here is an example:

```
...
<xq:executeCursor cursorId="mycursor"
                  resultItems="myresults"
                  toXMLDoc="mydoc"
                  errors="myerrors" ... >
   <xq:param ... />
   ...
</xq:executeCursor>
<%
  int items;
  do {
  // Populate output vehicles (myresults array and mydoc document)
%>
   <xq:next
                cursorId="mycursor"
                itemsFetched="fetched" />
<%
  items = fetched.intValue();
  } while(items>0)
  // ... Here is where you would retrieve results from the output vehicles ...
%>
...
```

XQS close Tag For a stateful JSP (using the `executeCursor` tag), use the `close` tag to free resources associated with the query. It has the following syntax:

```
<xq:close
    cursorId = "cursorname"
/>
```

Use attributes of the `close` tag as follows:

- `cursorId` (required): Use this attribute to reference the cursor name, as specified in the `executeCursor` tag, corresponding to the query you want to close.

JSP Tags for Stateless Access

Use the `execute` tag when you want a stateless access to XQuery.

XQS execute Tag Use this tag to prepare and execute the query. It has the following syntax:

```
<xq:execute
  [ xqueryString = "query" ]
  [ xqsViewName = "viewname" ]
  [ namespace = "namespace" ]
  [ maxItems = "maxnumber" ]
  [ toWriter = "true" | "false" ]
    toXMLDoc = "docname"
    resultItems = "arrayname"
    errors = "errorvarname" >



   <xq:param ... />
   <xq:param ... />
   ...

</xq:execute>
```

Important:

- For query input, you must use either the `xqueryString` attribute, or the `xqsViewName` and `namespace` attributes.
- You must provide variable names for output vehicles using the required attributes `toXMLDoc`, `resultItems`, and `errors`.

Aside from `cursorId`, this tag uses the same attributes as the `executeCursor` tag (for stateful access). It also uses the XQS `param` subtag, as does `executeCursor`, for bind parameters. See "XQS executeCursor Tag" on page 8-79 and "XQS param Tag" on page 8-82.

XQSError Class Reference

This section provides reference documentation for the class `oracle.xds.client.XQSError`.

If you configure any of the XQS functions used in your query to continue even if errors are encountered, as discussed in "Configuring XQS Function Error Handling" on page 8-69, you can retrieve information about any errors that may have occurred during execution of the XQS function for the query. Errors are returned in an iterator of `XQSError` objects.

The `XQSError` class provides a number of methods to return its key information—the name of the XQS function that encountered the error, the type of error (an XQS-specific designation), the error message, and the error code (if applicable)—in various formats. See "Obtaining Information from XQS Error Objects" on page 8-72 for usage instructions and an example.

- `String getFunctionQName()`

  Use this to get the fully qualified name of the XQS function that encountered the error, as follows:

  `{namespace_URI}/local_name`

- `int getErrorType()`

  Use this to get an integer constant that indicates the XQS error type designation:

  - `XQSERR_MISSING_SRC` if the data source cannot be found—such as if an HTTP 404 error is returned, the file is not found for a document source, the source is down, or the desired table in a database source does not exist.
  - `XQSERR_SRC_ACCESS` if the data source is found but returns an access error. This would include, for example, an `ORA-xxxxx` error from an Oracle database, a Web site login error, or a parser error when reading a document source.
  - `XQSERR_SRC_ERROR` if the data source is found but returns some kind of error besides an access error, such as a business method error, a Web service fault message, or a SQL error for a database source.
  - `XQSERR_PARAM` if XQS encountered problems in passing bind parameters to the data source or receiving output parameters from the data source. For example, this is returned if XQS could not convert an input string to the corresponding data type required according to the WSDL document, or if XQS could not transform a non-XML file into an XML document with the particular transformer it is using.

  - `XQSERR_SRC_CONFIG` if XQS encountered a problem while processing configuration of the XQS function.
  - XQSERR_INTERNAL if XQS encountered an unexpected internal error.

- `String getErrorMessage()`

  Use this to retrieve the error message, either from the data source or from XQS, as applicable. This method never returns `null`. If a data source generates an error without returning a message, XQS returns the generic message "Function generated an error."

- `String getErrorCode()`

  For an error generated by a data source, use this to return the XQS error code string.

- `String toString()`

  Use this to combine the function name, error type, error code, and error message into a single string.

- `String toXMLString()`

  Use this to combine the function name, error type, error code, and error message into a string representation of an XML error element.

- `public static String[] typenames`

  Use this static array to store one of the following string representations of XQS error types in each element of the array:

  - `XQSERR_MISSING_SRC`
  - `XQSERR_SRC_ACCESS`
  - `XQSERR_PARAM`
  - `XQSERR_SRC_ERROR`
  - `XQSERR_SRC_CONFIG`
  - `XQSERR_INTERNAL`

  Given a Java object error of type `XQSError`, the expression `XQSError.typNames[error.getErrorType()]` returns a string appropriate for the error type of the object.

- `org.w3c.dom.Node toXMLNode()`

  Use this to retrieve the function name, error type, error code, and error message in DOM node format.

XQS Configuration File Reference

This section has an alphabetical dictionary of elements of the XQS configuration files, `xqs-config.xml` and `global-xqs-config.xml`. The XML schema for XQS configuration files can be found at `http://www.oracle.com/technology/tech/xml/xqs/xqs-config.xsd`. It is also available with the XLSDemo application in `xds/samples/XQSDemo/XDS/xqs-config.xsd`, This XML schema applies to both the application-specific as well as global configurations Also see "How to Configure Your XQS Functions" on page 8-24 for information about the use of key elements for each category of data source.

At the application level, XQS looks for `xqs-config.xml` in the `xqs-resources.jar` file at the top level of the application EAR file. The global version, `global-xqs-config.xml`, is in the following location:

```
ORACLE_HOME/j2ee/home/config
```

For each external function declared for an XQuery expression, XQS will first look for configuration for the function in the application-specific `xqs-config.xml` file; if the configuration is not found there, XQS will look in `global-xqs-config.xml`, where you can configure data sources available to any application.

Following is a summary of the hierarchy of these files. For each XQS function you configure, you will use a `<wsdl-source>` element, `<xqsview-source>` element, or `<wsdl-source>` element, depending on the kind of data source being accessed.

```
<xqs-config>
   <bind-prefix>
      <use-prefix>
   <xqs-sources>
      <wsdl-source>
         <function-name>
         <cache-properties>
            <in-memory>
         <output-element>
         <error-message>
         <wsdlURL>
         <operation>
         <service>
         <portType>
         <port>
         <input-parameters>
            <part>
               <schema-type>
         <typeMap>
            <mapping>
               <xmlType>
      <xqsview-source>
         <function-name>
         <cache-properties>
            <in-memory>
         <output-element>
         <error-message>
         <repository>
         <queryName>
         <input-parameters>
            <part>
               <schema-type>
               <xquery-sequence>
                  <itemType>
      <document-source>
         <function-name>
         <cache-properties>
            <in-memory>
         <output-element>
         <error-message>
         <documentURL>
         <XMLTranslate>
            <schema-file>
```

<bind-prefix>

Parent element: <xqs-config>

Child elements: <use-prefix>

Required? Optional; zero or one

Use this element, with `<use-prefix>` subelements, if you want to designate prefixes to represent certain XML namespaces in XQS configuration elements. Each `<use-prefix>` subelement designates one prefix.

The `<bind-prefix>` element has no attributes.

<cache-properties>

Parent element: <document-source>, <wsdl-source>, or <xqsview-source>

Child elements: <in-memory>

Required? Optional; zero or one inside each occurrence of a parent element

If you enable XQS caching through the `isCached` attribute of `<document-source>`, `<wsdl-source>`, or `<xqsview-source>`, use the `<cache-properties>` element with its required `<in-memory>` subelement to set caching properties for use by the particular XQS function. See "XQS Caching Strategies" on page 8-67 for related considerations.

The `<cache-properties>` element is ignored if `isCached="false"` (the default).

Table 8–4 <cache-properties> Attributes

Name	Description
time-to-live	Values: Integer (seconds) Default: n/a (required) This specifies how long items are held in the cache before expiring.

<document-source>

Parent element: <xqs-sources>

Child elements: <cache-properties>, <documentURL>, <error-message>, <function-name>, <output-element>, <XMLTranslate>

Required? Optional; zero or more

Use a `<document-source>` element and its subelements (as appropriate) for each XQS function you configure that uses a document source.

You can configure the use of a generic document source where the URL is taken at runtime, or you can use the `<documentURL>` subelement to specify a fixed URL in the configuration. (For an example of taking a URL at runtime, see "Configuring an XQS Function That Accesses a Document Source" on page 8-24.)

For a non-XML document, you can use the `<XMLTranslate>` element to configure translation to XML, if the document conforms to formats supported by the D3L tool.

(See "Preparing to Use a Non-XML Document Source" on page 8-15 for information about D3L.)

Table 8–5 <document-source> Attributes

Name	Description
isCached	Values: Boolean Default: false Set this to "`true`" to cache results from the data source. Also see "Configuring XQS Caching" on page 8-66 and information about the `<cache-properties>` element.
largeData	Values: Boolean Default: false Set this to "`true`" if you want to provide a hint to XQS that it should try to use memory-saving internal optimizations to handle large volumes of data. (Use only when necessary.) Also see "Configuring XQS Document or View Sources for Large Data" on page 8-68.
onError	Values: dynamicError | emptySequence | errorMessage Default: dynamicError This determines which error-handling mode XQS uses. See "Configuring XQS Function Error Handling" on page 8-69.

<documentURL>

Parent element: <document-source>

Child elements: None

Required? Optional; zero or one inside each occurrence of the parent element

Use the value of this element to specify the document to use as the data source, such as in the following example:

```
<documentURL>http://host:port/xqsdemos/Repository/pos-2KB.xml</documentURL>
```

If it is a non-XML document, use the `<XMLTranslate>` subelement of `<document-source>` to specify the translation tool for XQS to use.

The `<documentURL>` element has no attributes.

Notes:

- If you are behind a firewall and the specified URL requires external Internet access, be aware that you must also configure OC4J with appropriate proxy settings.
- As an alternative to predefining a document URL through the `<documentURL>` element, you can omit this element and declare the XQuery external function (corresponding to the XQS function you are configuring) to take the document URL as a runtime argument.
- For your specification of the document URL, be aware that if the document is on the local file system, using `file://` protocol instead of `http://` protocol will give you faster data retrieval.

<error-message>

Parent element: <document-source>, <wsdl-source>, or <xqsview-source>

Child elements: None

Required? Optional; zero or one inside each occurrence of a parent element

For the XQS `errorMessage` error mode—determined by the `onError` attribute of `<document-source>`, `<wsdl-source>`, or `<xqsview-source>`—use the `<error-message>` element if you want to predefine a fixed error message instead of forwarding whatever message comes from the data source. The element value constitutes the message, as in the following example:

```
<error-message>No information available</error-message>
```

Also see "Introduction to XQS Error Handling" on page 8-13.

The `<error-message>` element has no attributes.

<function-name>

Parent element: <document-source>, <wsdl-source>, or <xqsview-source>

Child elements: None

Required? Required inside each occurrence of a parent element; one only

Use this element to declare the desired name of the XQuery function that XQS will implement to access the data source. The element value is the function name, and attribute settings specify the function namespace. (Each XQuery function must belong to some namespace.) Be aware that you must reference the function namespace every time you invoke the function in an XQuery expression, unless you have declared that particular namespace as the default for the XQuery expression.

Use the `namespace` attribute to specify the namespace directly; use the `prefix` attribute as a shortcut if you have previously defined a namespace prefix through a `<bind-prefix>` element. You must use one or the other, but do *not* use both `namespace` and `prefix` at the same time. For information about using the

`<output-element>` element, see "Considerations for Using <output-element >" on on page 8-29.

The function name you specify here is the function name you will declare in the XQuery external function declarations in your code, to access the corresponding data source.

Here is an example:

```
<function-name  namespace="http://xmlns.oracle.com/ias/xqs">
   customerInfo
</function-name>
```

Or, alternatively, if `xqs` has been defined as a prefix for the XQS namespace:

```
<function-name prefix="xqs">customerInfo</function-name>
```

Here is an example of an XQuery function declaration corresponding to the preceding configuration:

```
declare namespace xqs="http://xmlns.oracle.com/ias/xqs";
declare function xqs:customerInfo() external;
```

Table 8–6 <function-name> Attributes

Name	Description
namespace	Values: URI Default: None (required if `prefix` not used) Specifies the namespace directly.
prefix	Values: String Default: None (required if `namespace` not used) Specifies the namespace through a predefined prefix.

<in-memory>

Parent element: <cache-properties>

Child elements: None

Required? Required if the parent element is used; one only

Use the `<cache-properties>` element and its `<in-memory>` subelement to set XQS cache properties. Attributes of the `<in-memory>` element determine the caching mode. For simple memory-based caching, set `useSpool` and `useDiskCache` both to "`false`". For memory-based caching with the capability of spooling cached data to disk as necessary, in case of any memory shortage, set `useSpool` to "`true`" and `useDiskCache` to "`false`". For disk-based caching, set `useDiskCache` to "`true`" and `useSpool` to "`false`". With disk-based caching, cached data will survive a server crash. See "XQS Caching Strategies" on page 8-67 for related considerations.

For example, for memory-based caching that uses spooling if necessary:

```
<cache-properties time-to-live="600">
   <in-memory useSpool="true" useDiskCache="false"/>
</cache-properties>
```

See "Configuring XQS Caching" on page 8-66 for additional information.

Table 8–7 <in-memory> Attributes

Name	**Description**
useSpool	Values: Boolean Default: false Set `useSpool="true"` for memory-based caching of data, but with spooling enabled. This allows cached data to be spooled from memory to disk so that it is not lost if cached objects have to be removed from memory to reclaim space.
useDiskCache	Values: Boolean Default: false Set `useDiskCache="true"` for disk-based caching of data, which allows the cached data to be recovered in the event of a server crash.

<input-parameters>

Parent element: <wsdl-source> or <xqsview-source>

Child elements: <part>

Required? Required inside each occurrence of a parent element; one only

Use this element to specify input parameters to the XQS function for the associated data source. Use a `<part>` subelement for each parameter. Type-matching, for WSDL sources only, is according to the `type-match` attribute.

This element is required even if there are no input parameters; use an empty element in that case.

Table 8–8 <input-parameter> Attributes

Name	**Description**
type-match	Values: strict | none Default: strict For a WSDL source, this determines whether XQS performs type-matching for input parameters. With a setting of `type-match="strict"` (the default), XQS compares the input type you specify (through the `<schema-type>` subelement of `<part>`) to the type of the corresponding part in the WSDL document. "Strict" matching, the only level currently supported, requires an exact match of the type names. If the actual input type is not among the types supported by the `<schema-type>` element, choose the closest supported type and use a setting of `type-match="none"`. (Equivalently, you can avoid type-matching if you do not use a `<schema-type>` element.)

<itemType>

Parent element: <xquery-sequence>

Child elements: None

Required? Optional; zero or one inside each occurrence of the parent element

When you use the `<xquery-sequence>` element to specify a sequence type as an input type for an XQS view, where each member of the sequence is of the same type, you can use the `<itemType>` subelement to specify the type of the members. This is useful in allowing XQS to perform type-checking.

Use the `namespace` attribute to directly specify a namespace that defines the type; use the `prefix` attribute as a shortcut if you have previously defined a namespace prefix through a `<bind-prefix>` element. You must use one or the other, but do *not* use both `namespace` and `prefix` at the same time.

Table 8–9 <itemType> Attributes

Name	Description
namespace	Values: URI Default: None (required if `prefix` not used) Specifies the namespace directly.
prefix	Values: String Default: None (required if `namespace` not used) Specifies the namespace through a predefined prefix.
location	Values: URI Default: None For a user-defined type, you can specify a URI with the location where the schema that defines the type can be found. This is useful for type-checking by XQS, if a corresponding detailed declaration of the parameter exists in your XQuery prolog.

<mapping>

Parent element: <typeMap>

Child elements: <xmlType>

Required? Required inside the parent element if that is used; one or more

For a WSDL source with Java or EJB binding, use this in conjunction with its `<xmlType>` subelement to map between a Java type and an XML type. Use the `typeClass` attribute to specify the Java type.

Table 8–10 <mapping> Attributes

Name	Description
typeClass	Values: String Default: n/a (required) This attribute specifies a fully qualified Java class name for XML-Java type mapping. The class must be in the OC4J class path.

<operation>

Parent element: <wsdl-source>

Child elements: None

Required? Required inside each occurrence of the parent element; one only

Use this to specify the Web service operation to execute. It would be an operation defined in the WSDL document that the `<wsdlURL>` element points to, with the value of the XQS `<operation>` element corresponding to the `name` attribute of an `<operation>` element in the WSDL. Here is an example:

```
<operation>getCustomerByKey</operation>
```

The `<operation>` element has no attributes.

<output-element>

Parent element: <document-source>, <wsdl-source>, or <xqsview-source>

Child elements: None

Required? Optional; zero or one inside each occurrence of a parent element

The value of this element defines the item type of the sequence that is the result of the XQS function. This element is not required, but XQS can use the information for type-checking and optimizations. We especially advise you to use this element for an XQS view source with the setting `WSDLvisibility="true"`. XQS would still generate a valid Web service operation without the type information, but it is not a good practice to define an untyped Web service in a production system.

You can define the result item in either of two ways: by its type or by a reference to an element previously defined in another schema. The name of such an imported element is given as the text value of the element `<output-element>`. For example, if items in the result sequence will be `<po>` elements from the schema `urn:PurchaseOrders` namespace, `<output-element>` might look like this:

```
<output-element namespace="urn:purchaseOrders" > po </output-element>
```

If items in the result sequence will be elements of type `POType` defined in the schema `urn:PurchaseOrders` namespace, found at `http://myHost:/mySchemas/PurchaseOrders.xsd`, `<output-element>` might look like this:

```
<output-element prefix="po_ns"
location="http://myHost:/mySchemas/PurchaseOrders.xsd" type="POType"/>
```

Use the `namespace` attribute to specify the namespace directly; or use the `prefix` attribute as a shortcut if you have previously defined a namespace prefix through a `<bind-prefix>` element. You must use one or the other, but do *not* use both `namespace` and `prefix` at the same time.

Here is an example for a complex type:

```
<output-element namespace="http://xmlns.oracle.com/ias/xqs/CustomerDemo"
             location="http://host:port/xqsdemos/Customers.xsd"
type="CustomersType">
   cstomers
</output-element>
```

Here is an example for a simple type:

```
<output-element prefix="xs">float</output-element>
```

Table 8–11 <output-element> Attributes

Name	Description
namespace	Values: URI Default: None (required if `prefix` not used) Specifies a namespace directly.
prefix	Values: String Default: None (required if `namespace` not used) Specifies a namespace through a predefined prefix.
location	Values: URI Default: None Specify a URI with the location where the schema that defines the element or type can be found. This attribute is required when you use the `<output-element>` element in an `<xqsview-source>` element that has `WSDLvisibility="true"`. Otherwise it is not required, but is useful for type-checking by XQS if a corresponding detailed declaration of the parameter exists in your XQuery prolog.
type	Values: String Default: None Specify a local name for the XML type that will be the item type of the sequence that is the result of the XQS function. The namespace for the type name is determined from the `namespace` or `prefix` attribute. The `type` attribute is not required if the type of result item is defined through a reference to am element name given as the text value of the `<output-element>` element.

<part>

Parent element: <input-parameters>

Child elements: <schema-type>, <xquery-sequence>

Required? Required unless parent element is empty; one for each input argument

Use a `<part>` element for each input parameter for a WSDL source or XQS view. Specify the name and position (order) of the parameter through the attributes. Use a `<schema-type>` subelement or an `<xquery-sequence>` subelement (for XQS views only) to specify the type, as appropriate. Specifying the type is required for an XQS view that uses input parameters. For a WSDL source, specifying a type is not required, but is useful so that XQS can perform type-checking.

For a WSDL source, each `<part>` element must correspond to a part in the input message of the WSDL operation that is used to access the source. For an XQS view, each `<part>` element must correspond to an external bind variable of the XQuery that defines the view.

Table 8–12 <part> Attributes

Name	Description
name	Values: String Default: n/a (required) This specifies the name of the input parameter. For a WSDL source, the name must be the same as in the WSDL document. For an XQS view, it must be the same as the name of the external XQuery variable.
position	Values: Positive integer Default: n/a (required) The position attributes for all the `<part>` elements within an `<input-parameters>` element together determine the order in which the input parameters are assigned to message parts (for a WSDL source) or assigned to external variables (for an XQS view). The parameter with the lowest position setting, starting with `position="1"`, is taken first, and so on. Each `<part>` element must have a unique position setting.

<password>

Parent element: <document-source>

Child elements: None

Required? Optional; zero or one

This element is reserved for future security enhancements.

The `<password>` element has no attributes.

<port>

Parent element: <wsdl-source>

Child elements: None

Required? Required inside each occurrence of the parent element; one only

The value of this element specifies the name of a Web service port defined in the WSDL document that the `<wsdlURL>` element points to, corresponding to the `name` attribute of a `<port>` element in the WSDL.

Attribute settings specify the namespace, but you are not required to specify a namespace if the applicable service has only one port.

Use the `namespace` attribute to specify the namespace directly; use the `prefix` attribute as a shortcut if you have previously defined a namespace prefix through a `<bind-prefix>` element. Do *not* use both `namespace` and `prefix` at the same time.

For example:

```
<port namespace="http://customer.myeis.com/">CustomerInfo</port>
```

Alternatively:

```
<bind-prefix>
   <use-prefix prefix="myeis">http://customer.myeis.com/</use-prefix>
```

```
</bind-prefix>
...
<port prefix="myeis">CustomerInfo</port>
```

Table 8–13 <port> Attributes

Name	Description
namespace	Values: URI Default: None Specifies the namespace directly.
prefix	Values: String Default: None Specifies the namespace through a predefined prefix.

<portType>

Parent element: <wsdl-source>

Child elements: None

Required? May be required; zero or one inside each occurrence of the parent element

The value of this element specifies the name of a Web service port type defined in the WSDL document that the `<wsdlURL>` element points to, corresponding to the `name` attribute of a `<portType>` element in the WSDL. This element is required if the port used for the operation has multiple bindings in the WSDL.

Attribute settings specify the namespace, but you are not required to specify a namespace if all port types in the WSDL belong to the same namespace.

Use the `namespace` attribute to specify the namespace directly; use the `prefix` attribute as a shortcut if you have previously defined a namespace prefix through a `<bind-prefix>` element. Do *not* use both `namespace` and `prefix` at the same time.

For example:

```
<portType namespace="http://customer.myeis.com/">CustomerInfoType</portType>
```

Table 8–14 <portType> Attributes

Name	Description
namespace	Values: URI Default: None Specifies the namespace directly.
prefix	Values: String Default: None Specifies the namespace through a predefined prefix.

<queryName>

Parent element: <xqsview-source>

Child elements: None

Required? Optional; zero or one inside each occurrence of the parent element

The value of this element specifies the name of the XQuery expression text file (`.xq` file) where the XQS view is defined, such as in the following example:

```
<queryName>UserByOrderNum.xq</queryName>
```

You can leave off the `.xq` file name extension, as XQS assumes that automatically. The directory where XQS looks for the file is according to the `<repository>` subelement of `<xqsview-source>` (or else according to a default location).

If you do not use the `<queryName>` element, XQS assumes the XQS view file to have the same name as the function you specify in the `<function-name>` subelement of `<xqsview-source>`.

The `<queryName>` element has no attributes.

<repository>

Parent element: <xqsview-source>

Child elements: None

Required? Optional; zero or one inside each occurrence of the parent element

The value of this element is the absolute path to the directory containing the XQuery expression text file (`.xq` file) where the XQS view is defined. Specify the file name in the `<queryName>` subelement of `<xqsview-source>`.

If you do not use the `<repository>` element, XQS looks in the directory you specified through the OC4JPackager `-repository` option. Here is an example explicitly specifying what would be the default directory for the XQS demo application:

```
<repository>/META-INF/xqs/mydir/</repository>
```

The `<repository>` element has no attributes.

<schema-file>

Parent element: <XMLTranslate>

Child elements: None

Required? Required if the parent element is used; one only

If a document source is non-XML and conforms to formats supported by the D3L tool, use the `<XMLTranslate>` element and `<schema-file>` subelement to give information to XQS about how to translate the document to XML for you. (See "Preparing to Use a Non-XML Document Source" on page 8-15 for information about D3L.)

The `<XMLTranslate>` element specifies the translation tool to use (D3L), and the `<schema-file>` element specifies the schema file to use in translation.

Here is an example:

```
<documentURL>http://host:port/xqsdemos/paymentInfo.csv</documentURL>
<XMLTranslate method="D3L">
   <schema-file>http://host:port/xqsdemos/paymentInfoD3L.xml</schema-file>
```

```
</XMLTranslate>
```

The `<schema-file>` element has no attributes.

<schema-type>

Parent element: <part>

Child elements: None

Required? See description; one only

For each input parameter for an XQS view, either this element or the `<xquery-sequence>` element (for an input sequence) is required to specify the parameter type.

For each input parameter for a WSDL source, this element is optional inside the `<part>` parent element, but is useful for type-checking, or for organizational purposes—to have all your types listed in the XQS configuration file. (A WSDL source cannot have an input sequence.)

Note that to perform type-checking for WSDL sources, the `<input-parameters>` element must have the attribute setting `type-match="strict"`.

For `<schema-type>`, the element value specifies the type, and attribute settings specify the namespace. Use the `namespace` attribute to specify the namespace directly; use the `prefix` attribute as a shortcut if you have previously defined a namespace prefix through a `<bind-prefix>` element. You must use one or the other, but do *not* use both `namespace` and `prefix` at the same time.

Here is an example:

```
<input-parameters>
  <part position="1" name="empname">
    <schema-type namespace="http://www.w3.org/2001/XMLSchema">string</schema-type>
  </part>
</input-parameters>
```

Or, alternatively, defining `xs` as a prefix for the XMLSchema namespace:

```
<bind-prefix>
   <use-prefix prefix="xs">http://www.w3.org/2001/XMLSchema</use-prefix>
</bind-prefix>
...
<input-parameters>
  <part position="1" name="empname">
    <schema-type prefix="xs">string</schema-type>
  </part>
</input-parameters>
```

Or, for a user-defined type:

```
<input-parameters>
   <part position="1" name="DBServiceSelect_cust_id_inparameters">
      <schema-type namespace=
         "http://xmlns.oracle.com/pcbpel/adapter/db/top/TestDBAdapter">
         DBServiceSelect_cust_id
      </schema-type>
   </part>
</input-parameters>
```

In addition to any user-defined XML types, XQS currently supports the following subset of types defined in the XQuery 1.0 and XPath 2.0 data model: `xs:boolean`, `xs:string`, `xs:int`, `xs:long`, `xs:float`, `xs:double`, `xs:decimal`, `xs:base64Binary`, `xs:hexBinary`, `xs:anyURI`, `xs:dateTime`, `xs:duration`, and `xs:anyType`. (Also see "Supported Types for Query Parameters" on page 8-40.) If you have an XQuery/XPath input type that is not among these supported types, choose a supported type with the Java representation that best suits your data, and use a setting of `type-match="none"` in the `<input-parameters>` element. (For a WSDL source with a Java or EJB binding, this is a situation where you would also use the `<typeMap>` subelement of `<wsdl-source>`.)

Table 8–15 <schema-type> Attributes

Name	Description
namespace	Values: URI Default: None (required if `prefix` not used) Specifies the namespace directly.
prefix	Values: String Default: None (required if `namespace` not used) Specifies the namespace through a predefined prefix.

<service>

Parent element: <wsdl-source>

Child elements: None

Required? May be required; zero or one inside each occurrence of the parent element

The value of this element specifies the name of a service. It would be a service defined in the WSDL document that the `<wsdlURL>` element points to, with the value of the XQS `<service>` element corresponding to the `name` attribute of a `<service>` element in the WSDL. The XQS `<service>` element is required if the WSDL document defines multiple services.

Attribute settings specify the namespace, but you are not required to specify a namespace if all service elements in the WSDL belong to the same namespace.

Use the `namespace` attribute to specify the namespace directly; use the `prefix` attribute as a shortcut if you have previously defined a namespace prefix through a `<bind-prefix>` element. Do *not* use both `namespace` and `prefix` at the same time.

For example:

```
<service namespace="http://customer.myeis.com/">CustomerInfoMYEISService</service>
```

Table 8–16 <service> Attributes

Name	Description
namespace	Values: URI Default: None Specifies the namespace directly.
prefix	Values: String Default: None Specifies the namespace through a predefined prefix.

<typeMap>

Parent element: <wsdl-source>

Child elements: <mapping>

Required? Optional; zero or one within each occurrence of the parent element

For a WSDL source with a Java or EJB binding, you can use this element as necessary to map XML types to Java types. Specify the Java type through the `<mapping>` subelement, and the XML type through the `<xmlType>` subelement of `<mapping>`.

The `<typeMap>` element has no attributes.

<use-prefix>

Parent element: <bind-prefix>

Child elements: None

Required? Required if the parent element is used; one or more

Use a `<bind-prefix>` element and `<use-prefix>` subelements if you want to designate prefixes to represent certain XML namespaces. Each `<use-prefix>` subelement designates one prefix, with the element value specifying a namespace and the prefix attribute specifying the prefix to represent that namespace. Here is an example:

```
<bind-prefix>
   <use-prefix prefix="xs">http://www.w3.org/2001/XMLSchema</use-prefix>
   <use-prefix prefix="xqs">http://xmlns.oracle.com/ias/xqs</use-prefix>
   <use-prefix prefix="myeis">http://customer.myeis.com/</use-prefix>
</bind-prefix>
```

Given these settings, XQS configuration elements that must specify a namespace can use, for example, `prefix="myeis"` instead of `namespace="http://customer.myeis.com/"`.

Note: A `<use-prefix>` designation is valid only in the XQS configuration file in which it appears, not in XQuery expressions.

Table 8–17 <use-prefix> Attributes

Name	Description
prefix	Values: String Default: n/a (required) This attribute specifies the desired prefix.

<username>

Parent element: <document-source>

Child elements: None

Required? Optional; zero or one

This element is reserved for future security enhancements.

The `<username>` element has no attributes.

<wsdl-source>

Parent element: <xqs-sources>

Child elements: <cache-properties>, <error-message>, <function-name>, <input-parameters>, <operation>, <output-element>, <port>, <portType>, <service>, <typeMap>, <wsdlURL>

Required? Optional; zero or more

Use a `<wsdl-source>` element and its subelements (as appropriate) for each XQS function you configure that will access a WSDL-based source. (See "Supported Categories of Data Sources" on page 8-7 for an overview of the kinds of WSDL-based sources you can use with XQS.)

Table 8–18 <wsdl-source> Attributes

Name	Description
fetchSize	Values: Integer Default: 1 This attribute takes effect only for WSDL sources that use XQS SQL binding to connect to a relational database. The attribute is optional, and the default is 1 row. The `fetchSize` attribute is translated into a call to the `setFetchSize` method of `java.sql.PreparedStatement`. The attribute serves as a hint to JDBC to define the number of rows in the result set that will be fetched from the database in one round trip. Note that the `setFetchSize` parameter in JDBC (and, therefore, the `fetchSize` attribute in XQS) is only a hint; it is not binding.
isCached	Values: Boolean Default: false Set this to "`true`" to cache results from the data source. Also see "Configuring XQS Caching" on page 8-66 and information about the `<cache-properties>` element.
largeData	Values: Boolean Default: false Set this to "`true`" if you want to provide a hint to XQS that it should try to use memory-saving internal optimizations to handle large volumes of data. (Use only when necessary.) Also see "Configuring XQS Document or View Sources for Large Data" on page 8-68.
onError	Values: dynamicError | emptySequence | errorMessage Default: dynamicError This determines which error-handling mode XQS uses. See "Introduction to XQS Error Handling" on page 8-13.

<wsdlURL>

Parent element: <wsdl-source>

Child elements: None

Required? Required inside each occurrence of the parent element; one only

Use this to point to the WSDL document that defines the operation to be executed. The WSDL will be accessed at runtime. Here is an example:

```
<wsdlURL>http://api.mySearch.com/mySearch.wsdl</wsdlURL>
```

The `<wsdlURL>` element has no attributes.

Note: If you are behind a firewall and the specified URL requires external Internet access, be aware that you must also configure OC4J with appropriate proxy settings.

<xqs-config>

Parent element: n/a (root)

Child elements: <bind-prefix>, <xqs-sources>

Required? Required; one only

This is the root element of `xqs-config.xml` and `global-xqs-config.xml`. At a minimum, you must have this element, its `<xqs-sources>` subelement, and at least one subelement of `<xqs-sources>`—`<document-source>`, `<wsdl-source>`, or `<xqsview-source>`.

Table 8–19 <xqs-config> Attributes

Name	Description
version	Values: XML name token (`NMTOKEN`) Default: 1.0 (current version) This attribute notes the version number of the `xqs-config.xml` schema definition. For the XQS 10.1.3 implementation, use the default value.

<xqs-sources>

Parent element: <xqs-config>

Child elements: <document-source>, <wsdl-source>, <xqsview-source>

Required? Required; one only

This is the parent element for the source elements for all XQS functions being configured. It does not have any attributes.

<xqsview-source>

Parent element: <xqs-sources>

Child elements: <cache-properties>, <error-message>, <function-name>, <input-parameters>, <output-element>, <queryName>, <repository>

Required? Optional; zero or more

Use an `<xqsview-source>` element and its subelements (as appropriate) for each XQS function you configure that uses an XQS view.

Table 8–20 <xqsview-source> Attributes

Name	Description
isCached	Values: Boolean Default: false Set this to "`true`" to cache results from the data source. Also see "Configuring XQS Caching" on page 8-66 and information about the `<cache-properties>` element.
largeData	Values: Boolean Default: false Set this to "`true`" if you want to provide a hint to XQS that it should try to use memory-saving internal optimizations to handle large volumes of data. (Use only when necessary.) Also see "Configuring XQS Document or View Sources for Large Data" on page 8-68.
onError	Values: dynamicError | emptySequence | errorMessage Default: dynamicError This determines which error-handling mode XQS uses. See "Introduction to XQS Error Handling" on page 8-13.
WSDLvisibility	Values: Boolean Default: false Set this to "`true`" if you want XQS to expose the view as a Web service operation in the Web service component of your application. Also see "Using an XQS View Exposed as a Web Service Operation" on page 8-58.

<XMLTranslate>

Parent element: <document-source>

Child elements: <schema-file>

Required? Optional; zero or one inside each occurrence of the parent element

If a document source is non-XML and conforms to formats supported by the D3L tool, use the `<XMLTranslate>` element and `<schema-file>` subelement to give information to XQS about how to translate the document to XML for you. This is useful for translating formatted files, such as Excel comma-separated-values (CSV) files, to XML. See "Preparing to Use a Non-XML Document Source" on page 8-15 for information about D3L.

The `method` attribute of `<XMLTranslate>` specifies the translation tool to use. XQS currently supports the D3L translation tool.

The `<schema-file>` subelement is required, to specify the schema to use in translation.

Table 8–21 <XMLTranslate> Attributes

Name	Description
method	Values: D3L Default: n/a (required) Specifies the translation tool for XQS to use. XQS currently supports D3L.

<xmlType>

Parent element: <mapping>

Child elements: None

Required? Required inside each occurrence of the parent element; one only

For a WSDL source with Java or EJB binding, use this in conjunction with its `<mapping>` parent element to map between a Java type and an XML type. Use attribute settings to specify the namespace of the XML type: use the `namespace` attribute to specify the namespace directly; use the `prefix` attribute as a shortcut if you have previously defined a namespace prefix through a `<bind-prefix>` element. You must use one or the other, but do *not* use both `namespace` and `prefix` at the same time.

The `<mapping>` parent element specifies the Java type.

Here is an example:

```
<typeMap>
   <mapping typeClass="org.w3c.dom.Node">
      <xmlType prefix="myeis">Customer</xmlType>
   </mapping>
   ...
</typeMap>
```

Table 8–22 <xmlType> Attributes

Name	Description
namespace	Values: URI Default: None (required if `prefix` not used) Specifies the namespace directly.
prefix	Values: String Default: None (required if `namespace` not used) Specifies the namespace through a predefined prefix.

<xquery-sequence>

Parent element: <part>

Child elements: <itemType>

Required? This or <schema-type> is required inside each occurrence of the parent element or an XQS view; one only

For each input parameter for an XQS view, where the parameter is an XQuery/XPath sequence type, you must use this element (as opposed to the `<schema-type>` element for a non-sequence type) inside the `<part>` parent element to specify the type.

For a WSDL source, input sequences are not supported so this element is not relevant.

The `<xquery-sequence>` element can be used for a heterogeneous sequence consisting of items of multiple types, in which case it should be an empty element. Alternatively, it can include the `<itemType>` subelement to specify the common type of each member of a homogeneous sequence.

The `<xquery-sequence>` element has no attributes.

OC4JPackager Reference

This section provides reference documentation for parameters and properties of the XQS packager utility, OC4JPackager. For an overview, see "Introduction to OC4JPackager" on page 8-11. For usage instructions, see "How to Use OC4JPackager to Package Your XQS Application" on page 8-58.

Note: For specifying paths in parameter or property settings, OC4JPackager supports paths that are either relative to the current directory (including the use of "`.`" and "`..`" directory notation) or absolute.

OC4JPackager Parameters

This section documents OC4JPackager parameters in alphabetical order, with descriptions and usage examples. Run OC4JPackager as follows (where `%` is the command prompt):

```
% java -jar OC4JPackager.jar -option1 value1 -option2 value2 ... -optionN valueN
```

appArchives

Required. Specify a directory path.

Use this to specify where your application is located—either the directory containing an EAR file, or the top-level directory of an EAR structure.

```
-appArchives /dir1/myappdir
```

globalXqsConfig

Optional. Specify a file path.

If any of the XQS views you want to expose as Web services are configured in the `global-xqs-config.xml` file for global access, use this parameter to specify the path (including file name) of this file. This will bring those views into your local configuration.

```
-globalXqsConfig ORACLE_HOME/j2ee/home/config/global-xqs-config.xml
```

help

Optional. This is a toggle flag.

Use this without any other parameters to show OC4JPackager usage information and a parameter list.

```
-help
```

jsp

Optional (required if you use the XQS JSP tag library). This is a toggle flag.

Use this to include the XQS JSP tag library JAR file (`xquerytaglib.jar`) in the output EAR file.

```
-jsp
```

name

Required. Specify an EAR file name.

Use this to specify the desired name of the output EAR file (with or without the `.ear` extension). Its location will be determined by the `-output` parameter.

```
-name myxqsapp.ear
```

output

Required. Specify a directory path.

Use this to specify where OC4JPackager should place the EAR file that it creates. The file name is according to the `-name` parameter.

```
-output /dir1/mydeployments
```

repository

Optional. Specify a directory path.

Use this to specify your XQS repository location, where your XQS views are stored.

```
-repository ORACLE_HOME/xds/samples/repository
```

sf

Optional (required if you use the XQS stateful EJB client API). This is a toggle flag.

Use this to include the XQS stateful EJB client JAR file (`xqsclient-stateful.jar`) in the output EAR file.

```
-sf
```

This also results in the following `<module>` element being added to the `application.xml` file:

```
<module>
   <ejb>xqsclient-stateful.jar</ejb>
</module>
```

sl

Optional (required if you use the XQS stateless EJB client API). This is a toggle flag.

Use this to include the XQS stateless EJB client JAR file (`xqsclient-stateless.jar`) in the output EAR file.

```
-sl
```

This also results in the following `<module>` element being added to the `application.xml` file:

```
<module>
   <ejb>xqsclient-stateless.jar</ejb>
</module>
```

xqsConfigFile

Required. Specify a file path.

Use this to specify the path (including file name) of the XQS configuration file, `xqs-config.xml`, for your application.

```
-xqsConfigFile /dir1/subdir/myconfigdir/xqs-config.xml
```

Java Properties for OC4JPackager

This section documents Java properties supported by OC4JPackager, in alphabetical order, with descriptions and usage examples. Use the Java `-D` option, as follows:

```
java -Dproperty1=value1 -Dproperty2=value2 -jar OC4JPackager.jar \
     -option1 value1 -option2 value2 ... -optionN valueN
```

oracle.home

Optional. Specify a directory path.

OC4JPackager supports the `oracle.home` property to specify your Oracle home directory. For example:

```
-Doracle.home=/myroot/myoraclehome
```

Specifying an Oracle home directory allows XQS to find any client JAR files requested through the `-jsp`, `-sf`, and `-sl` flags and bundle them with your application. If you do not set `oracle.home`, these flags are ignored.

java.home

Required. Specify a directory path.

Specify the path to your Java home directory, from which OC4JPackager will run the `java` command for its work (such as unbundling and bundling JAR files).

```
-Djava.home=/dir1/myjavahome
```

OC4JPackager will exit if you do not specify this property.

java.util.logging.properties.file

Optional. Specify a file path.

Use this to indicate a path to the file, including the file name, where you specify Java logging properties, as in the following example (where `myfile.properties` is in the current directory):

```
-Djava.util.logging.properties.file=myfile.properties
```

Without specifying a logging properties file, you will not see any Java error output.

OC4JPackager logs messages using the standard J2SE logging framework, as described at the following location:

http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/

Within the properties file, OC4JPackager supports the property `oracle.xds.tools.OC4JPackager.level` to specify any of the following logging levels for messages from the packager: `SEVERE`, `WARNING`, `INFO`, or `FINE`. For high-level progress messages use `INFO`; for low-level debugging messages, use `FINE`.

Here are sample entries for a logging properties file, to send messages to the console and use the `INFO` level for messages from OC4JPackager:

```
#  output to console
handlers= java.util.logging.ConsoleHandler
...
# set packager level to INFO
oracle.xds.tools.OC4JPackager.level=INFO
```

xds.packager.work.dir

Optional. Specify a directory path.

Use this to specify a working directory that OC4JPackager can use when it unbundles and rebuilds EAR files.

```
-Dxds.packager.work.dir=/some/dir
```

The default location is your `user.home` directory.

Summary of XQS MBeans and Administration

Standards-compliant MBeans play a role in OC4J runtime configuration. The following sections provide an overview:

- General Overview of OC4J MBean Administration
- Summary of XQS MBeans

Note: This information is provided for reference, but for the current release we advise you to make your XQS configuration settings directly through the configuration files. See "How to Configure Your XQS Functions" on page 8-24.

General Overview of OC4J MBean Administration

OC4J support for the JMX specification (JSR-77) allows standard interfaces to be created for managing resources dynamically, including resources relating to XQS, in a J2EE environment. The OC4J implementation of JMX provides a JMX client, the System MBean Browser, that you can use to manage an OC4J instance through MBeans that are provided with OC4J.

An MBean is a Java object that represents a JMX manageable resource. Each manageable resource within OC4J, such as an application, is managed through an instance of the appropriate MBean. Each MBean provided with OC4J exposes a management interface that is accessible through the System MBean Browser in the Application Server Control Console. You can set MBean attributes, execute operations to call methods on an MBean, subscribe to notifications of errors or specific events, and display execution statistics.

To access the browser from the OC4J home page, select the Administration tab and then, under the list of tasks, go to the task "System MBean Browser". From the browser, you can do the following:

- Select the MBean of interest in the left-hand frame.
- Use the Attributes tab in the right-hand frame to view or change attributes. An attribute that can be set has a field where you can type in a new value. Then apply the change.
- Use the Operations tab in the right-hand frame to invoke methods on the MBean. Select the operation of interest, and then specify to invoke it.
- Use the Notifications tab (where applicable) in the right-hand frame to subscribe to notifications. You can select each item for which you want notification, and then apply the changes.
- Use the Statistics tab (where applicable) in the right-hand frame to display execution statistics.

Be aware that MBeans and their attributes vary regarding when changes take effect. In the *runtime model*, changes take effect immediately. In the *configuration model*, some changes take effect when the resource is restarted, others when the application is restarted, and still others when OC4J is restarted. There is also variation in whether changes are persisted.

See the ***Oracle Containers for J2EE Configuration and Administration Guide*** for additional general information about OC4J MBeans. The System MBean Browser itself also provides information about the MBeans.

Summary of XQS MBeans

Table 8–23 summarizes the OC4J implementation of MBeans that relate to XQS. These implementations are in the `oracle.xds.management` package.

Note that if you use these MBeans, XQS does the work of persisting values to the `xqs-config.xml` (or `global-xqs-config.xml`) configuration file, as soon as you make the updates.

Note: MBeans are self-documenting in the System MBean Browser, providing some documentation of MBean attributes, operations, and notifications (as applicable).

Table 8–23 System MBeans for XQS

MBean	Description
XDSConfig	The overall configuration MBean for XQS.
XDSDocumentSourceConfig	The configuration MBean for XQS document sources, corresponding to the XQS `<document-source>` configuration element.
XDSWsdlSourceConfig	The configuration MBean for XQS WSDL sources, corresponding to the XQS `<wsdl-source>` configuration element.
XDSViewSourceConfig	The configuration MBean for XQS views, corresponding to the `<xqsview-source>` configuration element.

XQS Troubleshooting

This section describes how to enable OC4J logging and offers an assortment of troubleshooting tips for your XQS application.

Enabling OC4J Logging

You can enable OC4J logging by using the Java property `java.util.logging.properties.file` to specify a logging properties file. In this file you can specify Java logging settings, including the OC4JPackager logging level. Logging properties are defined by standard J2SE logging, as specified at the following location:

`http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/`

For more information about logging, see "java.util.logging.properties.file" on page 8-108, the chapter on logging in OC4J in the *Oracle Containers for J2EE Configuration and Administration Guide*, and the logging implementation guidelines in the *Oracle Containers for J2EE Developer's Guide*.

Key XQS Symptoms, Causes, and Remedies

This section discusses possible problems you may encounter with your XQS application, and their solutions.

Note: There are no Dynamic Monitoring Service (DMS) metrics for the XQS 10.1.3 implementation.

Trouble with large volumes of data: If you have trouble loading large XML documents into memory, enabling the `largeData` flag may allow you to proceed. See "Configuring XQS Document or View Sources for Large Data" on page 8-68.

XQS function did not load properly: You may see an error message such as the following (where "arity" refers to the number of arguments the function takes, and "*`ns`*" and "*`funcname`*" are replaced by the namespace and name of your function):

```
XP0017: It is a static error if the expanded QName and number of arguments in a
function call do not match the name and arity of an in-scope function in the
static context. Detail: unknown function 'ns:funcname'
```

The key point is that XQuery did not recognize the function, which is an indication that the function could not be loaded properly by XQS, perhaps because it was not configured correctly. Examine your OC4J error log file (such as *`ORACLE_HOME`*`/j2ee/home/log/oc4j/log.xml`) and look for a detailed explanation of the particular problem with this function.

External function has inconsistent signatures: Consider an error such as the following:

```
external function "Foo" has inconsistent signatures: the one defined in function
library framework is different from the one declared in function declaration
prolog
```

Be aware that the term "function library framework" in the context of XQS refers to the signature defined within an `<input-parameters>` element in the XQS configuration file. You must ensure that `<part>` definitions under the `<input-parameters>` element match function signatures in your XQuery prologs.

If you have trouble bringing your XQuery signatures into consistency with your configuration, you may be trying to use an unsupported XML type. (See "Supported Types for Query Parameters" on page 8-40.) As a workaround, however, you can avoid prolog type declarations for XQuery parameters or XQuery return values altogether. For example, instead of a fully typed function declaration such as the following:

```
declare function xqs:Foo ($s as element()) as xs:positiveInteger external;
```

You can declare the function without types, as follows:

```
declare function xqs:Foo ($s) external;
```

However, you are always required to specify the number of input parameters, through your XQS configuration.

XQS Sample

For an XQS sample, see the technology preview of Oracle XML Query Service on the Oracle Technology Network at the following location:

http://www.oracle.com/technology/sample_code/tech/java/oc4j/index.html

A

Third Party Licenses

This appendix includes the third party licenses for third party products related to content discussed in this book.

ANTLR

This program contains third-party code from ANTLR. Under the terms of the Apache license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the ANTLR software, and the terms contained in the following notices do not change those rights.

The ANTLR License

Apache

This program contains third-party code from the Apache Software Foundation ("Apache"). Under the terms of the Apache license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the Apache software, and the terms contained in the following notices do not change those rights.

The Apache Software License

License for Apache Web Server 1.3.29

```
/* ====================================================================
```

License for Apache Web Server 2.0

Apache SOAP

This program contains third-party code from the Apache Software Foundation ("Apache"). Under the terms of the Apache license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the Apache software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the Apache software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or Apache.

Apache SOAP License

Apache SOAP license 2.3.1

DBI Module

Perl Artistic License

The "Artistic License"

Preamble

The intent of this document is to state the conditions under which a Package may be copied, such that the Copyright Holder maintains some semblance of artistic control over the development of the package, while giving the users of the package the right to use and distribute the Package in a more-or-less customary fashion, plus the right to make reasonable modifications.

Definitions

"Package" refers to the collection of files distributed by the Copyright Holder, and derivatives of that collection of files created through textual modification.

"Standard Version" refers to such a Package if it has not been modified, or has been modified in accordance with the wishes of the Copyright Holder as specified below.

"Copyright Holder" is whoever is named in the copyright or copyrights for the package.

"You" is you, if you're thinking about copying or distributing this Package.

"Reasonable copying fee" is whatever you can justify on the basis of media cost, duplication charges, time of people involved, and so on. (You will not be required to justify it to the Copyright Holder, but only to the computing community at large as a market that must bear the fee.)

"Freely Available" means that no fee is charged for the item itself, though there may be fees involved in handling the item. It also means that recipients of the item may redistribute it under the same conditions they received it.

1. You may make and give away verbatim copies of the source form of the Standard Version of this Package without restriction, provided that you duplicate all of the original copyright notices and associated disclaimers.
2. You may apply bug fixes, portability fixes and other modifications derived from the Public Domain or from the Copyright Holder. A Package modified in such a way shall still be considered the Standard Version.
3. You may otherwise modify your copy of this Package in any way, provided that you insert a prominent notice in each changed file stating how and when you changed that file, and provided that you do at least ONE of the following:
   a. place your modifications in the Public Domain or otherwise make them Freely Available, such as by posting said modifications to Usenet or an equivalent medium, or placing the modifications on a major archive site such as uunet.uu.net, or by allowing the Copyright Holder to include your modifications in the Standard Version of the Package.
   b. use the modified Package only within your corporation or organization.
   c. rename any non-standard executables so the names do not conflict with standard executables, which must also be provided, and provide a separate manual page for each non-standard executable that clearly documents how it differs from the Standard Version.
   d. make other distribution arrangements with the Copyright Holder.
4. You may distribute the programs of this Package in object code or executable form, provided that you do at least ONE of the following:
   a. distribute a Standard Version of the executables and library files, together with instructions (in the manual page or equivalent) on where to get the Standard Version.
   b. accompany the distribution with the machine-readable source of the Package with your modifications.
   c. give non-standard executables non-standard names, and clearly document the differences in manual pages (or equivalent), together with instructions on where to get the Standard Version.
   d. make other distribution arrangements with the Copyright Holder.
5. You may charge a reasonable copying fee for any distribution of this Package. You may charge any fee you choose for support of this Package. You may not charge a fee for this Package itself. However, you may distribute this Package in aggregate with other (possibly commercial) programs as part of a larger (possibly commercial) software distribution provided that you do not advertise this Package as a product of your own. You may embed this Package's interpreter within an executable of yours (by linking); this shall be construed as a mere form of aggregation, provided that the complete Standard Version of the interpreter is so embedded.
6. The scripts and library files supplied as input to or produced as output from the programs of this Package do not automatically fall under the copyright of this Package, but belong to whoever generated them, and may be sold commercially, and may be aggregated with this Package. If such scripts or library files are aggregated with this Package through the so-called "undump" or "unexec"

methods of producing a binary executable image, then distribution of such an image shall neither be construed as a distribution of this Package nor shall it fall under the restrictions of Paragraphs 3 and 4, provided that you do not represent such an executable image as a Standard Version of this Package.

7. C subroutines (or comparably compiled subroutines in other languages) supplied by you and linked into this Package in order to emulate subroutines and variables of the language defined by this Package shall not be considered part of this Package, but are the equivalent of input as in Paragraph 6, provided these subroutines do not change the language in any way that would cause it to fail the regression tests for the language.
8. Aggregation of this Package with a commercial distribution is always permitted provided that the use of this Package is embedded; that is, when no overt attempt is made to make this Package's interfaces visible to the end user of the commercial distribution. Such use shall not be construed as a distribution of this Package.
9. The name of the Copyright Holder may not be used to endorse or promote products derived from this software without specific prior written permission.
10. THIS PACKAGE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.

The End

expat

This program contains third-party code from CPAN. Under the terms of the CPAN license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the CPAN software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the CPAN software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or CPAN.

FastCGI

This program contains third-party code from Open Market, Inc. Under the terms of the Open Market license, Oracle is required to license the Open Market software to you under the following terms. Note that the terms contained in the Oracle program license that accompanied this product do not apply to the Open Market software, and your rights to use the software are solely as set forth below. Oracle is not responsible for the performance of the Open Market software, does not provide technical support for the software, and shall not be liable for any damages arising out of any use of the software.

FastCGI Developer's Kit License

This FastCGI application library source and object code (the "Software") and its documentation (the "Documentation") are copyrighted by Open Market, Inc ("Open Market"). The following terms apply to all files associated with the Software and Documentation unless explicitly disclaimed in individual files.

Open Market permits you to use, copy, modify, distribute, and license this Software and the Documentation solely for the purpose of implementing the FastCGI specification defined by Open Market or derivative specifications publicly endorsed

Module mod_fastcgi License

Open Market shall retain all right, title and interest in and to the Software and Documentation, including without limitation all patent, copyright, trade secret and other proprietary rights.

OPEN MARKET MAKES NO EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE SOFTWARE OR THE DOCUMENTATION, INCLUDING WITHOUT LIMITATION ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL OPEN MARKET BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY DAMAGES ARISING FROM OR RELATING TO THIS SOFTWARE OR THE DOCUMENTATION, INCLUDING, WITHOUT LIMITATION, ANY INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES OR SIMILAR DAMAGES, INCLUDING LOST PROFITS OR LOST DATA, EVEN IF OPEN MARKET HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS". OPEN MARKET HAS NO LIABILITY IN CONTRACT, TORT, NEGLIGENCE OR OTHERWISE ARISING OUT OF THIS SOFTWARE OR THE DOCUMENTATION.

Info-ZIP Unzip Package

This program contains third-party code from Info-ZIP. Under the terms of the Info-ZIP license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the Info-ZIP software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the Info-ZIP software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or Info-ZIP.

The Info-ZIP Unzip Package License

```
Copyright (c) 1990-1999 Info-ZIP. All rights reserved. For the purposes of this
copyright and license, "Info-ZIP" is defined as the following set of individuals:

Mark Adler, John Bush, Karl Davis, Harald Denker, Jean-Michel Dubois, Jean-loup
Gailly, Hunter Goatley, Ian Gorman, Chris Herborth, Dirk Haase, Greg Hartwig,
Robert Heath, Jonathan Hudson, Paul Kienitz, David Kirschbaum, Johnny Lee, Onno
van der Linden, Igor Mandrichenko, Steve P. Miller, Sergio Monesi, Keith Owens,
George Petrov, Greg Roelofs, Kai Uwe Rommel, Steve Salisbury, Dave Smith,
Christian Spieler, Antoince Verheijen, Paul von Behren, Rich Wales, Mike White

This software is provided "AS IS," without warranty of any kind, express or
implied.  In no event shall InfoZIP or its contributors be held liable for any
direct, indirect, incidental, special or consequential damages arising out of the
use of or inability to use this software."
```

Jabberbeans

This program contains third-party code from Jabber, Inc. Under the terms of the Jabber Open Source License, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the JabberBeans software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the JabberBeans software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or Jabber, Inc.

The source code for JabberBeans is available from Jabber, Inc. at `www.jabber.com`.

JSR 110

This program contains third-party code from IBM Corporation ("IBM"). Under the terms of the IBM license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the IBM software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the IBM software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or IBM.

Jaxen

This program contains third-party code from the Apache Software Foundation ("Apache") and from the Jaxen Project ("Jaxen"). Under the terms of the Apache and Jaxen licenses, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the Apache and Jaxen software, and the terms contained in the following notices do not change those rights.

The Jaxen License

JGroups

This program contains third-party code from GNU. Under the terms of the GNU license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the GNU software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the GNU software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or GNU.

The GNU License

or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e) Verify that the user has already received a copy of these materials or that you

JTidy

This program contains third-party code known as Java HTML Tidy "JTidy" from the World Wide Web Consortium ("W3C") and the following contributing authors: Dave Raggett dsr@w3.org, Andy Quick <ac.quick@sympatico.ca> (translation to Java), Gary L Peskin <garyp@firstech.com> (Java development), Sami Lempinen <sami@lempinen.net> (release management). Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the W3C JTidy software. Notwithstanding anything to the contrary in the Oracle program license, the W3C JTidy software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle, the W3C or the contributing authors.

mod_dav

This program contains third-party code from Greg Stein. Under the terms of the Greg Stein license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the Greg Stein software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the Greg Stein software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or Greg Stein.

mod_mm and mod_ssl

This program contains third-party code from Ralf S. Engelschall ("Engelschall"). Under the terms of the Engelschall license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the Engelschall software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the mod_mm software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or Engelschall.

mod_mm

mod_ssl

OpenSSL

This program contains third-party code from the OpenSSL Project. Under the terms of the OpenSSL Project license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the OpenSSL software, and the terms contained in the following notices do not change those rights.

OpenSSL License

Perl

This program contains third-party code from the Comprehensive Perl Archive Network ("CPAN"). Under the terms of the CPAN license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the CPAN software, and the terms contained in the following notices do not change those rights.

Perl Kit Readme

provide the Perl source as specified by the GPL. (The fact that a Perl interpreter and your code are in the same binary file is, in this case, a form of mere aggregation.) This is my interpretation of the GPL. If you still have concerns or difficulties understanding my intent, feel free to contact me. Of course, the Artistic License spells all this out for your protection, so you may prefer to use that.

mod_perl 1.29 License

mod_perl 1.99_16 License

```
of any other Contributor, and only if You agree to indemnify,
defend, and hold each Contributor harmless for any liability
incurred by, or claims asserted against, such Contributor by reason
of your accepting any such warranty or additional liability.
```

Perl Artistic License

The "Artistic License"

Preamble

The intent of this document is to state the conditions under which a Package may be copied, such that the Copyright Holder maintains some semblance of artistic control over the development of the package, while giving the users of the package the right to use and distribute the Package in a more-or-less customary fashion, plus the right to make reasonable modifications.

Definitions

"Package" refers to the collection of files distributed by the Copyright Holder, and derivatives of that collection of files created through textual modification.

"Standard Version" refers to such a Package if it has not been modified, or has been modified in accordance with the wishes of the Copyright Holder as specified below.

"Copyright Holder" is whoever is named in the copyright or copyrights for the package.

"You" is you, if you're thinking about copying or distributing this Package.

"Reasonable copying fee" is whatever you can justify on the basis of media cost, duplication charges, time of people involved, and so on. (You will not be required to justify it to the Copyright Holder, but only to the computing community at large as a market that must bear the fee.)

"Freely Available" means that no fee is charged for the item itself, though there may be fees involved in handling the item. It also means that recipients of the item may redistribute it under the same conditions they received it.

1. You may make and give away verbatim copies of the source form of the Standard Version of this Package without restriction, provided that you duplicate all of the original copyright notices and associated disclaimers.
2. You may apply bug fixes, portability fixes and other modifications derived from the Public Domain or from the Copyright Holder. A Package modified in such a way shall still be considered the Standard Version.
3. You may otherwise modify your copy of this Package in any way, provided that you insert a prominent notice in each changed file stating how and when you changed that file, and provided that you do at least ONE of the following:
   a. place your modifications in the Public Domain or otherwise make them Freely Available, such as by posting said modifications to Usenet or an equivalent medium, or placing the modifications on a major archive site such as uunet.uu.net, or by allowing the Copyright Holder to include your modifications in the Standard Version of the Package.
   b. use the modified Package only within your corporation or organization.
   c. rename any non-standard executables so the names do not conflict with standard executables, which must also be provided, and provide a separate

PHP

This program contains third-party code from Henry Spencer and the PHP Group. Under the terms of the Henry Spencer copyright notice and the PHP license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the PHP software, and the terms contained in the following notices do not change those rights.

The PHP License

SAXPath

This program contains third-party code from SAXPath. Under the terms of the SAXPath license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the SAXPath software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the SAXPath software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or SAXPath.

The SAXPath License

Sun Microsystems, Inc.

The Java Logo

Java™
COMPATIBLE
ENTERPRISE EDITION

W3C DOM

The W3C License

Index