Oracle9*i*AS Containers for J2EE

Services Guide,

Release 2 (9.0.3)

August 2002

Part No. A97690-01

ORACLE

Oracle9*i*AS Containers for J2EE Services Guide, Release 2 (9.0.3)

Part No. A97690-01

Contributing Authors: Elizabeth Hanes Perry, Janis Greenberg, and Mark Kennedy

Contributors: Ashok Banerjee, Ellen Barnes, Rachel Chan, Gary Gilchrist, Min-Hank Ho, Sunil Kunisetty, Stella Li, Sastry Malladi, Sheryl Maring, Raymond Ng, Thomas Van Raalte, Mike Sanko, Anirruddha Thakur, Brian Wright, Irene Zhang

Contents

3 Overview of JAAS in Oracle9*i*AS

4 Quick Start JAAS Provider Demo

5 Integrating the JAAS Provider with Java2 Applications

6 Managing the JAAS Provider

7 Developing Secure J2SE Applications

8 Developing Secure J2EE Applications

9 Java Message Service

10 Interoperability and RMI Tunneling

11 Data Sources

12 Java Transaction API

13 J2EE Connector Architecture

14 Working with Java Object Cache

15 Oracle HTTPS for Client Connections

A JAAS Provider APIs

List of Examples

List of Figures

List of Tables

Send Us Your Comments

Oracle9*i*AS Containers for J2EE Services Guide, Release 2 (9.0.3)

Part No. A97690-01

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

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

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

- Electronic mail: jpgreader_us@oracle.com
- FAX: (650) 506-7225 Attn: Java Platform Group, Information Development Manager
- Postal service:
  Oracle Corporation
  Java Platform Group, Information Development Manager
  500 Oracle Parkway, Mailstop 4op9
  Redwood Shores, CA 94065
  USA

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

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

Preface

This Services Guide describes the services provided by Oracle9*i*AS Containers for J2EE (OC4J).

This preface contains these topics:

Intended Audience

This book was written for developers familiar with the J2EE architecture who want to understand Oracle's implementation of J2EE Services.

Documentation Accessibility

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

visit the Oracle Accessibility Program Web site at `http://www.oracle.com/accessibility/`.

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

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

Structure

This book contains the following chapters and appendices:

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

Chapter 2, "Java Naming And Directory Interface"—Discusses using the JNDI to look up objects.

Chapter 3, "Overview of JAAS in Oracle9iAS"—Introduces Oracle's explanation of the Java Authentication and Authorization Service.

Chapter 4, "Quick Start JAAS Provider Demo"—Demonstrates how to configure and start a JAAS-based application, as illustrated by the `CallerInfo` example.

Chapter 5, "Integrating the JAAS Provider with Java2 Applications"—Discusses using the JAAS provider from Java-based applications.

Chapter 6, "Managing the JAAS Provider"—Discusses using Oracle Enterprise Manager to configure and run the JAAS provider.

Chapter 7, "Developing Secure J2SE Applications"—Describes how to use JAAS for authentication and authorization in a J2SE environment.

Chapter 8, "Developing Secure J2EE Applications"—Describes how to use JAAS for authentication and authorization in a J2EE environment.

Chapter 9, "Java Message Service"—Discusses plugging Resource Providers into the JMS.

Chapter 10, "Interoperability and RMI Tunneling"—Discusses OC4J support for EJB2.0 interoperation using RMI/IIOP and other technologies.

Chapter 11, "Data Sources"—Discusses data sources, a higher-level abstraction of a database connection or other source of information.

Chapter 12, "Java Transaction API"—Discusses Oracle's implementation of the JTA.

Chapter 13, "J2EE Connector Architecture"— Describes how to use the J2EE Connector Architecture in an OC4J application.

Chapter 14, "Working with Java Object Cache"—Describes the OC4J Java Object Cache, including its architecture and programming features.

Chapter 15, "Oracle HTTPS for Client Connections"—Describes using HTTPS for secure communications.

Chapter A, "JAAS Provider APIs"—Describes the JAAS Provider public packages.

Chapter B, "JAAS Provider Standards and Samples"—Provides sample JAAS Provider code.

Related Documents

See the following additional OC4J documents available from the Oracle Java Platform group:

- *Oracle9iAS Containers for J2EE User's Guide*

  This book presents an overview and general information for OC4J; primer chapters for servlets, JSP pages, and EJBs; and general configuration and deployment instructions.

- *Oracle9iAS Containers for J2EE Support for JavaServer Pages Reference*

  This book provides information for JSP developers who want to run their pages in OC4J. It includes a general overview of JSP standards and programming considerations, as well as discussion of Oracle value-added features and steps for getting started in the OC4J environment.

- *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*

  This book provides conceptual information and detailed syntax and usage information for tag libraries, JavaBeans, and other Java utilities provided with OC4J.

- *Oracle9iAS Containers for J2EE Servlet Developer's Guide*

  This book provides information for servlet developers regarding use of servlets and the servlet container in OC4J. It also documents relevant OC4J configuration files.

- *Oracle9iAS Containers for J2EE Enterprise JavaBeans Developer's Guide and Reference*

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

Also available from the Oracle Java Platform group:

- *Oracle9i JDBC Developer's Guide and Reference*
- *Oracle9i SQLJ Developer's Guide and Reference*
- *Oracle9i JPublisher User's Guide*
- *Oracle9i Java Stored Procedures Developer's Guide*

The following documents are available from the Oracle9*i* Application Server group:

- *Oracle9i Application Server Administrator's Guide*
- *Oracle Enterprise Manager Administrator's Guide*
- *Oracle HTTP Server Administration Guide*
- *Oracle9i Application Server Performance Guide*
- *Oracle9i Application Server Globalization Support Guide*
- *Oracle Web Cache Administration and Deployment Guide*
- *Oracle9i Application Server: Migrating from Oracle9i Application Server 1.x*

The following are available from the JDeveloper group:

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

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

The following documents from the Oracle Server Technologies group may also contain information of interest:

- *Oracle9i Application Developer's Guide - XML*
- *Oracle9i Application Developer's Guide - Fundamentals*

- *Oracle9i Supplied Java Packages Reference*
- *Oracle9i Supplied PL/SQL Packages and Types Reference*
- *PL/SQL User's Guide and Reference*
- *Oracle9i SQL Reference*
- *Oracle Net Services Administrator's Guide*
- *Oracle Advanced Security Administrator's Guide*
- *Oracle9i Database Reference*
- *Oracle9i Database Error Messages*

For information about Oracle9*i*AS Personalization, which is the foundation of the Personalization tag library, refer to the following documents from the Oracle9*i*AS Personalization group:

- *Oracle9iAS Personalization Administrator's Guide*
- *Oracle9iAS Personalization Recommendation Engine API Programmer's Guide*

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

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

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

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

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

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

`http://otn.oracle.com/admin/account/membership.html`

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

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

The following Oracle Technology Network (OTN) resources are available for further information about OC4J:

- OTN Web site for OC4J:

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

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

  `http://www.oracle.com/forums/forum.jsp?id=486963`

Conventions

This book generally uses UNIX syntax for file paths and shell variables. In most cases file names and directory names are the same for Windows NT, unless otherwise noted. The notation `$ORACLE_HOME` indicates the full path of the Oracle home directory. It is equivalent functionally to the Windows NT environment variable `%ORACLE_HOME%`, though of course the Oracle installation paths are different between NT and UNIX.

The following conventions are used in this manual:

Convention	Meaning
. . .	Vertical ellipsis points in an example mean that information not directly related to the example has been omitted.
. . .	Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted
italicized regular text	Italicized regular text is used for emphasis or to indicate a term that is being defined or will be defined shortly.
< >	Angle brackets enclose user-supplied names.
`code text`	Code text (Courier font) within regular text indicates class names, object names, method names, variable names, Java types, Oracle data types, file names, URL or URI fragments, and directory names.
%	At the beginning of a command, indicates an operating system shell prompt.
$	At the beginning of a command, indicates an Oracle JVM session shell prompt.
SQL>	At the beginning of a command, indicates a SQL*Plus prompt.

1

Introduction

Oracle9*i*AS Containers for J2EE (OC4J) supports the following technologies, each of which has its own chapter(s) in this book:

- Java Naming and Directory Interface (JNDI)
- Java Authentication and Authorization Service (JAAS)
- Java Message Service (JMS)
- J2EE Interoperability and Remote Method Invocation (RMI)
- Data Sources
- Java Transaction API (JTA)
- Java Connector Architecture
- Java Object Cache
- HTTPS

The remainder of this chapter gives a brief overview of each technology in the above list.

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

Java Naming and Directory Interface (JNDI)

JNDI 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. For details, see Chapter 2, "Java Naming And Directory Interface".

Java Authentication and Authorization Service (JAAS)

JAAS enables applications to authenticate and enforce access control. Oracle9*i*AS supports JAAS by implementing a JAAS provider. The JAAS provider provides application developers with user authentication, authorization, and delegation services to integrate into their application environments. Instead of devoting resources to developing these services, application developers can focus on the presentation and business logic of their applications.

For information about the Oracle implementation, see Chapter 3, "Overview of JAAS in Oracle9iAS", Chapter 4, "Quick Start JAAS Provider Demo", Chapter 5, "Integrating the JAAS Provider with Java2 Applications", Chapter 6, "Managing the JAAS Provider", Chapter 7, "Developing Secure J2SE Applications", Chapter 8, "Developing Secure J2EE Applications", Appendix A, "JAAS Provider APIs" and Appendix B, "JAAS Provider Standards and Samples".

Java Message Service (JMS)

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. For details, see Chapter 9, "Java Message Service".

J2EE Interoperability and Remote Method Invocation (RMI)

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.Version 2.0 of the Enterprise Java Beans specification uses RMI over the IIOP protocol to make it easy for EJB-based applications to invoke one another across different containers. You can make your existing EJB interoperable without changing a line of code: simply edit the bean's properties and redeploy

J2EE uses RMI to provide interoperability between EJB running on different containers. In addition, OC4J supports invoking RMI over HTTP, a technique known as "RMI tunneling." For details, see Chapter 10, "Interoperability and RMI Tunneling".

Data Sources

A data source, which is the instantiation of an object that implements the *javax.sql.DataSource* interface, enables you to retrieve a connection to a database server. For details, see Chapter 11, "Data Sources".

Java Transaction API (JTA)

JTA supplies a standard interface to support communications among the parties to a distributed transaction. These parties include the resource manager, the application server, and the transactional applications. For details, see Chapter 12, "Java Transaction API".

Java Connector Architecture

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

For details, see Chapter 13, "J2EE Connector Architecture".

Java Object Cache

The Java Object Cache (formerly OCS4J) is a set of Java classes designed to manage Java objects within a process, across processes, and on local disk. The primary goal of the Java Object Cache is to provide a powerful, flexible, easy to use service that will significantly improve 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.

For details, see Chapter 14, "Working with Java Object Cache".

HTTPS

HTTPS is vital to securing client-server interactions. Java applications that act as a clients, such as servlets that initiate connections to other Web servers, need their own HTTPS implementation to make requests and to receive information securely from the server. Java application developers who are familiar with the HTTP package, `HTTPClient`, or the Sun Microsystems, Inc., `java.net` package can easily use Oracle HTTPS to secure client interactions with a server. For details, see Chapter 15, "Oracle HTTPS for Client Connections".

2

Java Naming And Directory Interface

This chapter describes the Java Naming and Directory Interface (JNDI) service implemented by Oracle9*i*AS Containers for J2EE (OC4J) applications. It covers the following topics:

Introduction

JNDI, part of the J2EE specification, and provides naming and directory functionality for Java applications. Because JNDI is defined independently of any specific naming or directory service implementation, it enables Java applications to access different, 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.

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

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

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

Initial Context

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

1. Creating a new `InitialContext` object (in the `javax.naming` package).
2. Using the `InitialContext`, *looking up* a J2EE or other resource.

When OC4J starts up, it constructs a JNDI initial context *for each application* by reading each of the application's configuration XML files that can contain resource references. Applications are defined in the `server.xml` configuration file.

> **Note:** After the initial configuration, the JNDI tree for each application is purely memory-based. Additions made to the context are not persisted. When OC4J is restarted, any new bindings made in application code are no longer available.

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

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

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

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

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

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

Constructing a JNDI Context

When OC4J starts up, it constructs a JNDI context for each application deployed in the server (in `server.xml`). 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-written applications inherit properties from the global application. User-written applications can override property values defined in the global application, define new values for properties, and define new properties as required.

In the default OC4J server, as shipped, the global application is the *default application*, as defined in `server.xml`. For more information about configuring the OC4J server and its contained applications, see the *Oracle9iAS Containers for J2EE User's Guide*, in particular the "Advanced Information" chapter.

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

- System property values, as set either by the OC4J server or possibly by the application container.
- A `jndi.properties` file contained in the application EAR file (as part of `application-client.jar`).
- An environment specified explicitly in a `Hashtable` passed to the JNDI initial context constructor.

The JNDI Environment

The JNDI `InitialContext` has two constructors:

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

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

The second constructor takes an environment parameter. The second form of the `InitialContext` constructor is normally used in client applications, where it is necessary to specify the JNDI environment. The `env` parameter in this constructor is a `Hashtable` that contains properties required by JNDI. These properties, defined in the `javax.naming.Context` interface, are listed in Table 2–1.

Table 2–1 InitialContext Properties

Property	Meaning
`INITIAL_CONTEXT_FACTORY`	Value for the `java.naming.factory.initial` property; this property specifies which initial context factory to use when creating a new initial context object.
`PROVIDER_URL`	Value for the `java.naming.provider.url` property; this property specifies the URL that the application client code uses to look up objects on the server. Also used by the `RMIInitialContextFactory` to search for objects in different applications.
`SECURITY_PRINCIPAL`	Value for the `java.naming.security.principal` property; this property specifies the user name. Required in application client code to authenticate the client. Not required for server-side code, because the authentication has already been done.
`SECURITY_CREDENTIAL`	Value for the `java.naming.security.credential` property; this property specifies the password. Required in application client code to authenticate the client. Not required for server-side code, because the authentication has already been done.

See "Remote Client Example" on page 2-10 for a code example that sets these properties and gets a new JNDI initial context.

Initial Context Factories

The three JNDI initial context factories available for use by application code. They are

- `ApplicationClientInitialContextFactory`
- `ApplicationInitialContextFactory`
- `RMIInitialContextFactory`

The following sections describe each of these factories and their uses in OC4J applications.

ApplicationClientInitialContextFactory

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

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

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

When clients use the `ApplicationClientInitialContextFactory` to construct JNDI initial contexts, they can look up local objects (objects contained in the immediate application, or in its parent application) using the `java:comp/env` mechanism, and can use ORMI to look up remote objects.

Environment Properties

`ApplicationClientInitialContextFactory` invokes `RMIInitialContextFactory` to read the properties listed in Table 2–2 from the environment.

Table 2–2 JNDI-Related Environment Properties

Property	Meaning
`dedicated.connection`	Each JNDI lookup retrieves a connection to the server. Each subsequent JNDI lookup for this same server uses the connection returned by the first JNDI lookup. That is, all requests are forwarded over and share the same connection. The `dedicated.connection` JNDI property overrides this default behavior. If you set `dedicated.connection` to `true` before you retrieve an `InitialContext`, you will retrieve a separate physical connection for each lookup, each with its own designated username/password. `dedicated.connection` defaults to `false`. Reset to `true` if: 1. You want to connect using a different username/password each time. ORMI connections are associated with an authenticated ID; setting this property to true will open a new connection instead of reusing a cached connection. If this property is set to false, the first username/password is used for all subsequent connections, even when an alternate username/password is supplied. 2. You want to make a remote connection and look up an object on the remote connection before looking up the same object locally.
`java.naming.provider.url`	The URL to use when looking for local or remote objects. The format is either `[http: | https:]ormi://`*`hostname/appname`* or `corbaname:hostname:port`. For details on the `corbaname` URL, see "The corbaname URL" on page 10-4. Multiple hosts (for failover) can be supplied in a comma-separated list.
`http.tunnel.path`	Specifies an alternative `RMIHttpTunnelServlet` path. The default path is `/servlet/rmi`, as bound to the target site's Web application.
`Context.SECURITY_PRINCIPAL`	The user name. Required in client-side code to authenticate the client. Not required for server-side code because authentication has already been done.

Table 2–2 JNDI-Related Environment Properties

Property	Meaning
`Context.SECURITY_CREDENTIAL`	The password. Required in client-side code to authenticate the client. Not required for server-side code because authentication has already been done.

Remote Client Example

The following example code shows how JNDI properties can be specified in a client application:

```
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
          "com.evermind.server.ApplicationClientInitialContextFactory");
env.put(Context.PROVIDER_URL, "ormi://<hostname>/employee");
env.put(Context.SECURITY_PRINCIPAL, "admin");
env.put(Context.SECURITY_CREDENTIALS, "welcome");

Context context = new InitialContext(env);
//do the lookups...
...
```

Server-Side Clients

Server-side clients need not specify an `InitialContextFactory` to look up resources defined within the client application. By default, server-side clients have `InitialContextFactory` set to `ApplicationInitialContextFactory`. This allows clients to perform lookups using names in the style `java:comp:/env`.

To look up resources that are not defined within the client application, clients must set the `InitialContextFactory` to `RMIInitialContextFactory` and look up the resources or EJB using an explicit URL.

ApplicationInitialContextFactory

When code is running in a server, it is, by definition, part of an application. Therefore, as part of an application, OC4J can establish defaults for properties that JNDI uses. For the `java.naming.factory.initial` property, OC4J sets `ApplicationInitialContextFactory` in the `com.evermind.server` package as the default value for this system property.

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

However, when you use the default `ApplicationInitialContextFactory`, only application-local resources are available using the `java:comp/env` lookup mechanism. If your application needs to look up a remote reference, either a resource in another J2EE application or perhaps a resource external to any J2EE application, then you must use `RMIInitialContextFactory`.

Example

As a concrete example, consider a servlet that needs to get a data source to perform a JDBC operation on a database. The data source reference is mapped in `orion-web.xml` as:

```
<resource-ref-mapping name="jdbc/OracleDS1" location="jdbc/pool/OracleCache" />
```

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

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

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/OracleDS1");
  ...
}
catch (NamingException ne) {
  throw new ServletException(ne);
}
...
```

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

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

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

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

RMIInitialContextFactory

Occasions arise for use of the `RMIInitialContextFactory` property in the `com.evermind.server.rmi` package. Using either the default server-side `ApplicationInitialContextFactory` or specifying `ApplicationClientInitialContextFactory` works for most application purposes.

In some cases, however, an additional context factory must be used:

1. When looking up an object that is part of another J2EE application, and for which a resource reference either cannot be or is not specified in the current application's `application-client.xml` file.
2. When performing a general lookup for external JNDI objects, that may or may not be part of a J2EE application. A generalized JNDI object browser is an example of this usage.
3. When accessing the entire remote JNDI namespace, in contrast to a specific application context. For further details, see:

   `http://www.orionserver.com/docs/remote-access/remote-access.xml`

The RMIInitialContextFactory uses the same environment properties used by ApplicationClientInitialContextFactory—namely:

- dedicated.connection
- java.naming.provider.url
- http.tunnel.path
- SECURITY_PRINCIPAL
- SECURITY_CREDENTIALS

Remote Client Example

You can use the following code to look up a remote object using RMIInitialContextFactory:

```
Hashtable env = new Hashtable();
env.put("java.naming.factory.initial",
    "com.evermind.server.rmi.RMIInitialContextFactory");
env.put("java.naming.provider.url","ormi://localhost/ejbsamples");
env.put("java.naming.security.principal","admin");
env.put("java.naming.security.credentials","welcome");
Context context = new InitialContext(env);
/**
* Lookup the Cart home object. The reference should be retrieved from the
* application-local context (java:comp/env, the variable is
* specified in the assembly descriptor; META-INF/application-client.xml)
* but for simplicity this example uses a global variable.
*/
System.out.println("Context = " + context);

Object homeObject = context.lookup("MyCart");
Hashtable env1 = new Hashtable();
env1.put("java.naming.factory.initial",
    "com.evermind.server.rmi.RMIInitialContextFactory");
env1.put("java.naming.provider.url","ormi://localhost/ejbsamples1");
env1.put("java.naming.security.principal","admin");
env1.put("java.naming.security.credentials","welcome");
Context context1 = new InitialContext(env1);
Object homeObject1 = context1.lookup("MyProduct");
System.out.println("HomeObject1 = " + homeObject1);
```

3

Overview of JAAS in Oracle9*i*AS

This chapter introduces support for Java Authentication and Authorization (JAAS), in Oracle9*i*AS Containers for J2EE (OC4J). JAAS enables application developers to integrate authentication, authorization, and delegation services with their applications.

This chapter contains these topics:

> **Note:** Chapter 7 of the *Oracle9i Application Server Security Guide* also contains important information about configuring JAAS.

JAAS Support

JAAS is a Java package which enables applications to authenticate and enforce access control.

Oracle9*i*AS supports JAAS by implementing a JAAS provider. The JAAS provider provides application developers with user authentication, authorization, and delegation services to integrate into their application environments. Instead of

devoting resources to developing these services, application developers can focus on the presentation and business logic of their applications.

Note: Some class and component names contain the word "JAZN", which is the internal code name for "JAAS provider".

What Are Authentication, Authorization, and Delegation?

Authentication is the process of verifying the identity of a user, device, or other entity in a computer system, often as a prerequisite to granting this entity access to resources in a system. For example, when a user enters a username and password to access resources on a computer, such as a database, the user must first be authenticated (verified) by means of the login information before being permitted access to these resources.

Once a user's username and password have been authenticated, the authorization process occurs. Authorization is the process of determining the following for the authenticated user: Who has the right to perform an operation on an object (such as updating a table in a database)?

Delegation provides support for impersonation of a specified user. An application can be configured to run with the permissions associated with a specified user by means of the `run-as` element.

Foundations of the JAAS Provider

The JAAS framework and the Java2 Security model form the foundation of the JAAS provider. That is, the JAAS provider implements JAAS and integrates with J2SE and J2EE applications that use the Java2 Security model.

JAAS

The JAAS provider implements support for JAAS policies. Policies contain the rules (permissions) that authorize a user to use resources, such as reading a file. JAAS enables services to authenticate and enforce access control upon users of these resources.

Java2 Security Model

The JAAS provider integrates with J2SE and J2EE applications that use the Java2 Security Model. Unlike the original Java security model, under Java2 security, many levels of restrictions can be configured.

See Also:

- "What Is JAAS?" on page 3-7
- "What Is the Java2 Security Model?" on page 3-4

Java Application Environments

Developers can easily integrate the JAAS provider with these applications for quick development and deployment:

- Standalone Java applications in Java2 Platform, Standard Edition (J2SE) environments
- Web-based applications in Java2 Platform, Enterprise Edition (J2EE)

See Also:

"Integrating the JAAS Provider with Basic Authentication" on page 5-12 for additional information on the J2SE and J2EE environments.

Provider Types

The JAAS provider supports two types of repository providers, referred to as provider types.

These provider types are repositories for secure, centralized storage, retrieval, and administration of provider data. This data consists of realm (users and roles) and JAAS policy (permissions) information.

Use the provider type appropriate to your environment.

LDAP-Based Provider Type

The LDAP-based provider type is based on the Lightweight Directory Access Protocol (LDAP) for centralized storage of information in a directory. The Oracle9*i*AS JAAS Provider uses the LDAP-based Oracle Internet Directory.

Use this provider type if you are using Oracle9*i*AS and Oracle Internet Directory.

XML-Based Provider Type

The XML-based provider type is used for lightweight storage of information in XML files.

Use this provider type if you are using an XML file, such as `jazn-data.xml`, to store your user and realm information.

Note: Don't confuse the XML-based provider type with XML files in general. XML files are used as property and configuration files in both LDAP-based and XML-based provider types or environments. If an XML file such as `jazn-data.xml` is used to store realm and user information, then the provider type is called XML-based.

See Also:

"JAAS Provider Realm and Policy Management" on page 3-16

What Is the Java2 Security Model?

Sun's Java2 Security Model is fundamental to the JAAS provider.

The Java2 Security Model enables configuration of security at all levels of restriction. This provides developers and administrators with increased control over many aspects of enterprise applet, component, servlet, and application security.

The Java2 Security Model is capability-based and enables you to establish protection domains, and set security policies for these domains. When the JAAS provider is integrated with applications developed for the J2SE or J2EE environments, these environments use the Java2 Security Model to different degrees.

Permissions are the basis of the Java2 Security Model. All Java classes (whether run locally or downloaded remotely) are subject to a configured security policy that defines the set of permissions available for those classes. Each permission represents a specific access to a particular resource. Table 3–1 identifies the elements that comprise a Java permission instance.

Table 3–1 Java Permission Instance Elements

Element	Description	Example
Class name	The permission class	`java.io.FilePermission`
Target	The **target name** (resource) to which this permission applies	Directory `/home/*`
Actions	The **actions** associated with this target	Read, write, and execute permissions on directory `/home/*`

Each Java class, when loaded, is associated with a protection domain. Protection domains can be configured for all levels of restriction (from complete restriction on resources to full access to all resources). Each protection domain is assigned a group of permissions based on a configured security policy at Java virtual machine (JVM) startup.

At runtime, the authorization check is done by stack introspection. This consists of reviewing the runtime stack and checking permissions based on the protection domains associated with the classes on the stack. This is typically triggered by a call to either:

- `SecurityManager.checkPermission()`
- `AccessController.checkPermission()`

The permission set in effect is defined as the intersection of all permission sets assigned to protection domains at the moment of the security check.

Figure 3–1 shows the basic model for authorization checking at runtime.

Figure 3–1 Java2 Security Model

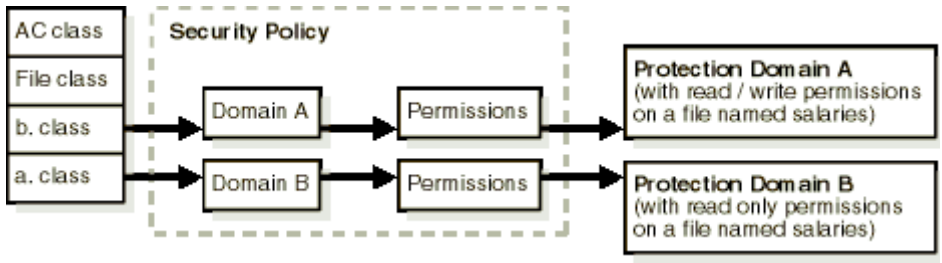


Table 3–2 lists the permission classes provided by the JAAS provider that enables you to enforce access upon users of resources.

Table 3–2 JAAS Provider Permission Classes

Permission	Part of Package...	Description	See Also...
`AdminPermission`	`oracle.security.jazn.policy`	Represents the right to administer a permission (that is, grant or revoke another user's permission assignment)	"AdminPermission" page A-6 for specific syntax examples
`RoleAdminPermission`	`oracle.security.jazn.policy`	The grantee of this permission is granted the right to further grant / revoke the target role.	"AdminPermission" page A-6
`JAZNPermission`	`oracle.security.jazn`	For authorization permissions. `JAZNPermission` contains a name (also called a target name), but no actions list; you either have or do not have the named permission.	"JAZNPermission" o page A-3 for a list of target names for `JAZNPermission`, what the permission allow, and the risks granting the permiss
`RealmPermission`	`oracle.security.jazn.realm`	Represents permission actions for a realm (such as `createRealm`, `dropRealm`, and so on). `RealmPermission` extends from `java.security.Permission`, and is used like any regular Java permission.	"RealmPermission" page A-9 for a list of permission actions

See Also:

- "JAAS Provider Integration in J2SE Application Environments" on page 5-2
- "JAAS Provider Integration in J2EE Application Environments" on page 5-3
- Chapter 6, "Managing the JAAS Provider"
- Sun Java documentation by visiting the following URL:

  `http://java.sun.com/security/`

What Is JAAS?

The JAAS interface is implemented by the JAAS provider. JAAS is a Java package that enables applications to authenticate and enforce access controls upon users.

JAAS is designed to complement the existing code-based security in JDK 1.3. JAAS implements a Java version of the standard Pluggable Authentication Module (PAM) framework. This enables an application to remain independent from the authentication service.

JAAS extends the access control architecture of the Java2 Security Model to support principal-based authorization.

This section describes JAAS support for the following authorization, authentication, and user community (realm) features. Some of these features are fully supported in this release of JAAS, while others are not explicitly defined. The JAAS provider provides enhancements to some of these features.

- Principals
- Subjects
- Login Module Authentication
- Roles
- Realms
- Policies and Permissions

See Also:

- "JAAS Provider Realm and Policy Management" on page 3-16 for information on how the JAAS provider enhances JAAS to more explicitly define key authorization, authentication, and user community (realm) features
- JAAS documentation at the following Web site for more specific discussions of key JAAS features:

  `http://java.sun.com/products/jaas/`

Principals

A *principal* is a specific identity, such as a user named `frank` or a role named `hr`. A principal is associated with a subject upon successful authentication to a computing service.

A principal is represented by an instance of a concrete class that implements the `java.security.Principal` interface. Each class defines a namespace for its instances, within which each principal instance has a unique name. The name and class of a principal instance uniquely describes the instance.

For LDAP-based environments, an `X500Principal` class is defined that accepts the X.500 style name as the name of the principal.

Subjects

A *subject* represents a grouping of related information for a single user of a computing service, such as a person, computer, or process. Such information includes the subject's identities and security-related attributes (such as passwords and cryptographic keys).

Subjects can have multiple identities, where principals represent identities in the subject. A subject becomes associated with a principal (user `frank`) upon successful authentication to a computing service, that is, the subject provides evidence (such as a password) to prove its identity.

Principals bind names to a subject. For example, a person subject, user `frank`, may have two principals:

- One binds the principal `frank doe` (name on his driver license) to the subject
- Another binds the identification principal `999-99-9999` (number on his student identification card) to the subject

Both principals refer to the same subject.

Subjects can also own security-related attributes (known as credentials). Sensitive credentials requiring special protection, such as private cryptographic keys, are stored in a private credential set. Credentials intended to be shared, such as public key certificates or Kerberos server tickets are stored in a public credential set. Different permissions are required to access and modify different credential sets.

Subjects are represented by the `javax.security.auth.Subject` class.

To perform work as a particular subject, an application invokes the method `Subject.doAs(Subject, PrivilegedAction)` (or one of its variations). This method associates the subject with the current thread's `AccessControlContext`, and then executes the specified request.

Login Module Authentication

To associate a principal (such as `frank`) with a subject, a client attempts to log into an application. In login module authentication, the `LoginContext` class provides the basic methods used to authenticate subjects such as users, roles, or computing services. The `LoginContext` class consults configuration settings to determine whether the authentication modules (known as login modules) are configured for use with the particular application that the subject is attempting to access. Different login modules can be configured with different applications.

Since the `LoginContext` separates the application code from the authentication services, a different login module can be plugged in under an application without affecting the application code.

Actual authentication occurs with the method `LoginContext.login()`. If authentication succeeds, the authenticated subject can be retrieved by invoking `LoginContext.getSubject()`. The real authentication process can involve multiple login modules. JAAS defines a two-phase authentication process to coordinate the login modules configured for an application.

After retrieving the subject from the `LoginContext`, the application then performs work as the subject by invoking `Subject.doAs()`.

See Also:

- "Authentication in the J2SE Environment" on page 7-2
- "Authentication in the J2EE Environment" on page 8-2

Roles

JAAS does not explicitly define roles or groups. Instead, roles or groups are implemented as concrete classes that use interface `java.security.Principal`.

JAAS does not define how to support the RBAC role hierarchy (granting a role to a role). The Sun provider of `javax.security.auth.Policy` recognizes a special type of principal, as defined by the `PrincipalComparator` interface. However, `PrincipalComparator` is not fully integrated with the JAAS provider, and is therefore not supported.

For LDAP-based environments, an `X500GroupPrincipal` class is defined that accepts an X.500 style name as the name of the group.

Realms

JAAS does not explicitly define user communities. However, the J2EE reference implementation (RI) defines a similar concept of user communities called realms. A realm provides access to users and roles (groups) and optionally provides administrative functionality. A user community instance is essentially a realm that is maintained internally by the authorization system. The J2EE RI Realm API supports user-defined realms through subclassing. The J2EE RI Realm API, however, is:

- Not as fully developed as the JAAS provider realm framework
- Not being proposed as a standard
- Expected to undergo further changes to be integrated with JAAS

See Also:

- "JAAS Provider Realm Framework" on page 3-18 for JAAS provider enhancements to realms
- "XML-Based Realm and Policy Information Storage" on page 3-25

Applications

JAAS does not explicitly define an application or subsystem for partitioning authorization rules. However, JAAS meets many of the requirements for the subsystem concept. For example, JAAS defines the notion of a codebase (plus a signer) as the target and grantee of a grant statement. This enables permissions to be granted application-specific code. The Java notion of namespace partitioning through packages also allows for partitioning of permission classes in an application-specific manner.

Policies and Permissions

A policy is a repository of JAAS authorization rules. The policy includes grants of permissions to principals, thus answering the question: given a grantee, what are the granted permissions of the grantee?

Policy information is supplied by the JAAS provider. JAAS does not define an administrative API for policy administration. The administrative API is implementation specific.

Table 3–3 describes Sun's implementation of policy file parameters.

Table 3–3 Policy File Parameters

Where...	Is Defined As...	Example
subject	one or more principal(s)	`duke`
codesource	*`codebase`*, *`signer`*	`http://www.foo.com, foo`

File-Based Policy Example

The following example shows a typical entry in the JAAS policy file as implemented by Sun's implementation of the JAAS file-based policy provider:

```
 grant   CodeBase "http://www.foo.com",
        Principal com.sun.security.auth.SolarisPrincipal "duke"
{
        permission java.io.FilePermission "/home/duke", "read, write";
};
```

Code from `www.foo.com`, signed by `foo`, and running as a `SolarisPrincipal` with the username `duke`, has the permission that permits the executing code to read and write files in `/home/duke`.

XML-Based Example

The JAAS provider also provides an XML file to store policy information. In the following example, a segment of the `jazn-data.xml` file grants the `jazn.com /administrators` various permissions:

```
<!--JAZN Policy Data -->
<jazn-policy>
  <grant>
     <grantee>
        <principals>
            <principal>
                <realm>jazn.com/realm>
                <type>role/type>
                <class>oracle.security.jazn.spi.xml.XMLRealmRole
                     </class>
                <name>jazn.com/administrators/name>
             </principal>
         </principals>
     </grantee>
     <permissions>
```

```
            <permission>
               <class>oracle.security.jazn.policy.AdminPermission</class>
               <name>oracle.security.jazn.realm.
                   RealmPermission$jazn.com$modifyrealmmetadata</name>
            </permission>
            <permission>
                <class>oracle.security.jazn.policy.AdminPermission</class>
                <name>oracle.security.jazn.realm.
                    RealmPermission$jazn.com$droprealm</name>
            </permission>
            <permission>
                <class>oracle.security.jazn.policy.AdminPermission</class>
                <name>oracle.security.jazn.realm.RealmPermission$jazn.
                    com$createrole</name>
            </permission>
            <permission>
                <class>oracle.security.jazn.realm.RealmPermission</class>
                <name>jazn.com</name>
                <actions>createrealm</actions>
              </permission>
         </permissions>
      </grant>
    </jazn-policy>
```

See Also:

- "Sample jazn-data.xml Code" on page B-2 to view a complete `jazn-data.xml` file.
- "JAAS Provider Policy Administration" on page 3-27 for information on JAAS provider enhancements to policies

JAAS Provider Features

Table 3–4 lists the JAAS features provided by Oracle9*i*AS.

Table 3–4 JAAS Provider Features

Feature	Description	See Also...
Realms	Realms provide access to user and role information. An Oracle Realm API package (`oracle.security.jazn.realm`) is provided to support user and role management. This API includes a `RealmPrincipal` interface that extends from `java.security.Principal` and associates a realm with users and roles	"Realms" on page 3-10 "JAAS Provider Realm Framework" on page 3-18
Role-based access control (RBAC)	Support is provided for secure, centralized, and customizable RBAC management	"Role-Based Access Control (RBAC)" on page 3-14
Login Module Authentication	■ Provides a `RealmLoginModule` class for non-SSO environments ■ Integrates with Oracle9*i*AS Single Sign-On (SSO) for SSO login authentication in J2EE application environments	Chapter 7, "Developing Secure J2SE Applications" Chapter 8, "Developing Secure J2EE Applications"
JAAS provider type management	Several methods for managing JAAS provider type information are available: ■ An Admintool command line tool that supports management of information in both provider types ■ An Oracle Enterprise Manager graphical user interface (GUI) tool that supports management of information in LDAP-based Oracle Internet Directory ■ Programmatic level management of both provider types	"JAAS Provider Policy Administration" on page 3-27 Chapter 6, "Managing the JAAS Provider"
JAZNUserManager	`JAZNUserManager` is an implementation of the OC4J `UserManager` that integrates with both LDAP-based and XML-based provider types.	"JAAS Provider Integration in J2SE Application Environments" on page 5-3 Chapter 8, "Developing Secure J2EE Applications"

JAAS Provider User Services

The Oracle9*i*AS implementation of JAAS provides these user services for application developers to integrate into their applications. This section describes several JAAS provider authorization features.

- Capability Model of Access Control
- Role-Based Access Control (RBAC)

Capability Model of Access Control

The **capability model** is essentially a method for organizing authorization information. The JAAS provider is based on the Java2 Security Model, which uses the capability model of access control to control access to permissions. With the capability model, authorization is associated with the principal (a user named `frank` in the following example). Table 3–5 shows the permissions that user `frank` is authorized to use:

Table 3–5 User Permissions

User	Has These File Permissions...
`frank`	Read and write permissions on a file named `salaries.txt` in the `/home/user` directory

When user `frank` logs in and is successfully authenticated, the permissions described in Table 3–5 are retrieved from the JAAS provider (whether the LDAP-based Oracle Internet Directory or XML-based provider type) and granted to user `frank`. User `frank` is then free to execute the actions permitted by these permissions.

See Also:

- "What Is the Java2 Security Model?" on page 3-4
- "Principals" on page 3-7
- "JAAS Provider Policy Administration" on page 3-27

Role-Based Access Control (RBAC)

RBAC enables you to assign permissions to roles. Users are then granted their permissions by being made members of appropriate roles. Support for RBAC is a key JAAS provider feature. This section describes the following RBAC features:

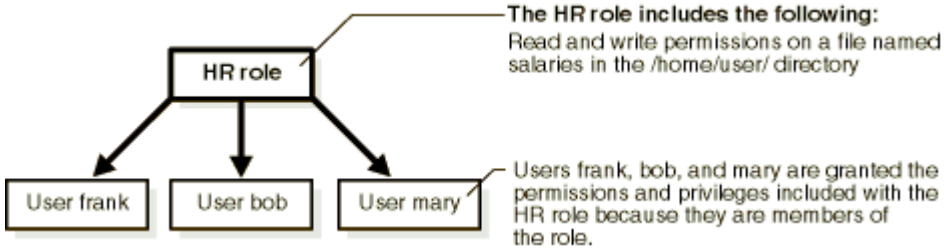
- Role Hierarchy
- Role Activation

Role Hierarchy

RBAC simplifies the management problems created by direct assignment of permissions to users. Assigning permissions directly to multiple users is potentially a major management task. If multiple users no longer require access to a specific permission, you must individually remove that permission from each user.

Instead of directly assigning permissions to users, permissions are assigned to a role, and users are granted their permissions by being made members of that role. Multiple roles can be granted to a user. A role can also be granted to another role, thus forming a **role hierarchy** that provides administrators with a tool to model enterprise security policies. Figure 3–2 provides an example.

Figure 3–2 Role-Based Access Control



When a user's responsibilities change (for example, through a promotion), the user's authorization information is easily updated by assigning a different role to the user instead of a massive update of access control lists containing entries for that individual user.

For example, if multiple users no longer require write permissions on a file named `salaries` in the `/home/user` directory, those privileges are removed from the `HR` role. All members of the `HR` role then have their permissions and privileges automatically updated.

Role Activation

A user is typically granted multiple roles. However, not all roles are enabled by default. The user can selectively enable the required roles to accomplish a specific task in a user session with the `run-as` security identity and `Subject.doAS()`. This

ensures the principle of least privilege. This way, the user is not enabling permissions or privileges unnecessary for the task. This limits the damage that can potentially result from an accident or error.

See Also: Sun Java documentation by visiting the following URL:

`http://java.sun.com/security/`

JAAS Provider Realm and Policy Management

The JAAS provider supports two types of repository providers, referred to as provider types:

- The LDAP-based provider type used with Oracle Internet Directory (OiD)
- The XML-based provider type used with an XML file, typically `jazn-data.xml`

OiD and `jazn-data.xml` are repositories used to store realm (users and roles) and policy (permissions) information. This section discusses the following topics in relation to the two different provider types:

- Realm and Policy Management Tools
- JAAS Provider Realm Framework
- JAAS Provider Policy Administration

Realm and Policy Management Tools

Several tools are provided for managing realm and policy information. Table 3–6 describes these tools and indicates the environment in which they operate.

Table 3–6 Realm and Policy Management Tools

Method/Environment	Description	See Also...
Oracle Enterprise Manager LDAP-based only	A graphical user interface tool that enables you to create principals (known as grantees) and assign permissions to these grantees.	"Using the Oracle Enterprise Manager Interface with the JAAS Provider" on page 6-3
JAZN Admintool Both LDAP and XML-based environments	A command line interface tool that enables administrators to create and manage users, realms, roles, and policies. The JAZN Admintool: ■ Uses the JAAS ProviderAPI packages described in Appendix A, "JAAS Provider APIs" to perform functions ■ Can be executed from the operating system command line The JAZN Admintool has the same capabilities and limitations as the JAAS Provider APIs. For example, you cannot create users with the JAZN Admintool if your provider type is LDAP-based Oracle Internet Directory. However, you can create users if your provider type is XML-based.	"Using the JAZN Admintool" on page 6-12

See Also:

- "What JAAS Provider Components Do You Need to Install?" in the *Oracle9i Application Server Installation Guide* for information on installing the provider type you want to use
- "Realms" on page 3-10
- "Package oracle.security.jazn.realm" on page A-7

JAAS Provider Realm Framework

The J2EE environment defines the concept of user communities. A user community instance is essentially a realm maintained internally by the authorization system.

The API package `oracle.security.jazn.realm` is provided to support realms. This API package is an enhancement to the JAAS policy provider.

Realms can be managed in both provider type environments:

- LDAP-based Oracle Internet Directory

  Provides for centralized storage of realms and JAAS policy in a directory

- XML-based

  Provide a lightweight form of storage for realms and JAAS policy

Realm Management in LDAP-Based Environments

A realm provides user and role management. An LDAP-based realm's data can be managed:

- Internally by creating and managing user information with the JAAS provider. See Chapter 6, "Managing the JAAS Provider".
- Externally by creating and managing user and role information with Oracle Internet Directory, and then integrating it with the JAAS provider.

LDAP-Based Realm Types

The JAAS provider supports three types of realms for LDAP-based environments. Each realm provides different user and role management capabilities. Table 3–7 describes these realms.

Table 3–7 Implementation of Realm Types

Realms Type	Description	Use This Realm...	See Also...
External Realm	■ Supports external, read-only user and role management ■ Integrates existing user communities with the JAAS provider	For non-hosting environments	Figure 3–3 on page 3-20 "Creating an External Realm" on page 6-26
Subscriber Realm	■ Created through provisioning tools ■ Used in hosting environments ■ Supports external, read-only user and role management	In a hosting environment (with subscriber-based customers) where multiple customers or companies subscribe to shared services	Figure 3–4 on page 3-21
Application Realm	■ Supports external, read-only user management ■ Supports internal roles management	If you want to use the JAAS provider role management feature	Figure 3–5 on page 3-22 "Creating an Application Realm" on page 6-28

Each realm type consists of:

- A role manager for role management
- A user manager for user management

User and role managers internally perform their duties (through JAAS provider permissions) or externally (through OiD Delegated Administration Service (DAS)).

> **Note:** The JAAS provider does not provide an internal user manager for creating users. Instead, you can create users with DAS or a command line tool such as `ldapadd`.

Figure 3–3 shows a sample LDAP directory information tree (DIT) containing an External Realm that is registered as an instance with the JAAS provider. The realm type is created below a Realms container.

Figure 3–3 Simplified Directory Information Tree for the External Realm

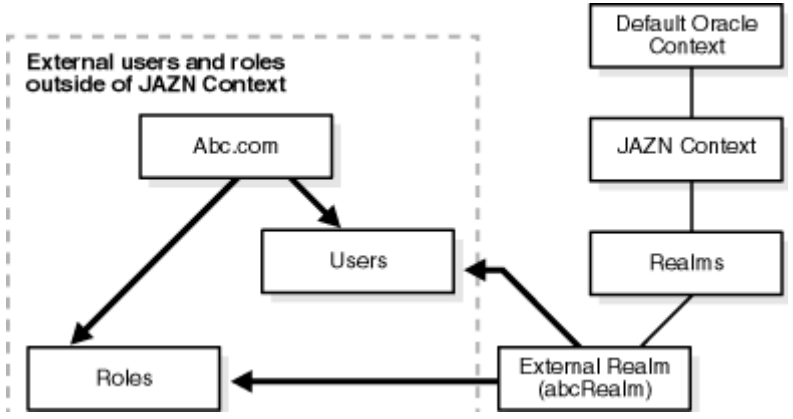


Table 3–8 describes the user and role management responsibilities of the External Realm.

Table 3–8 External Realm Responsibilities

External Realm Name	Role Management	User Management
`abcRealm`	Retrieves external, read-only roles	Retrieves external, read-only users

Figure 3–4 shows a sample LDAP directory information tree (DIT) containing a Subscriber Realm that is registered as an instance with the JAAS provider. The realm type is created below a Realms container.

Figure 3–4 Simplified Directory Information Tree for the Subscriber Realm

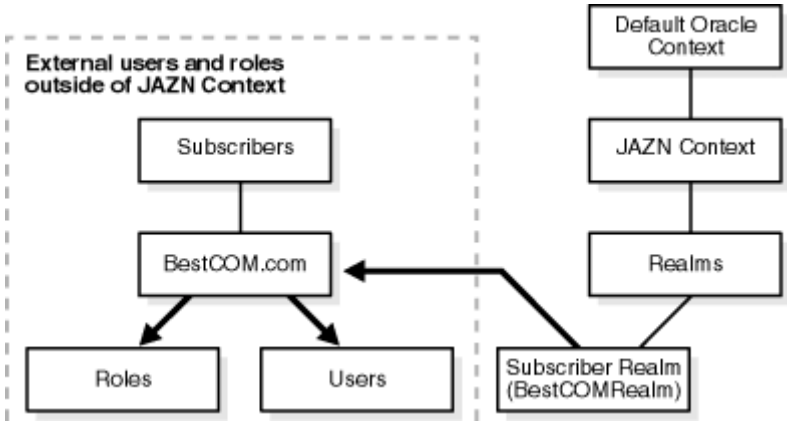


Table 3–9 describes the user and role management responsibilities of the Subscriber Realm.

Table 3–9 Subscriber Realm Responsibilities

Subscriber Realm Name	Role Management	User Management
`BestCOMRealm`	Retrieves external, read-only roles of a subscriber	Retrieves external, read-only users of a subscriber

Figure 3–5 shows a sample LDAP directory information tree (DIT) containing an Application Realm that is registered as an instance with the JAAS provider. The realm type is created below a Realms container.

Figure 3–5 Simplified Directory Information Tree for the Application Realm

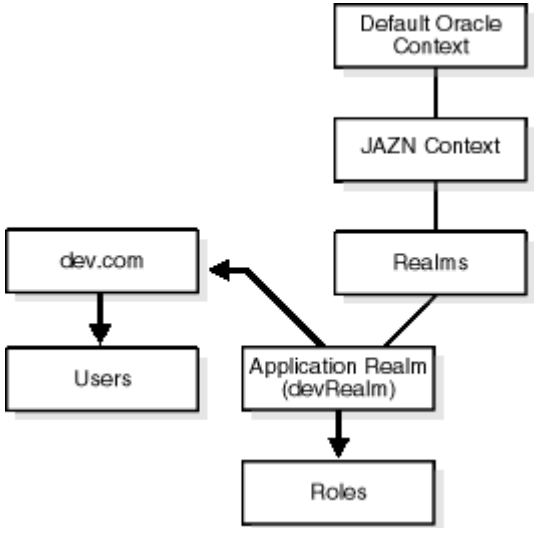


Table 3–10 describes the user and role management responsibilities of the Application Realm.

Table 3–10 Application Realm Responsibilities

Application Realm Name	Role Management	User Management
`devRealm`	Internally creates and manages modifiable roles	Retrieves external, read-only users

LDAP-Based Realm Data Storage

The realm framework provides a means for registering realm instances with the JAAS Provider and managing their information.

A Realms container object is created under the site-wide JAAS context. (For example, see the Realms container in Figure 3–3 on page 3-20.) For each registered realm instance, a corresponding realm entry is created under the Realms container that stores the realm's attributes. This directory hierarchy is known to the JAAS

provider, which enables the JAAS provider to create new realm instances in the desirable directory location and find all the registered realms in runtime.

For example, the distinguished name (DN) for a realm called `oracle` can be `"cn=oracle,cn=realms,cn=JAZNContext,cn=site root"`.

Upon successful installation of the JAAS provider, a default realm (External Realm) instance is installed. Predefined realm properties are configured for starting the default realm. Any realm type must provide concrete implementations for the system defined Java interfaces `UserManager` and `RoleManager`. In runtime, the JAAS provider finds all the registered realms and their attributes (name, user manager implementation class, role manager implementation class, and their properties) from the provider type (Oracle Internet Directory) and instantiates the realm's implementation class with the properties for initialization.

Realm Hierarchy As Figure 3–6 illustrates, JAZN stores its entries within the product container `cn=JAZNContext`. Beneath `cn=JAZNContext` is a `cn=Realms` container, which stores realm entries, and a `cn=Policy` container, which stores global JAZN policies. The `cn=Policy` container in turn stores two types of entries, `cn=Permissions` and `cn=Grantees`.

Note that JAZN has its own Groups and Users containers. The first contains the groups `JAZNAdminGroup` and `JAZNClientGroup`. The second contains the users that populate these groups. These user entries fall under the headings `JAZNAdminUser` and `JAZNClient`. `JAZNAdminUser` is the JAZN superuser and is, by default, a member of `JAZNAdminGroup`.

Figure 3–6 Global JAZNContext Subtree

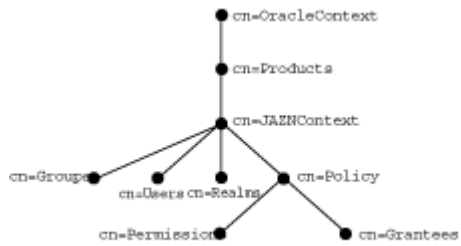
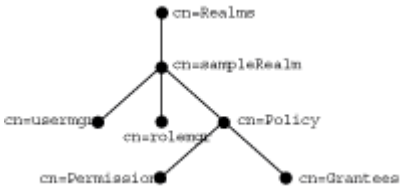


Figure 3–7 shows the directory entries that are placed under the hypothetical realm `cn=sampleRealm`. The entry `cn=usermgr` stores information related to user management while the entry `cn=rolemgr` stores information related to role (group)

management. The policy-related entries under `cn=sampleRealm` store realm-specific policies.

Figure 3–7 A Realm-Specific Subtree



In a subscriber-based environment, a subscriber is registered as a realm. Using the subscriber DN, JAZN locates the subscriber-specific Oracle Context and creates a `cn=JAZNContext` subtree. In this case, JAZN stores the entries `cn=usermgr` and `cn=rolemgr` and policy-related entries under the subscriber's `JAZNContext`.

In Figure 3–8 `cn=oracle` is a subscriber.

Figure 3–8 Subscriber JAZNContext Subtree

Security Measures For Java Authorization Service JAZN directory entries are protected by ACLs at the root of the product subtree. These ACLs grant the group `JAZNAdminGroup` and the JAZN superuser `JAZNAdminUser` full privileges (read, write) for JAZN directory objects. Members of `JAZNClientGroup` have read-only privileges. Users who are not members of one of these groups are denied access to JAZN entries.

Because subscriber `JAZNContext` subtrees are mirror images of their site-wide parents, the security measures that they use to protect entries are the same.

> **See Also:** Oracle9*i* Application Server Java Authorization Developer's Guide

LDAP-Based Realm Permissions

A `RealmPermission` class is defined to represent realm permissions. `RealmPermission` extends from `java.security. Permission`. It is used like any regular Java permission. `RealmPermission` has the following characteristics:

- Realm name, also known as target name
- List of actions (permissions applicable to the realm, such as creating a realm, dropping a role, and so on)

> **See Also:**
>
> - "RealmPermission" on page A-9
> - The JAAS Provider API Reference (Javadoc) is located in the Oracle9*i* Application Server Documentation Library on the J2EE & Internet Applications tab

Realm Management in XML-Based Environments

A realm provides user and role management. For XML-based environments, realm management is less restrictive and faster: a more lightweight implementation than LDAP-based realm management.

XML-Based Realm Types

The JAAS provider enables you to create a single realm type for an XML-based environment.

> **See Also:** "Using the JAZN Admintool" on page 6-12 for instructions on creating realm types.

XML-Based Realm and Policy Information Storage

An XML-based realm enables you to:

- Create realms, users, and roles
- Grant roles to users and to other roles
- Assign permissions to specific users and roles (principals)

This information is stored in an XML file, typically, `jazn-data.xml`. The following example shows the structure used in a `jazn-data.xml` file to create realms, users, and roles.

```
<!--JAZN Realm Data -->

   <jazn-realm>
        <realm>
             <name>jazn.com</name>
             <users>
                  <user>
                       <name>admin</name>
                       <displayName>Realm Administrator</displayName>
                       <description>Administrator for this realm</description>
                       <credentials>Qj+w7NJulLM=</credentials>
                  </user>
                  <user>
                       <name>anonymous</name>
                       <description>The default guest/anonymous
                                user</description>
                  </user>
             </users>
             <roles>
                  <role>
                       <name>guests</name>
                       <members>
                            <member>
                                 <type>user</type>
                                 <name>admin</name>
                            </member>
                            <member>
                                 <type>user</type>
                                 <name>anonymous</name>
                            </member>
                       </members>
                  </role>
                  <role>
                       <name>administrators</name>
                       <displayName>Realm Admin Role</displayName>
                       <description>Administrative role for this
                                    realm</description>
                       <members>
                            <member>
                                 <type>user</type>
                                 <name>admin</name>
```

```
                    </member>
                </members>
            </role>
            <role>
                <name>users</name>
                <members>
                    <member>
                        <type>user</type>
                        <name>admin</name>
                    </member>
                </members>
            </role>
        </roles>
    </realm>
</jazn-realm>
```

See Also: "Sample jazn-data.xml Code" on page B-2 for a completed `jazn-data.xml` file.

Note: Setting the `<credentials>` element as follows enables you to use clear (readable) passwords in the `jazn-data.xml` file the first time.

- `<credentials clear="true">welcome</credentials>`
- `<credentials>!welcome</credentials>`

This enables the administrator to directly edit `jazn-data.xml` with a text editor. When the file is read and persistence occurs, the password in `jazn-data.xml` is obfuscated and becomes unreadable.

JAAS Provider Policy Administration

The JAAS provider implementation of `javax.security.auth.Policy` uses either an LDAP-based Oracle Internet Directory or XML-based provider type for storing policy (authorization rules). The JAAS provider administrator uses various grant and revoke methods of the `JAZNPolicy` class to create authorization policies for principals.

The provider must be administered in a secure manner. There are several ways to administer the JAAS provider policy:

- Oracle Enterprise Manager (LDAP environments only)
- JAZN Admintool

- Oracle Internet Directory Administration
- AdminPermission Class

> **See Also:** Table 3–6 on page 3-17 for information on Oracle Enterprise Manager and "Using the JAZN Admintool" on page 6-12 for information on the JAZN Admintool

Oracle Internet Directory Administration

For LDAP-based application environments, you manage realm and policy data as Oracle Internet Directory entries through:

- The OiD DAS and Oidadmin administrative tools
- Definition of access control lists in Oracle Internet Directory

Two possible administrative groups can manage the data:

- A JAAS provider site-wide administrative group that is granted permissions to access and modify the site-wide `JAZNContext` and any subscriber-specific `JAZNContext`
- A realm-specific administrative group for each realm instance or administrative user

In hosted application environments, part of the policy data may be partitioned along subscriber boundaries and thus stored in a subscriber subtree. That policy data cannot be administered by the realm-specific administrative group. The same is true with role information.

With the JAAS provider policy data (including realm data), only users that belong to `JAZNClientGroup` or `JAZNAdminGroup` have read-access capabilities on provider data.

The LDAP-based environment caches provider policy data; for details, see “Managing JAAS Provider Policy” on page 33.

> **See Also:** *Oracle Internet Directory Administrator’s Guide*

AdminPermission Class

The `AdminPermission` class can be used in either LDAP-based or XML-based environments.

The `AdminPermission` class represents the right to administer a permission. This enables a grantee (such as a user named `frank`) to further grant and revoke the granted right / permission to other grantees. Instances of this permission class

include instances of other permissions. Since this is a permission about permission, it varies slightly from the permission definition, which includes a simple name, actions pair. This variation is resolved by encoding a permission instance as a string and using that as the name of the AdminPermission instance. Table 3–11 provides an example:

Table 3–11 ADMIN Option Example

If User...	Then User...
frank is granted the AdminPermission for java.io.FilePermission("/tmp/*","read, write")	frank can further grant and revoke any permission implied by the embedded permission (that is, FilePermission in this instance).

When expressed in the format recognized by the policy provider, this results in the following:

```
 grant Principal com.oracle.security.jazn.JAZNPrincipal "frank"
{
  permission com.oracle.security.jazn.policy.AdminPermission
     "class=java.io.FilePermission, name=\"/tmp/*\", actions=\"read, write\""
};
```

Note that another permission instance is encoded in the target name for this AdminPermission instance.

Recursive embedding of AdminPermission (that is, an AdminPermission instance embedded within another AdminPermission instance) is not supported. In the initial policy, the JAAS user is granted AdminPermission to java.security.AllPermission, enabling the JAAS user to grant and revoke all permissions to anyone.

A RoleAdminPermission class is defined for roles. This means that when role hr is granted to frank, frank is granted both role hr and a RoleAdminPermission that enables frank to further grant and revoke role hr.

> **See Also:** "Policies and Permissions" on page 3-10 for an example of an XML-based policy file

Policy Partitioning

The JAAS provider supports policy partitioning among realms (that is, each realm has its own realm-specific policy). This realm-specific policy is administered by the realm-specific administrative group.

In a *hosted environment*, a subscriber is represented by a realm and the subscriber-specific information subtree is stored under a subscriber-specific `JAZNContext`. This subscriber-specific subtree, however, is primarily administered by the JAAS Provider administrative group from the perspective of the LDAP server (Oracle Internet Directory).

4

Quick Start JAAS Provider Demo

This chapter describes how to quickly configure and run a sample Java2 Platform, Enterprise Edition (J2EE) application that uses the JAAS Provider, the Oracle9*i*AS Containers for J2EE (OC4J) user authentication, authorization, and delegation service.

This chapter contains these topics:

> **Notes:** For the purpose of this Quick Start demonstration, many terms and concepts in this chapter are described at a high level. Where appropriate, references are provided to other sections in this and other guides for specific information on these terms and concepts.
>
> This example provides instructions for use with the standalone version of OC4J. Please refer to the OC4J User's Guide for instructions on using the example with the complete Oracle9*i*AS installation.

Quick Start JAAS Provider Demo Overview

This Quick Start demo is designed to get you up and running with JAAS provider using the sample demo application, `callerInfo`. It also demonstrates the use of the JAZN Admintool.

The `callerInfo` demo indicates whether or not the user attempting to log into the application has succeeded and with which roles and permissions.

The `callerInfo` demo application demonstrates use of the following features:

- OC4J as the HTTP listener that listens for user login requests and functions as the Web container that stores the `callerInfo` application
- Basic authentication for validating the login credentials of the user attempting to access the `callerInfo` demo application (authentication)
- The JAAS provider for enforcing the roles and permissions assigned to the authenticated user (authorization)
- The XML-based provider type as the JAAS provider repository provider for storing users, roles, and permissions
- The J2EE environment to run the application

See Also: The following sections for more detailed information on the concepts covered in this Quick Start demo:

- *Oracle9iAS Containers for J2EE User's Guide* for further information on OC4J configuration
- *Oracle9i Application Server Security Guide* for further information on JAAS Provider configuration
- "Integrating the JAAS Provider with Basic Authentication" on page 5-12 for further information on Basic authentication
- "Realm Management in XML-Based Environments" on page 3-25 for further information on using XML files as the JAAS Provider environment type
- "JAAS Provider Integration in J2SE Application Environments" on page 5-2 for further information on the J2EE environment

Setting Up the Demo

These are the basic tasks you must perform to set up the Quick Start demo:

- Task 1: Modifying OC4J Configuration Files
- Task 2: Changing Default Configurations (Optional)

Task 1: Modifying OC4J Configuration Files

In order to use the `callerInfo` demo, you must modify two OC4J files in `$ORACLE_HOME/j2ee/home/config/`.

1. Modify the `server.xml` file by removing the comments around:

```
<application name="callerInfo" path="../jazn/demo/callerInfo/callerInfo.ear" />
```

2. Modify the `default-web-site.xml` file by removing the comments around:

```
<web-app application="callerInfo" name="callerInfo-web" root="/jazn" />
```

See Also:

- *Oracle9iAS Containers for J2EE User's Guide* for further information on OC4J configuration
- *Oracle9i Application Server Security Guide* for further information on JAAS Provider configuration

Task 2: Changing Default Configurations (Optional)

The sample `callerInfo` application is installed with several default configuration settings that enable you to immediately run the JAAS provider. If you want to run the JAAS provider using these default settings, you can skip this section and go to "Running the Demo" on page 4-4.

If you make any changes to the default configurations, rebuild the directory with jar or Ant.

For the purpose of this demo, two different realms are available for experimentation. Realms provide access to users and roles. The two realms are contained in `jazn-data.xml` files located in the directory `j2ee/home/jazn/config/`:

- A sample realm, `sample_subrealm`, is defined in the `jazn-data.xml` file. `sample_subrealm` and the `jazn-data.xml` file are the current defaults.
- A more complex sample realm, `jazn.com`, is defined in the `jazn-data1.xml` file.

To use a realm other than the default `sample_subrealm`, you must modify the `jazn` element of the OC4J `orion-application.xml` (in the directory `jazn/demo/callerinfo/etc/`) as follows:

- Change the realm, `default-realm`, from the default value, `sample_subrealm`, to `jazn.com` or any realm that you have created.
- Change `location` from the default value, `jazn-data.xml`, to `jazn-data1.xml` or any properly configured data file that you have created.

> **See Also:** "Managing XML-Based Provider Data with the XML Schema" on page 6-33 for further information on the `jazn-data.xml` file

Running the Demo

To start OC4J and connect to the demo application:

1. Start OC4J with the JAAS provider as follows:

   ```
   java -jar oc4j.jar
   ```

   For the purposes of this Quick Start demo, an insecure and simple manner for starting OC4J is presented. For more information about starting OC4J in secure mode, see "Starting an Application" on page 8-8.

2. Run the `callerInfo` application from a Web browser:

   `http://`*`hostname`*`:8888/jazn`

3. Follow instructions on the Web page.
4. Log in with either of the following usernames and passwords:
   - `admin/welcome`

     Username `admin` is assigned the role `manager`, which is mapped to `sr_manager`.
   - `user/456`

     Username `user` is assigned the role `developer`, which is mapped to `sr_developer`.

See Also:

- *Oracle9iAS Containers for J2EE User's Guide*
- "Testing and Executing the J2EE Application" on page 8-4 for further information on starting OC4J with the JAAS provider
- Chapter 8, "Developing Secure J2EE Applications" to view the code for the `callerInfo` demo used in this Quick Start demo

Viewing the Results of the callerInfo Demo

When the call to the `callerInfo` demo application is successful, with the username `user`, for example, the browser displays a message similar to the following:

```
Time stamp: Fri Aug 24 19:11:37 PDT 2001 request.getRemoteUser =
sample_subrealm/user
request.isUserInRole('FOO') = false
request.isUserInRole('ar_manager') = false
request.isUserInRole('ar_developer') = true
request.getUserPrincipal = ([JAZNUserAdaptor: user=[XMLRealmUser:
sample_subrealm/user])
```

In summary, this Quick Start demo performed the following:

- The login request from username `user` used basic authentication to access the `callerInfo` demo application.
- The OC4J listener listened for the login request from username `user`.
- The JAAS provider enforced the roles and permissions assigned to the authenticated user `user`.
- The users, roles, and permissions were retrieved from the XML-based JAAS provider type.

Testing the JAZN Admintool

The JAZN Admintool is a Java console application that manages provider data from the command prompt.

You can invoke the JAZN Admintool from the UNIX command line interface as follows:

```
java -jar jazn.jar -listusers sample_subrealm
```

These are a few of the command options that you can experiment with from a command-line interface.

```
-listusers [realm [-role role|-perm permission]]
-listroles [realm [user|-role role]|-perm permission]
-listrealms
-listperms {realm user |-role role|-realm realm}
-help
```

The JAZN Admintool also includes a shell. The following screen listing shows how to access the JAZN Admintool shell and some basic shell commands that you can run, with results.

```
> java -jar jazn.jar -shell
JAZN:> ls
realms       policy
JAZN:> cd realms
JAZN:> ls
sample_subrealm

JAZN:> cd sample_subrealm
JAZN:sample_subrealm> ls
users        roles
JAZN:sample_subrealm> cd users
JAZN:sample_subrealm> ls
admin
rachel
naresh
ray
stella
anonymous
```

```
JAZN:sample_subrealm> add scott tiger
JAZN:sample_subrealm> ls
anonymous
rachel
ray
scott
stella
admin
naresh

JAZN:sample_subrealm> rm scott
JAZN:sample_subrealm> ls
admin
rachel
naresh
ray
stella
anonymous

JAZN:sample_subrealm> exit
JAZN:sample_subrealm>
```

See Also: "Using the JAZN Admintool" on page 6-12

5

Integrating the JAAS Provider with Java2 Applications

This chapter describes how to integrate the JAAS provider with applications developed for Java2 environments in Oracle9*i*AS Containers for J2EE (OC4J).

This chapter contains these topics:

Java2 Application Environments Overview

The JAAS provider integrates into applications developed for several Java2 environments:

- **Java2 Platform, Standard Edition (J2SE)**

  For developing, deploying, and managing standalone Java applications

  See Also: "JAAS Provider Integration in J2SE Application Environments" on page 5-2

- **Java2 Platform, Enterprise Edition (J2EE)**

  For developing, deploying, and managing multi-tier, Web-based applications

  See Also: "JAAS Provider Integration in J2EE Application Environments" on page 5-3

Oracle Components Available on the Java2 Platform

When the JAAS provider is integrated with applications developed for the Java2 Platform, the following Oracle components are available to developers:

- The JAAS provider, which provides support for storage, retrieval, and administration of realm information (users and roles) and policy information (permissions). The JAAS provider supports two possible repositories or *provider types*:
  - LDAP-based Oracle Internet Directory (available only with Oracle9*i*AS Infrastructure installation)
  - XML-Based Provider Type
- Login modules, such as the JAAS provider `RealmLoginModule`

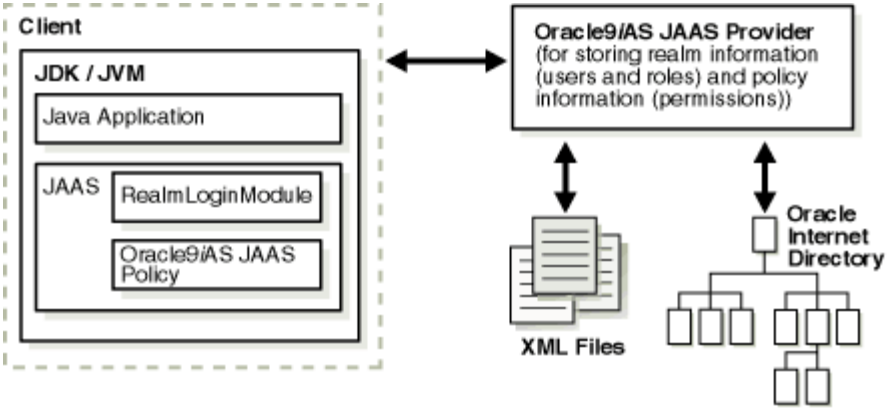
See Also:
- "Provider Types" on page 3-3 for further information about provider types
- Chapter 7 of the *Oracle9i Application Server Security Guide* for required components

---

JAAS Provider Integration in J2SE Application Environments

Figure 5–1 is an overview of an application running in a J2SE environment.

Figure 5–1 Oracle Component Integration in J2SE Environment



A Typical Scenario in the J2SE Environment

The following section describes the responsibilities of the Oracle components illustrated in Figure 5–1 when a client request is initiated.

1. A client attempts to access a local, desktop application.
2. `RealmLoginModule` or another `LoginModule` authenticates the client's login attempt.
3. The Java virtual machine (JVM) examines the authorization context associated with the current thread, consults the JAAS provider policy, determines that the current subject has the required permission to write to the file, and returns `checkPermission()` safely.

> **See Also:** Your Sun Java documentation for more information on J2SE by visiting the following URL:
>
> `http://java.sun.com/j2se/`

JAAS Provider Integration in J2EE Application Environments

When the JAAS provider is integrated with applications developed for the J2EE environment, the functionality of the J2SE environment extends to the enterprise level. Additional features in the J2EE environment include:

- Oracle9iAS Containers for J2EE (OC4J)
- JAZNUserManager

Oracle9*i*AS Containers for J2EE (OC4J)

OC4J is a key component of the JAAS provider integration in the J2EE environment. OC4J is a Web container that accepts HTTP and RMI client connections. These connections permit access to servlets, Java Server Pages (JSPs), and Enterprise JavaBeans (EJBs).

J2EE containers separate business logic from resource and lifecycle management. This enables developers to focus on writing business logic, rather than writing enterprise infrastructure. For example, Java servlets simplify Web development by providing an infrastructure for component, communication, and session management in a Web container integrated with a Web server.

The JAAS provider is also integrated with OC4J to enhance application security. This integration provides the following benefits:

- Integration with either single sign-on (SSO) and `mod_osso` or secure socket layer (SSL) and `mod_ossl`
- Fine-grained access control through Java2 permissions
- `run-as` identity support, delegation support (from servlet to Enterprise JavaBeans)
- Secure file-based storage of passwords

JAZNUserManager

Another key component of JAAS provider integration in the J2EE environment is `JAZNUserManager`. `JAZNUserManager` is an implementation of the OC4J `UserManager` interface.

Replacing principals.xml

`JAZNUserManager` permits secure replacement for or migration from the OC4J `principals.xml` file with the following:

- Secure storage of obfuscated passwords
- Full role-based access control (RBAC), including hierarchical roles
- Full support for the Java2 permission model and JAAS
- Secure implementation based on the Java2 permission model, to allow untrusted (or partially trusted) code to run in the same JVM as the JAAS provider

> **See Also:** For information on using the JAZN Admintool to migrate from `principals.xml`, "Migrating Principals from the principals.xml File" on page 6-19

JAZNUserManager Features

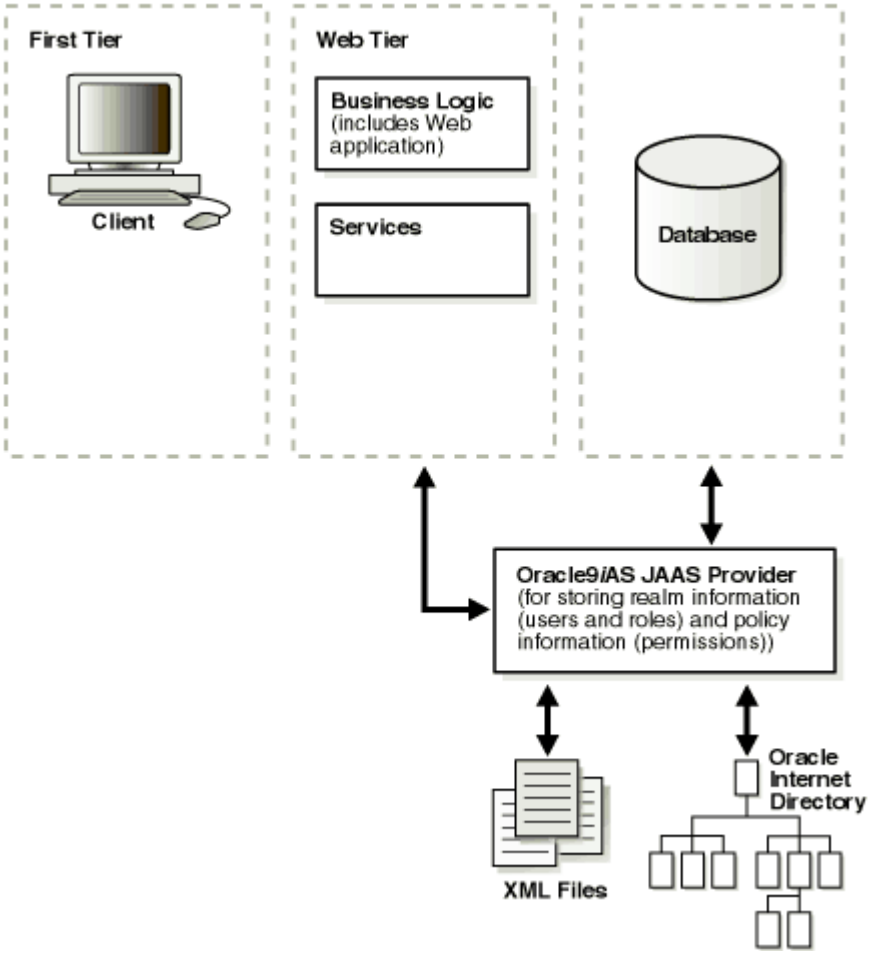
In addition to the features mentioned in "Replacing principals.xml" on page 5-4, `JAZNUserManager` provides many other features, including:

- Single Sign-On (SSO) integration with OC4J
- `RealmLoginModule` integration in non-SSO environments
- Identity propagation
- Location, reading, editing, removal, and management of user and group objects
- Enforcement of security constraints
- A filter for changing the content of HTTP requests, responses, and header information.

> **See Also:** Chapter 7 of the *Oracle9i Application Server Security Guide* for information on the `JaznUserManager`

Figure 5–2 provides an overview of an application running in a J2EE environment.

Figure 5–2 J2EE Application Model



Authentication Environments

The JAAS provider integrates with three different login authentication environments in a J2EE applications.

- **SSO**

  Uses Oracle9*i*AS Single Sign-On to authenticate logins

- **SSL**
  - Uses Secure Socket Layers for client certificate-based authentication
  - Uses a login module (for example, `RealmLoginModule`) to authenticate logins
- **Basic Authentication**
  - Prompts user directly for username and password, without going through Oracle9*i*AS Single Sign-On
  - Uses a login module (for example, `RealmLoginModule`) to authenticate logins

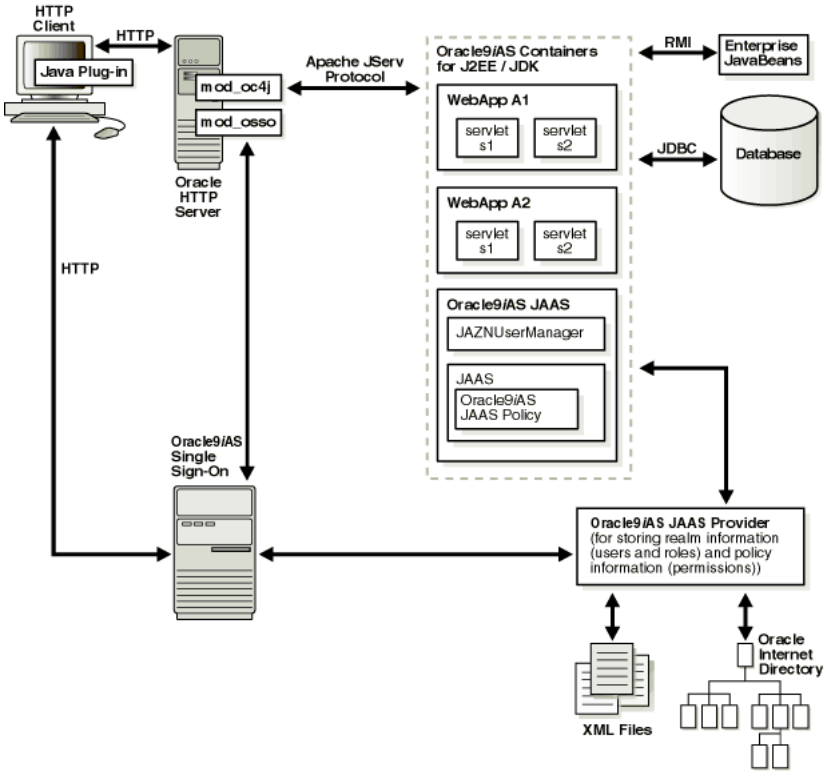
The following sections discuss how the JAAS provider integrates with each of these authentication types.

> **See Also:** Chapter 7 of the *Oracle9i Application Server Security Guide* for information on configuring authentication methods

Integrating the JAAS Provider with SSO-Enabled Applications

SSO lets a user access multiple accounts and applications with a single set of login credentials. Figure 5–3 shows JAAS provider integration in an application running in an SSO-enabled J2EE environment.

Figure 5–3 Oracle Component Integration in SSO-Enabled J2EE Environments



SSO-Enabled J2EE Environments: A Typical Scenario

This section describes the responsibilities of Oracle components when an HTTP client request is initiated in an SSO-enabled J2EE environment.

1. An HTTP client attempts to access a Web application (named WebApp A1) hosted by OC4J (the Web container for executing servlets). Oracle HTTP Server (using an Apache listener) handles the request.

2. mod_osso / Oracle HTTP Server receives the request and:
   - Determines that WebApp A1 application requires Web-based SSO for authenticating HTTP clients
   - Redirects the HTTP client request to the Web-based SSO Oracle9*i*AS Single Sign-On (since it has not yet been authenticated).
3. The HTTP client is authenticated by Oracle9*i*AS Single Sign-On through HTTP or public key infrastructure (PKI) Authentication. Oracle9*i*AS Single Sign-On then:
   - Validates the user's stored login credentials
   - Sets the SSO cookie (including the user's distinguished name and realm)
   - Redirects back to the WebApp A1 application (in OC4J)
4. The JAAS provider retrieves the SSO user.
5. The final step or steps depend on the setting of the `runas-mode` in the `jazn-web-app` element.

   If the `runas-mode` is set to false, then the following happens:

   a. The target servlet is invoked.

   If the `runas-mode` is set to true, then the following happens:

   a. The JAAS provider invokes the target servlet's `service()` method within a `PrivilegedAction` block through `Subject.doAs()`. The `JAZNUserManager` enforces security constraints.
      - When `Subject.doAs()` is called, JAAS consults the provider for permissions associated with the SSO user through the `getPermissions()` method.
      - The provider retrieves the permissions associated with the given grantee from the provider type (Oracle Internet Directory or XML-based), and updates the policy cache as appropriate. The provider then returns the granted set of permissions to JAAS runtime.
      - JAAS runtime constructs a new `AccessControlContext` based on the permissions returned from `getPermissions()`.

   b. The servlet's code runs under the `AccessControlContext` of the SSO user.

   c. The servlet's code attempts to write to a file in the operating system's file system, triggering a call to `SecurityManager.checkPermission()`.

d. The JVM then:

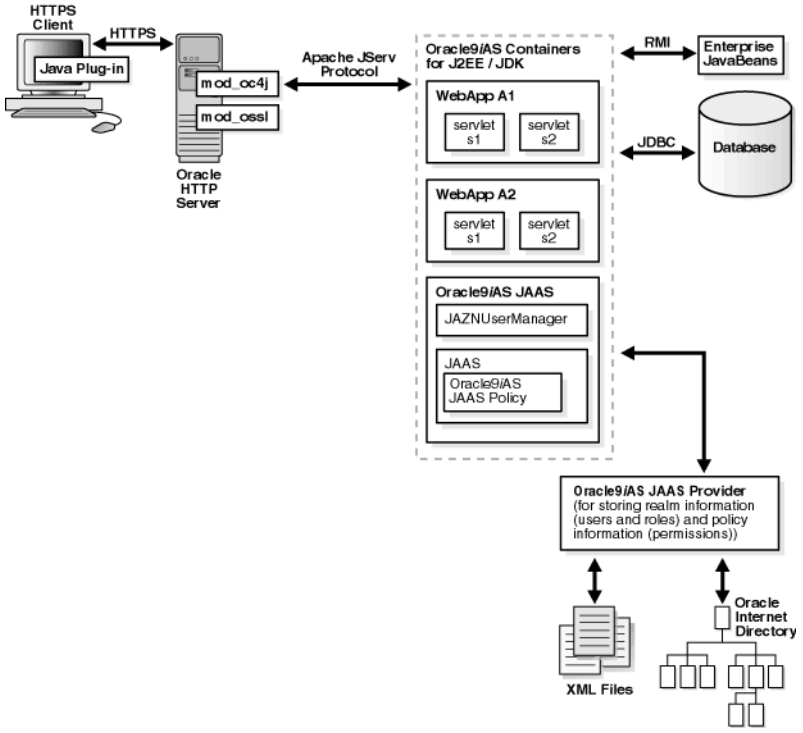
- Examines the authorization context associated with the current thread
- Determines that the current subject has the required permissions to write to the file

e. `SecurityManager.checkPermission()` returns safely and the client HTTP request proceeds.

Integrating the JAAS Provider with SSL-Enabled Applications

SSL is an industry standard protocol for managing the security of message transmission on the Internet. Figure 5–4 shows the JAAS provider integration in an application running in an SSL-enabled J2EE environment.

Figure 5–4 Oracle Component Integration In SSL-Enabled J2EE Environments



SSL-Enabled J2EE Environments: A Typical Scenario

This section describes the responsibilities of Oracle components when an HTTP client request is initiated in an SSL-enabled J2EE environment. In this environment, Oracle9*i*AS Single Sign-On is not used. A login module (for example, `RealmLoginModule`) is used.

1. An HTTP client attempts to access a Web application (named WebApp A1) hosted by OC4J (the Web container for executing servlets). Oracle HTTP Server (using an Apache listener) handles the request.
2. mod_ossl / Oracle HTTP Server receives the request and determines that the WebApp A1 application requires SSL server authentication for HTTP clients.
3. If a server and / or client wallet certificate is configured, the HTTP client is prompted to accept the server certificate and provide the client certificate.
4. The JAAS provider retrieves the SSL client certificate.
5. The JAAS provider retrieves the SSL user.
6. The final step or steps depend on the setting of the `runas-mode` in the `jazn-web-app` element.

   If the `runas-mode` is set to false, then the following happens:

   a. The target servlet is invoked.

   If the `runas-mode` is set to true, then the following happens:

   a. The JAAS provider invokes the target servlet's `service()` method within a `PrivilegedAction` block through `Subject.doAs()`. The `JAZNUserManager` enforces security constraints.
      - When `Subject.doAs()` is called, JAAS consults for permissions associated with the SSL user through the `getPermissions()` method.
      - The provider retrieves the permissions associated with the given grantee from the provider type (Oracle Internet Directory or XML-based), and updates the policy cache as appropriate. The provider then returns the granted set of permissions to JAAS runtime.
      - JAAS runtime constructs a new `AccessControlContext` based on the permissions returned from `getPermissions()`.

   b. The servlet's code runs under the `AccessControlContext` of the SSL user.

c. The servlet's code attempts to write to a file in the operating system's file system, triggering a call to `SecurityManager.checkPermission()`.

d. The JVM then:

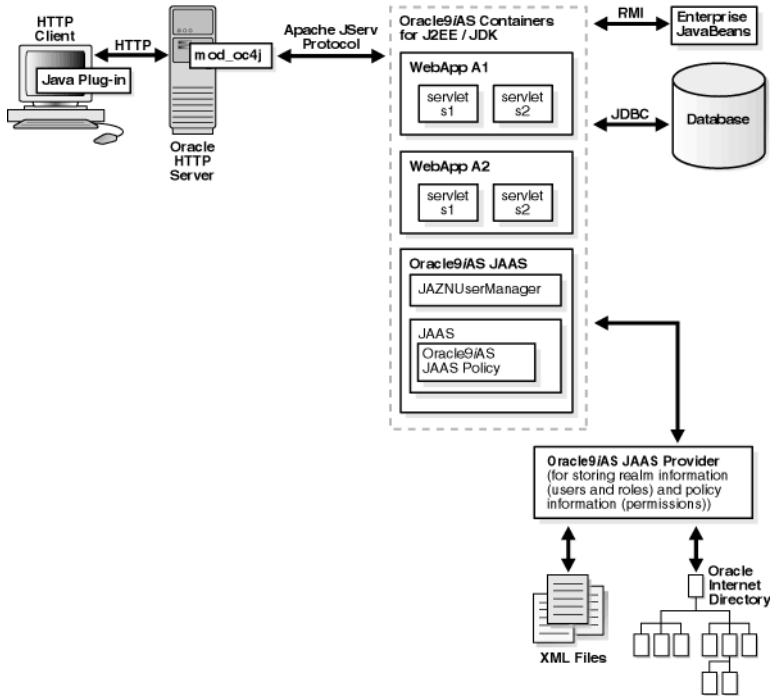
- Examines the authorization context associated with the current thread
- Determines that the current subject has the required permissions to write to the file

e. `SecurityManager.checkPermission()` returns safely and the client HTTP request proceeds.

Integrating the JAAS Provider with Basic Authentication

Basic authentication bypasses Oracle9*i*AS Single Sign-On. Figure 5–5 shows specific JAAS provider integration in an application configured for Basic authentication in a J2EE environment.

Figure 5–5 Oracle Component Integration in j2ee Environment



Basic Authentication J2EE Environments: Typical Scenario

This section describes the responsibilities of Oracle components when an HTTP client request is initiated in a J2EE environment configured for Basic authentication. In this environment, Oracle9*i*AS Single Sign-On is not used. A login module (for example, `RealmLoginModule`) is used.

Note: If you have configured BASIC authentication, OC4J invokes the `RealmLoginModule` whenever the user credentials are required. For example, when a request hits a protected page, OC4J will ask the JAAS provider to authenticate the user, then the `RealmLoginModule` will be invoked to authenticate the user, using the credentials sent by the user via the browser over HTTP.

---

1. An HTTP client attempts to access a Web application (named WebApp A1) hosted by OC4J (the Web container for executing servlets). The OC4J listener handles the request.
2. The JAAS provider retrieves the user.
3. The final step or steps depend on the setting of the `runas-mode` in the `jazn-web-app` element.

   If the `runas-mode` is set to false, then the following happens:

   a. The target servlet is invoked.

   If the `runas-mode` is set to true, then the following happens:

   a. The JAAS provider invokes the target servlet's `service()` method within a `PrivilegedAction` block through `Subject.doAs()`. The `JAZNUserManager` enforces security constraints.
      - When `Subject.doAs()` is called, JAAS consults the provider for permissions associated with the SSO user through the `getPermissions()` method.
      - The provider retrieves the permissions associated with the given grantee from the provider type (Oracle Internet Directory or XML-based), and updates the policy cache as appropriate. The provider then returns the granted set of permissions to JAAS runtime.
      - JAAS runtime constructs a new `AccessControlContext` based on the permissions returned from `getPermissions()`.

   b. The servlet's code runs under the `AccessControlContext` of the user.

   c. The servlet's code attempts to write to a file in the operating system's file system, triggering a call to `SecurityManager.checkPermission()`.

   d. The JVM then:
      - Examines the authorization context associated with the current thread
      - Determines that the current subject has the required permissions to write to the file

   e. `SecurityManager.checkPermission()` returns safely and the client HTTP request proceeds.

See Also: Your Sun Java documentation for more information on J2EE by visiting the following URL:

`http://java.sun.com/j2ee/`

J2EE and JAAS Provider Role Mapping

Two distinct roles types are available to application developers creating JAAS provider-integrated applications in J2EE environments: J2EE roles and JAAS provider roles. When these role types are mapped together using OC4J group mappings, users can access an application with a defined set of role permissions for as long as the user is mapped to this role.

This section describes these role types and how which they are mapped together.

- J2EE Security Roles
- JAAS Provider Roles and Users
- OC4J Group Mapping to J2EE Security Roles

J2EE Security Roles

The J2EE development environment includes a portable security roles feature defined in the `web.xml` file for servlets and Java Server Pages (JSPs). Security roles define a set of resource access permissions for an application. Associating a principal (in this case, a JAAS provider user or role) with a security role assigns the defined access permissions to that principal for as long as they are mapped to the role. For example, an application defines a security role called `sr_developer`:

```
<security-role>
     <role-name>sr_developer</role-name>
</security-role>
```

You also define the access permissions for the `sr_developer` role.

```
 <security-constraint>
    <web-resource-collection>
      <web-resource-name>access to the entire application</web-resource-name>
      <url-pattern>/*</url-pattern>
    </web-resource-collection>
        <!-- authorization -->
    <auth-constraint>
      <role-name>sr_developer</role-name>
    </auth-constraint>
  </security-constraint>
```

JAAS Provider Roles and Users

JAAS provider roles and Users are defined depending on the provider type, LDAP-based Oracle Internet Directory or XML-based.

For example, with the XML-based provider type, `developer is listed as a role element in the jazn-data.xml file:`

```
<role>
      <name>developer</name>
      <members>
            <member>
                  <type>user<type>
                  <name>john<name>
             </member>
       </members>
</role>
```

OC4J Group Mapping to J2EE Security Roles

OC4J enables you to map portable J2EE security roles defined in the J2EE `web.xml` file to groups in an `orion-application.xml` file.

The roles and users defined in your provider environment are mapped to the OC4J `developer` group role in the `orion-application.xml` file.

For example, the `sr_developer` security role is mapped to the group named `developer.`

```
<security-role-mapping name="sr_developer">
       <group name="developer" />
  </security-role-mapping>
```

This association permits the `developer` group to access the resources allowed for the `sr_developer` security role.

User `john` is listed as a member of the `developer` role. Because the `developer` group is mapped to the J2EE security role `sr_developer` in the `orion-application.xml` file, `john` has access to the application resources defined by the `sr_developer` role.

How Do I Get Started?

You are now ready to get started with the JAAS Provider. To get started quickly, follow the sections in Table 5–1 in the exact order listed:

Table 5–1 Getting Started with the JAAS Provider

To...	See...
Identify and install the JAAS provider components required for applications developed in the J2SE and J2EE environments	The *Oracle9i Application Server Installation Guide* for your operating system
Configure the JAAS provider after installation	Chapter 7 of the *Oracle9i Application Server Security Guide*
Create realms and associated components with the provider	Chapter 6, "Managing the JAAS Provider"
Create secure J2SE and J2EE applications with the JAAS provider	Chapter 7, "Developing Secure J2SE Applications" Chapter 8, "Developing Secure J2EE Applications"

6

Managing the JAAS Provider

This chapter describes how to manage the Oracle9*i*AS Containers for J2EE (OC4J) JAAS Provider in Java2 Platform, Standard Edition (J2SE) and Java2 Platform, Enterprise Edition (J2EE) environments.

This chapter contains these topics:

JAAS Provider Management Overview

Managing the JAAS provider in the J2SE and J2EE environments involves creating and managing realms, users, roles, permissions, and policy.

How you manage the JAAS provider depends on two things:

- Whether your provider is XML-based or LDAP-based Oracle Internet Directory
- Which of the available tools (alone or in combination) you are using:
  - Oracle Enterprise Manager (policy and permission management, only with this release)
  - JAZN Admintool, a command-line tool
  - Java Programs for LDAP Management, based on the JAAS Provider APIs

- Other Utilities including:

  ```
  -PermissionClassManager

  -PrincipalClassManager

  -LoginModuleManager
  ```

Note: Based on the provider type you are using, these tools are used in slightly different contexts and are not necessarily directly parallel in function. For example, the JAZN Admintool enables you to create users if your provider type is the XML-Based Provider Type, but not if your provider type is LDAP-based.

Therefore, if you are planning to rely on either the Oracle Enterprise Manager or the JAZN Admintool, also read the appropriate section, "Managing LDAP Provider Data with Java Programs" on page 6-24 or "Managing XML-Based Provider Data with the XML Schema" on page 6-33, for a fuller understanding of the functions available in each environment.

Table 6–1 describes the general functionality of each tool in both XML-based and LDAP-based provider type environments.

Table 6–1 Tools For Managing XML-Based and LDAP-Based Provider Environments

Using This Tool...	With LDAP-Based provider type	With XML-Based provider type
Oracle Enterprise Manager	You can create principals (known as grantees) and assign permissions to these grantees.	This tool is not available.
JAZN Admintool	A broad range of functions is available, including several not included in the API.	A broad range of functions is available, including several not included in the API.
Java Programs for LDAP Management	You have access to all the JAAS Provider API functionality available in an LDAP environment.	This tool is not available.

LDAP-Based and XML-Based JAAS Providers

XML-based and LDAP-based JAAS providers enable different functionalities as described in Table 6–2.

Table 6–2 JAAS Provider Management

JAAS Provider	Description	See Also...
LDAP-based Available with the Oracle9*i*AS Infrastructure installation type)	Enables you to: ■ Create realms ■ Manage roles (in an External Realm or Subscriber Realm) ■ Manage or create roles (in an Application Realm) ■ Assign permissions	"Realm Management in LDAP-Based Environments" on page 3-18 "Managing Realms" on page 6-25
XML-based (Available with all installation types	Enables you to: ■ Create and manage realms, users, and roles ■ Assign permissions	"Realm Management in XML-Based Environments" on page 3-25 "Managing XML-Based Provider Data with the XML Schema" on page 6-33

Using the Oracle Enterprise Manager Interface with the JAAS Provider

You can use Oracle Enterprise Manager to perform two JAAS provider tasks:

- Manage JAAS Policy
- Manage Java Permissions

> **See Also:** Your Oracle Enterprise Manager documentation for instructions on starting Oracle Enterprise Manager

Oracle Enterprise Manager functionality for the JAAS provider is currently only available for the LDAP provider environment and only for policy management tasks.

> **Note:** Oracle Enterprise Manager windows use Add buttons that operate as follows: You enter or select items to be acted upon or searched for, add them to a list using the Add button, and finally process the items.

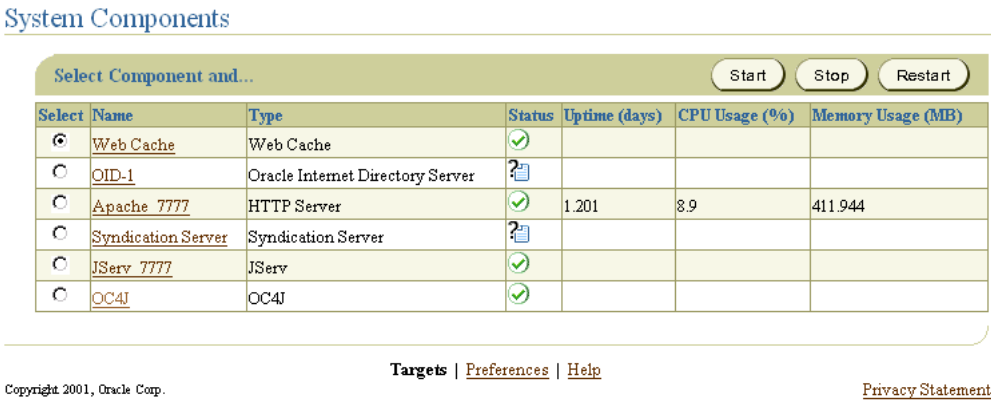
Accessing the JAAS Provider

To use the Oracle Enterprise Manager to perform JAAS provider tasks, navigate to the Oracle9*i* Application Server entry, then to the OC4J system component, and select the application default as follows:

To access the JAAS Provider:

1. Choose the appropriate Oracle9*i* Application Server entity in the Application Servers Name column.
2. Choose OC4J in the System Components list.

   The System Components panel appears:



3. Choose Oracle9*i* Application Server from the list of Application Defaults.

   The main window for the JAAS provider appears:

Task 1: Managing JAAS Policy

Policies, which store JAAS authorization rules, consist of one or more grants or grant entries. Grant entries are grantees (principals and codesource (optional)) and their assigned permissions.

Managing JAAS Policy enables you to:

- Search for existing grant entries and view grant entry data
- Delete grant entries
- Create new grant entries by assigning JAAS provider permissions to principals

Note: To manage JAAS policy, the policy cache must be disabled. This is the default setting.

Searching for And Viewing Existing Grant Entries

To search for and view grant entry data:

1. Choose JAAS Policy from the tab on the left of the main window.

   The JAAS Policy Management window appears. This is the same as the main JAAS provider window. See "Accessing the JAAS Provider" on page 6-4.

   The window immediately displays a results list that you can modify by entering a search phrase or using arrows that guide you to subsequent sections of the results list.

2. Enter the codesource URL, if any.
3. If the grant name you are searching for does not appear immediately on the results list, enter it.

   Wild cards are implied, that is, if you enter several letters, the results list shows all entries that begin with those letters, assuming the case is the same.

4. Choose Go or press Enter.
5. When the grant name you are searching for appears in the results list, click the name to view the grant entry data.

   For the grant name you have entered, the following data appears:

   - Principal Names and classes
   - Permission Names and classes
   - The codesource, if any, assigned to the grant entry

Deleting Grant Entries

To delete grant entry data:

1. Perform the search functions as described "Searching for And Viewing Existing Grant Entries" on page 6-6.
2. Select the grant entry from the results list by choosing the radio button besides the name.

3. Choose Delete.

Creating a New Grant Entry

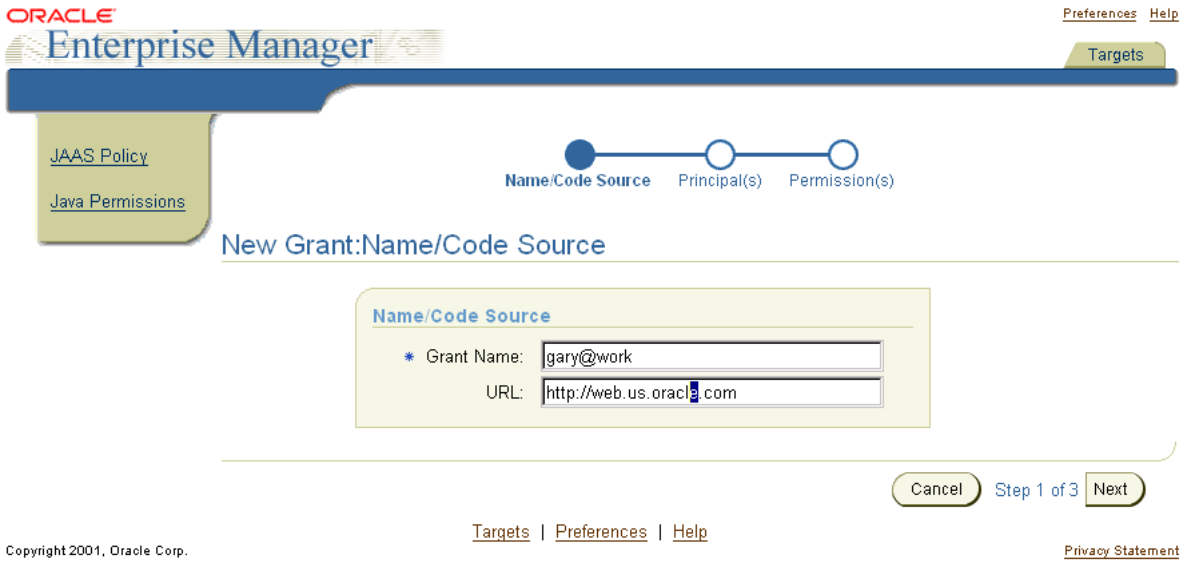
To create a new grant entry:

1. Choose JAAS Policy from the tab on the left.

   The JAAS Policy Management window appears.

2. Choose New Grant.

   The New Grant: Name/CodeSource window appears, and enables you to enter a name for the new grant entry and define a codesource. The codesource is the code associated with the policy entry.



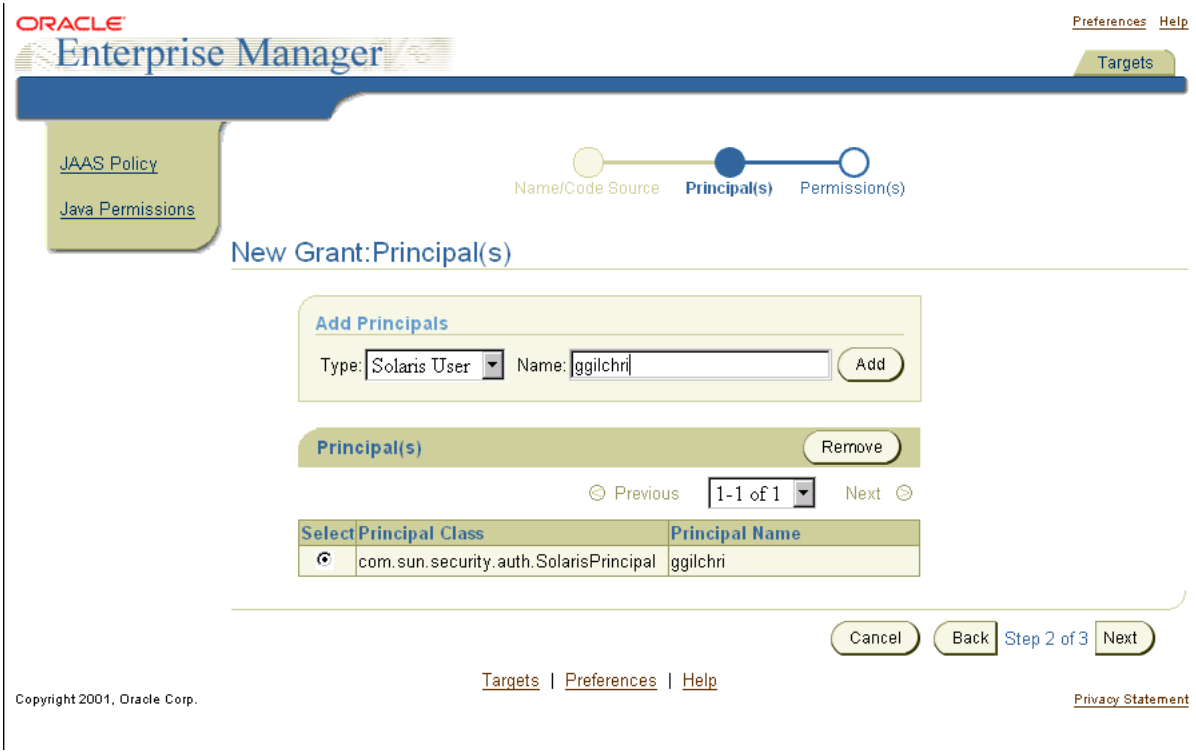
3. Enter a grant name and codesource.
4. Choose Next.

See Also: "Policies and Permissions" on page 3-10 for information on codesources

The New Grant: Principal(s) window appears and enables you to select the principal type and enter one or more principals to define the grant entry.

The available principal types are:

- Solaris User
- LDAP User
- Realm User



5. Select the type and enter the name of a principal.

   If you have selected the LDAP type, the name must be an X.500 distinguished name. Although the system accepts other names, they will be rejected when you finish. For other types, you can enter any name.

6. Choose Add to add this principal to the list of principals being added to this grant.

7. Repeat Steps 5 and 6 until all principals are added to the list of principals.
8. Choose Next to add all principals on the list to the grant.

   The New Grant: Permission window appears and enables you to enter the permission class, target, and action for the grant entry. These are essentially what the user is authorized to do with your application.

   - The class is the Java permission being assigned to the policy (for example, `java.io.FilePermission`).
   - The target is the resource to which this permission applies (for example, files in a directory named `/home/*`).
   - The action is the actions associated with this target (for example, read and write privileges on all files in `/home/*`).

9. Select the class, target, and action from the drop-down list boxes on the left or enter the names directly in the fields on the right.
10. Choose Add to add this permission to the list of permissions to be added the grant.
11. Repeat Steps 9 and 10 until all permissions have been added to the list of permissions.
12. Choose Finish.

    The entry is now granted these permissions on the designated target. The grant entry is complete.

Task 2: Managing Java Permissions

The Java Permissions task enables you to search for and view the permissions of a principal on a given codesource and revoke these permissions. You can search by principal class or principal name.

Searching for And Viewing Existing Permissions

To search for permissions on a principal:

1. Choose Java Permissions from the tab on the left.

   The Permission Management window appears:

2. Enter the codesource URL.
3. Select the principal type from the drop-down list.

   The available principal types are:

   - Solaris User
   - LDAP User
   - Realm User
4. Enter the name of a principal from the principal type.
5. Choose Add to add a principal to the search list. You can search for multiple principals at once.

6. Repeat Steps 4 and 5 until all principals have been added to the search list.
7. Choose Search.

The results display on-screen including permission class, permission target, and permission actions, but the codesource does not appear.

Revoking Permissions Assigned to a Principal

To revoke permissions assigned to a principal:

1. Perform the search function as described in "Searching for And Viewing Existing Permissions" on page 6-10.
2. Revoke permissions by selecting the radio button of an appropriate permission.

   You can only revoke one permission at a time.
3. Choose Revoke.

Using the JAZN Admintool

The JAZN Admintool can manage both XML-based and LDAP-based JAAS provider data from the command prompt.

The JAZN Admintool is a flexible Java console application, with functions that can be called directly from the command line or through the shell interface of the Admintool. The shell uses UNIX-derived commands to perform specific JAAS provider functions.

This section includes the following topics:

- Usage Examples
- Command Options
- Realm Operations
- JAZN Shell Interface
- JAZN Shell Commands

Usage Examples

The following examples illustrate the different ways that the JAZN Admintool commands can be used.

To list all users in realm foo:

From the UNIX command line:

```
java -jar jazn.jar -listusers foo
```

From the shell interface of the Admintool (using command-line options):

```
JAZN:> listusers foo
```

From the shell interface of the Admintool (through modified UNIX commands):

```
JAZN:> cd /realms/foo/users
JAZN:foo> ls
```

To add the role fooRole to realm foo:

From the UNIX command line:

```
java -jar jazn.jar -addrole foo fooRole
```

From the shell interface of the Admintool (using command-line options):

```
JAZN:> addrole foo fooRole
```

From the JAAS provider shell (through modified UNIX commands):

```
JAZN:> cd /realms/foo/users
JAZN:foo> mkdir fooRole
```

Command Options

The JAZN Admintool provides the following command options, which are described in greater detail in the following sections. The JAZN Admintool command options can be invoked several different ways as described in "Usage Examples" on page 6-12. Error messages display if the syntax or parameters specified are incorrect.

Realm Operations

```
-addrealm realm admin {adminpwd adminrole|adminrole
      userbase rolebase realmtype}
-addrole realm role
-adduser realm username password
-checkpasswd realm user [-pw password]
-grantrole role realm {user|-role to_role}
```

```
-listrealms
-listroles [realm [user|-role role]|-perm permission]
-listusers [realm [-role role|-perm permission]]
-remrealm realm
-remrole realm role
-remuser realm user
-revokerole role realm {user|-role to_role}
-setpasswd realm user old_pwd new_pwd
```

Policy Operations

```
-addperm permission permission_class action target [description]
-addprncpl principal_name prncpl_class params [description]
-grantperm realm {user|-role role} permission_class
      permission_actions
-listperms realm {user |-role role|-realm realm}
-listperm permission
-listprncpls
-listprncpl principal_name
-remperm permission
-remprncpl principal_name
-revokeperm realm {user|-role role} permission_class
      permission_actions
```

Interactive Shell

```
-shell
```

Configuration Operations

```
 -getconfig default_realm admin password
```

Migration Operations

```
-convert filename realm
```

Miscellaneous

```
-help
-version
```

Realm Operations

Adding and Removing Realms

-addrealm *realm admin* {*adminpwd adminrole | adminrole userbase rolebase realmtype*}

-remrealm *realm*

The **-addrealm** option creates a realm of the specified type with the specified name, and **-remrealm** deletes a realm.

Valid realm types are:

- LDAP Environment: external and application
- XML Environment: XML

The user must provide the following:

- For an XML provider type:
  - realm name
  - administrator username
  - administrator password
  - administrator role
- For LDAP:
  - realm name
  - administrator name
  - administrator role
  - user search base in the directory
  - role search base in the directory
  - realm type

Adding and Removing Roles

-addrole *realm role*
-remrole *realm role*

The **-addrole** option creates a role in the specified realm, and **-remrole** deletes a role from the realm.

Adding and Removing Users

-adduser *realm username password*
-remuser *realm user*

The **-adduser** option adds a user to a specified realm, and **-remuser** deletes a user from the realm.

Checking Passwords

```
-checkpasswd [realm] user [-pw password]
```

The **-checkpasswd** option indicates whether the given user requires a password for authentication. If -pw is used, it displays a message indicating whether the specified password authenticates the user.

Granting and Revoking Roles

```
-grantrole role realm {user|-role to_role}
-revokerole role realm {user|-role to_role}
```

The **-grantrole** option grants the specified role to a user (when called with a user name) or a role (when called with -role). The **-revokerole** option revokes the specified role from a user or role.

Listing Realms

```
-listrealms
```

The **-listrealms** option displays all realms in the current JAAS provider environments.

Listing Roles

```
-listroles [realm [user|-role role|-perm permission]]
```

The **-listroles** option displays a list of roles that match the list criteria. This option lists the following:

- All roles in all realms, when called without any parameters
- All roles granted to a user, when called with a realm name and user name
- Roles that are granted the specified *role*, when called with a realm name and the option -role
- Roles that are granted the specified *permission*, when called with a realm name and the option -perm

Listing Users

-listusers [*realm* [-role *role*|-perm *permission*]]

The **-listusers** option displays a list of users that match the list criteria. This option lists the following:

- All users in all realms, when called without any parameters
- All users in a realm, when called with a realm name
- Users that are granted a certain role or permission, when called with a realm name and the option -role or -perm

Setting a Password

-**setpasswd** *realm user old_pwd new_pwd*

The **-setpasswd** option allows administrators to reset the password of a user given the old password.

Policy Operations

Adding and Removing Permissions

-addperm *permission permission_class action target* [*description*]
-remperm *permission*

The **-addperm** option registers a permission with the JAAS provider PermissionClassManager. The **-remperm** option unregisters the specified permission class. *permission* and *description* can be multiple words if enclosed by quotation marks ("").

Adding and Removing Principals

-addprncpl *principal_name prncpl_class params* [*description*]
-remprncpl *principal_name*

The **-addprncpl** option registers a principal with the JAAS Provider PrincipalClassManager. The **-remprncpl** option unregisters the specified principal class. *principal_name* and *description* can be multiple words if enclosed by quotation marks ("").

Granting and Revoking Permissions

-grantperm *realm* {user|-role *role*} *permission_class permission_actions*
-revokeperm *realm* {user|-role *role*} *permission_class permission_actions*

The **-grantperm** option grants the specified permission to a user (when called with a username) or a role (when called with -role). The **-revokeperm** option revokes the specified permission from a user or role. A permission is denoted by its explicit class name (for example, oracle.security.jazn.realm.RealmPermission) and its action and target parameters (for RealmPermission, realmname action). Note that there may be multiple action and target parameters.

Listing Permissions

-listperms *realm* {*user* |-role *role*| realm *realm*}

The **-listperms** option displays all permissions that match the list criteria. This option lists the following:

- All permissions registered with the JAAS Provider PermissionClassManager
- Permissions that are granted a role, when called with a realm name and the option -role

Listing Permission Information

-listperm *permission*

The **-listperm** option displays detailed information about the specified permission, including the permission's display name, class, description, actions, and targets.

Listing Principal Classes

-listprncpls

The **-listprncpls** option lists all principal classes registered with the PrincipalClassManager.

Listing Principal Class Information

-listprncpl *principal_name*

The **-listprncpl** option displays detailed information about the specified principal, including the display name, class, description, and actions.

Interactive Shell

Starting the JAZN Admintool Shell

-shell

The **-shell** option starts an JAAS provider interface shell. The JAAS Provider shell provides interactive administration of JAAS provider principals and policies through a UNIX-derived interface.

Configuration Operations

Getting XML Configuration Information

-getconfig *default_realm admin password*

The **-getconfig** option displays the current configuration setting in `jazn.xml`.

Migration Operations

Migrating Principals from the principals.xml File

-migrates *filename realm*|

The **-migrate** option migrates the OC4J `principals.xml` file into the specified realm of the current JAAS provider. *filename* specifies the name and location of the OC4J principals file (typically stored in `j2ee/home/config/principals.xml`).

The migration converts `principals.xml` users to JAAS Provider `RealmUsers` and `principals.xml` groups to JAAS Provider roles. All permissions previously granted to a `principals.xml` group are mapped to the JAAS Provider role. All users that were deactivated at the time of migration are not migrated. This is to ensure that no users can inadvertently gain access through the migration.

An error is returned if the specified file contains errors.

> **See Also:** "Replacing principals.xml" on page 5-4 for additional information on migration and replacement of `principals.xml`

Miscellaneous

Getting Help

`-help`

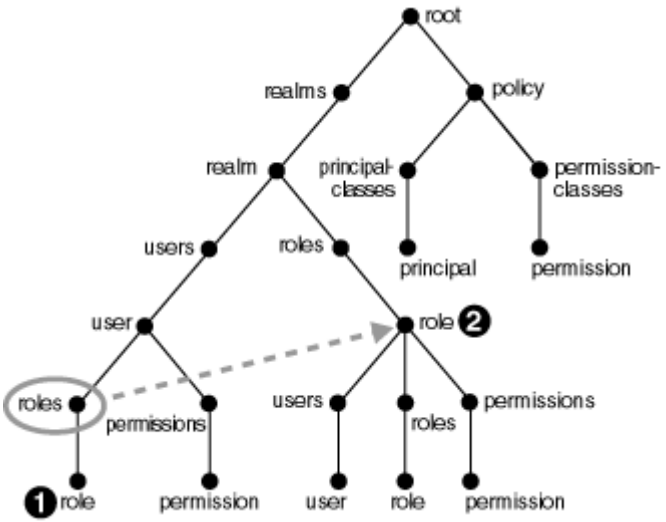
The **`-help`** option displays a list of command options available with the JAZN Admintool.

JAZN Shell Interface

The JAZN Admintool includes a shell called the JAZN shell interface. The JAZN shell provides an interactive interface to the JAAS Provider API.

The shell directory structure consists of nodes, where nodes contain subnodes that represent the parent node's properties. Figure 6–1 shows the node structure:

Figure 6–1 JAZN Shell Directory Structure

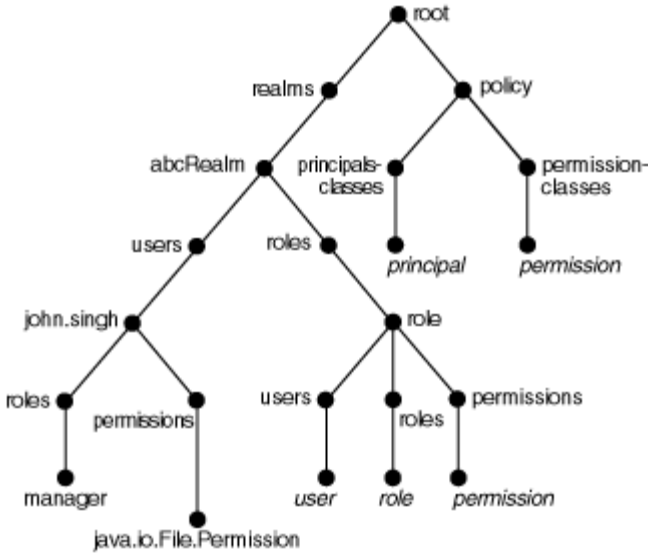


In this structure, the `user` and `role` nodes are linked together. Consequently, if you are at `/realms/realm/users/user/roles` in the tree and type `cd role`, you are taken to `/realms/realm/roles/`*`role`*.

Another way to look at this, is that `role` 1 is a symbolic link to `role` 2.

Figure 6–2 shows nodes of the `xmlRealm` created by the `jazn-data.xml` file in "Sample jazn-data.xml Code" on page B-2.

Figure 6–2 Illustrated Shell Directory Structure



The JAZN shell can be recognized by the shell prompt `JAZN:>`. At any point in time, the prompt indicates which realm the administrator is managing. The following is an example:

```
JAZN:> cd foo
JAZN:foo> ls
```

To start the shell, invoke the JAZN Admintool with the `-shell` option, as follows:

```
java -jar jazn.jar -shell
```

JAZN Shell Commands

Shell commands consists of the command options in "Realm Operations" on page 6-14 and the following series of UNIX derived commands for viewing the principals and policies in the structured way. Relative and absolute paths are supported for all relevant commands.

Using the ls Command to List JAAS Provider Data

ls [*path*]

The **ls** command mirrors its UNIX counterpart and lists the contents of the current directory or node. For example, if the current directory is the root, `ls` lists all realms. If the current directory is `/realm/users`, then `ls` lists all users in the realm. The results of the listing depends on the current directory. The `ls` command can operate with the * wildcard.

Using the cd Command to Navigate JAAS Provider Data

cd *path*

The **cd** command, mirroring its UNIX counterpart, allows users to navigate the directory tree. Relative and absolute path names are supported. To exit a directory, type `cd ...` Entering `cd /` returns the user to the root node. An error message is displayed if the specified directory does not exist.

Using the mkdir, mk, or add Commands to Create JAAS Provider Data

mkdir *directory_name* [*other_parameter*]
mk *directory_name* [*other_parameter*]
add *directory_name* [*other_parameter*]

The **mkdir**, **mk**, and **add** commands are synonyms of a command that creates a new subdirectory or node in the current directory. For example, if the current directory is the root, it creates a realm. If the current directory is `/realm/users`, it creates a user. The effect of `mkdir` depends upon the current directory. Some commands require additional parameters in addition to the name.

Using the rm Command to Remove JAAS Provider Data

rm *directory_name*

The **rm** command mirrors its UNIX counterpart and removes the directory or node in the current directory. For example, if the current directory is the root, it removes the specified realm. If the current directory is `/realm/users`, it removes the specified user. The effect of `rm` depends on the current directory. An error message is displayed if the specified directory does not exist.

The `rm` command can operate with the * wildcard.

Using the pwd Command to Display the Current Shell Working Directory

pwd

The **pwd** command displays the current location of the user through the UNIX directory format. Undefined values are left blank in this listing.

Using the help Command to List JAAS Provider Commands

help

The **help** command displays a list of all valid commands.

Using the man Command to Display Detailed JAAS Provider Commands

man *command_option*
man *shell_command*

The **man** command mirrors its UNIX counterpart and displays more detailed usage information for the specified shell command or JAZN Admintool command option. Where information presented by the man page and this document conflict, this document contains the correct usage for the command.

Using the clear Command to Clear the Screen

clear

The **clear** command clears the terminal screen by displaying 80 blank lines.

Using the exit Command to Exit the JAZN Shell

exit

The **exit** command exits the JAZN shell.

Managing LDAP Provider Data with Java Programs

You can manage JAAS provider data by creating Java programs using the JAAS Provider APIs.

This section discusses the JAAS provider in LDAP environments. The emphasis is on Java programming, but it also provides useful information for those using Oracle Enterprise Manager or the JAZN Admintool.

This section contains the following topics:

About the Sample Java Code

Some sample Java programs for managing LDAP environments are provided for you. In the example code, objects to be modified are presented in bold.

In most cases, relationships between examples are discussed after the code. The following chapters contain JAAS provider examples:

- Chapter 6, "Managing the JAAS Provider" (this chapter)
- Chapter 7, "Developing Secure J2SE Applications"
- Chapter 8, "Developing Secure J2EE Applications"
- Appendix B, "JAAS Provider Standards and Samples"

The example relationships discussed include the following:

- An example demonstrates creating a realm type, such as an Application Realm. A later example contains the code for dropping that same Application Realm.
- An example demonstrates setting permissions on a specific application. In a later section, the user granted those permissions is shown starting and running that application.

The JAZNContext and JAZNConfig Classes

The `JAZNContext` and `JAZNConfig` classes of the package `oracle.security. jazn` serve as a starting point for the JAAS provider. The `JAZNContext` and `JAZNConfig` classes contain methods such as `getPolicy`, `getProperty`, and `getRealmManager` that automatically retrieve information specific to the current JAAS provider instance.

The `JAZNConfig` class is designed for use with multiple instances of the JAAS provider.

The following code sample illustrates how `JAZNContext` or `JAZNConfig` are used in creating a realm in an LDAP-based environment:

```
RealmManager realmMgr = JAZNContext.getRealmManager();
...
realm = realmMgr.createRealm("abcRealm", realmInfo);
```

Managing Realms

After you have installed and configured the required components, you must create realms. A realm is a user community instance maintained by the authorization system. Realms consist of a user manager and role manager, and provides access to an LDAP-based provider environment of users and roles (groups).

This section contains the following topics:

- Realm Creation
- Creating an External Realm
- Creating an Application Realm
- Dropping a Realm

Realm Creation

Realms are created using the `createRealm()` method of the `RealmManager` class, which requires the following information:

- The realm name
- The role name (`adminRole`) given to the administrator. This role can then be granted to others, giving them administrative privileges
- Other properties in name/value pairs, including the location that contains the users and roles of the realm's organization in Oracle Internet Directory
- A user's searchbase property for locating the administrator and any user of the realm. This is required for External Realm and Application Realm.
- A role's searchbase property for locating the administrative role and any role for the realm. This is required for External Realm.
- Optional properties:
  - The administrator name (`adminUser`), a user with administrative privileges
  - A user object class to use as a filter to search for users
  - A role object class to use as a filter to search for roles

See Also:

- "Role-Based Access Control (RBAC)" on page 3-14
- "Realms" on page 3-10
- "JAAS Provider Realm and Policy Management" on page 3-16
- "The JAZNContext and JAZNConfig Classes" on page 6-25
- "Package oracle.security.jazn.realm" on page A-7
- "LDAP-Based Realm Types" on page 3-18 for definitions of realm types

Creating an External Realm

An External Realm is an LDAP-based realm that integrates existing user communities (user and role information not currently stored under the JAAS Provider context) with the JAAS provider.

User and role management in an External Realm must be handled by an Oracle Internet Directory tool.

The following code sample creates an External Realm with the objects shown in Table 6–3. The objects to be modified are presented in bold.

Table 6–3 Objects in Sample External Realm Creation Code

Objects	Names
sample organization	`abc.com`
`adminUser` (optional)	`John.Singh`
`adminRole`	`administrator`
sample realm name	`abcRealm`

Example 6–1 External Realm Creation Code

```
import oracle.security.jazn.spi.ldap.*;
import oracle.security.jazn.*;
import oracle.security.jazn.realm.*;

import java.util.*;

/**
 * Creates an external realm.
 */


public class CreateRealm extends Object
{
    public CreateRealm() {};

    public static void main (String[] args) {
      CreateRealm test = new CreateRealm();
      test.createExtRealm();
    }

    void createExtRealm() {
    Realm realm=null;


 try {
     Hashtable prop = new Hashtable();
     prop.put(Realm.LDAPProperty.USERS_SEARCHBASE,"cn=users,o=abc.com");
     prop.put(Realm.LDAPProperty.ROLES_SEARCHBASE,"cn=roles,o=abc.com");
```

```
      // specifying the following LDAP directory object class
      // is optional.  When specified, it will
      // be used as a filter to search for users
      prop.put(Realm.LDAPProperty.USERS_OBJ_CLASS,"orclUser");

     // adminUser is optional
      String adminUser = "John.Singh";

      String adminRole = "administrator";

      RealmManager realmMgr = JAZNContext.getRealmManager();

      InitRealmInfo realmInfo = new
           InitRealmInfo(InitRealmInfo.RealmType.EXTERNAL_REALM, adminUser,
           adminRole, prop);
      realm = realmMgr.createRealm("abcRealm", realmInfo);
      }

catch (Exception e) {
     e.printStackTrace();
    }
  }
}
```

Creating an Application Realm

An Application Realm is an LDAP-based realm that supports external read-only users and internal role management.

The code for creating an Application Realm is similar to the code for creating an External Realm, with the following exceptions:

- The property name for `InitRealmInfo.RealmType` is `APPLICATION_REALM`
- An Application Realm does not need to include the setting to search for roles as defined in `prop.put(Realm.LDAPProperty.ROLES_SEARCHBASE, "cn=roles,o=defaultOrganization");`

> **See Also:** "Supplementary Code Sample: Creating an Application Realm" on page B-8 for a complete code sample

---

Note: If both `adminUser` and `adminRole` exist, then `adminRole` is granted to `adminUser`, using RBAC.

Dropping a Realm

The `RealmManager` class of package `oracle.security.jazn.realm` enables you to drop a realm.

The following code sample shows how to drop a realm:

```
RealmManager realmMgr = JAZNContext.getRealmManager();
  realmMgr.dropRealm("abcRealm");
```

The JAAS provider administrator and the realm administrator both have permission to drop a realm.

Managing Users

You cannot create or manage users directly in the JAAS provider if you are using an LDAP-based provider type. For those tasks, use an Oracle Internet Directory tool.

You can add users to a realm using the realm's `UserManager` interface, as shown in the following code:

```
UserManager usermgr = realm.getUserManager();
RealmUser user = usermgr.getUser("Chitra.Kumar");
```

> **See Also:** *Oracle Internet Directory Administrator's Guide* for information on using Oracle Internet Directory tools

Managing Roles

The `RoleManager` interface provides methods to manage roles. Table 6–4 describes some of the methods available with the `RoleManager` interface.

Table 6–4 RoleManager Methods

Method	Description	Available to These Realms
`createRole`	Creates a role in a realm	Application Realm
`grantRole`	Grants a role to a `RealmPrincipal`	Application Realm
`dropRole`	Drops either named roles or a role given in the instance	Application Realm
`getRoles`	Gets roles in a realm	All realms
`revokeRole`	Revokes a role from a `RealmPrincipal`	Application Realm

Managing roles requires getting the realm from the `RealmManager` as described in "The JAZNContext and JAZNConfig Classes" on page 6-25. After that, you get an instance of the `RoleManager` interface with the method you are calling.

This section contains these topics:

- Creating Roles
- Granting Roles
- Dropping Roles

Note: You can internally create, grant, drop, and revoke roles in an Application Realm using the `RoleManager` interface.

However, in an External Realm, you cannot use the `RoleManager` interface. Roles can be created, granted, dropped, and revoked with an Oracle Internet Directory tool.

Creating Roles

Roles are created either externally in an External Realm with an Oracle Internet Directory tool or internally in an Application Realm with `RoleManager`.

The following code sample shows how to create a role with `RoleManager`:

```
RoleManager rolemgr = realm.getRoleManager();
RealmRole role = rolemgr.createRole("devManager_role");
```

Granting Roles

You can grant roles in an Application Realm, but not in an External Realm.

Roles are granted by an instance of `RoleManager`.

These lines show how to grant a role:

```
RoleManager rolemgr = realm.getRoleManager();
...
rolemgr.grantRole(user, director_role);
```

These lines are key to the sample code show in Example 6–2 on page 6-31.

This sample code demonstrates granting a role, `manager_role`, to another role, `director_role`, and granting the `director_role` to a user, `Chitra.Kumar`. Consequently, `Chitra` is granted the `director_role` directly, and the `manager_role` indirectly.

The objects to be modified are presented in bold.

Table 6–5 Objects in Sample Granting Roles Code

Objects	Names	Comments
`Realm`	`devRealm`	`devRealm` appears in this code and in the creation of the sample Application Realm which can be viewed in Example B–2 on page B-8.
`RealmUser user`	`Chitra.Kumar`	
`RealmRole`	`director_role`	
`RealmRole`	`manager_role`	
sample organization	`dev.com`	`dev.com` does not appear in this code directly, but was acted upon in the creation of the sample Application Realm which can be viewed in Example B–2 on page B-8.

Example 6–2 Granting Roles Code Sample

```
import oracle.security.jazn.spi.ldap.*;
import oracle.security.jazn.*;
import oracle.security.jazn.realm.*;
import java.util.*;

public class GrantRole extends Object
{
  public GrantRole() {}
  public static void main (String[] args)
  {
      GrantRole test = new GrantRole();
      test.grantRole();
  }
    void grantRole() {
  try {

      RealmManager realmMgr = JAZNContext.getRealmManager();
      Realm realm = realmMgr.getRealm("devRealm");
      RoleManager rolemgr = realm.getRoleManager();
      RealmRole manager_role = rolemgr.getRole("manager_role");
      RealmRole director_role = rolemgr.getRole("director_role");
      UserManager usermgr = realm.getUserManager();
        RealmUser user = usermgr.getUser("Chitra.Kumar");
```

```
      /* grants manager_role to director_role */
      rolemgr.grantRole( director_role, manager_role);

      /* grants director_role to Chitra */
      rolemgr.grantRole( user, director_role);
      }

  catch (JAZNException e) {
      System.out.println("Exception "+e.getMessage());

      }
    }
}
```

Dropping Roles

The following code sample shows how to drop a role with `RoleManager`:

```
RoleManager rolemgr = realm.getRoleManager();
rolemgr.dropRole("devManager_role");
```

Managing Permissions

Permissions are extended from the `java.security.Permission` class. The JAAS provider provides four classes of permissions representing types of actions that can be performed. See Table 3–2 on page 3-6 for the list of permissions.

Permissions are all created with constructors such as the following `RealmPermission`:

```
RealmPermission Perm1 = new RealmPermission("devRealm", "createRole");
```

See Also: The following for further information on permissions:

- "What Is the Java2 Security Model?" on page 3-4
- "What Is the Java2 Security Model?" on page 3-4
- Java Security documentation by visiting the following URL: `http://java.sun.com/j2se/1.3/docs/guide/security/`

Managing JAAS Provider Policy

JAAS provider policy grants permissions to principals, such as users and roles. The policy can be modified after initialization to grant and revoke permissions to grantees.

Managing Policy with JAAS Provider Packages

These lines of code are key to the sample class shown in "Modifying User Permissions Code" on page B-10.

```
final JAZNPolicy policy = JAZNContext.getPolicy();
...
policy.grant(new Grantee(propset, cs), new
     FilePermission("report.data", "read"));
```

Managing XML-Based Provider Data with the XML Schema

You can manage JAAS provider data by modifying XML files used by the JAAS Provider APIs.

This section discusses the JAAS provider in XML-based provider environments. The emphasis is on data files that you create yourself based on the XML schema, but it also provides useful information for those using the JAZN Admintool.

The XML-based environment provides fast, simple, lightweight JAAS provider management. You can use an XML file (named `jazn-data.xml` in this example) to manage the JAAS provider realm and policy information. Table 6–6 describes the sections of the `jazn-data.xml` file.

Table 6–6 Description of jazn-data.xml File

Section	This section enables you to:
Realm data	■ Create realms, users, and roles ■ Grant roles to users and to other roles
Policy data	Assign permissions to users and roles defined in the realm data section of the file

The `jazn-data.xml` file is specified as follows:

- For J2SE: in the `jazn.xml` configuration file
- For J2EE: in the `orion-application.xml` configuration file

See Also: *Oracle9i Application Server Security Guide* for configuration information on these two XML files

Managing Realms, Users, Roles, and Permissions

XML realm and provider information is stored in an XML file typically named `jazn-data.xml`. To work correctly, the XML file must conform to specific policy schema and DTD standards.

See Also:

- "Sample jazn-data.xml Code" on page B-2 to view an XML Schema and a sample `jazn-data.xml` file

DTD for jazn-data.xml

The JAAS provider data file must conform to the following DTD:

```
<!ELEMENT jazn-data (jazn-realm?, jazn-policy?, jazn-permission-classes?,
jazn-principal-classes?, jazn-loginconfig?)>

<!-- Realm Data -->

<!ELEMENT jazn-realm (realm*)>
<!ELEMENT realm (name, users?, roles?, jazn-policy?)>
<!ELEMENT users (user*)>
<!ELEMENT user (name, display-name?, description?, credentials?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT display-name (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT credentials (#PCDATA)>
<!ELEMENT roles (role*)>
<!ELEMENT role (name, display-name?, description?, members)>
<!ELEMENT members (member*)>
<!ELEMENT member (type, name)>
<!ELEMENT type (#PCDATA)>

<!-- Policy Data -->

<!ELEMENT jazn-policy (grant*)>
<!ELEMENT grant (grantee, permissions?)>
<!ELEMENT grantee (display-name?, principals?, codesource?)>
<!ELEMENT principals (principal*)>
<!ELEMENT principal (realm-name?, type?, class, name)>
<!ELEMENT realm-name (#PCDATA)>
```

```
<!ELEMENT codesource (url)>
<!ELEMENT url (#PCDATA)>
<!ELEMENT permissions (permission+)>
<!ELEMENT permission (class, name, actions?)>
<!ELEMENT class (#PCDATA)>
<!ELEMENT actions (#PCDATA)>

<!-- Principal Class Data -->

<!ELEMENT jazn-principal-classes (principal-class*)>
<!ELEMENT principal-class (name, description?, type, class,
name-description-map?)>
<!ELEMENT name-description-map (name-description-pair*)>
<!ELEMENT name-description-pair (name, description?)>

<!-- Permission Class Data -->

<!ELEMENT jazn-permission-classes (permission-class*)>
<!ELEMENT permission-class (name, description?, type, class, target-descriptors,
action-descriptors?)>
<!ELEMENT target-descriptors (target-descriptor*)>
<!ELEMENT target-descriptor (name, description?)>
<!ELEMENT action-descriptors (action-descriptor*)>
<!ELEMENT action-descriptor (name, description?)>

<!-- Login Module Data -->

<!ELEMENT jazn-loginconfig (application*)>
<!ELEMENT application (name, login-modules)>
<!ELEMENT login-modules (login-module+)>
<!ELEMENT login-module (class, control-flag, options?)>
<!ELEMENT control-flag (#PCDATA)>
<!ELEMENT options (option+)>
<!ELEMENT option (name, value)>
<!ELEMENT value (#PCDATA)>
```

Other Utilities

There are three additional utilities for managing the JAAS provider. These classes work with both LDAP-based and XML-based provider types. The classes can be used and managed programmatically. Additionally, two can be managed through the JAZN Admintool.

- `PermissionClassManager` - Integrates with the JAZN Admintool
- `PrincipalClassManager` - Integrates with the JAZN Admintool
- `LoginModuleManager` - Works only with J2EE applications and is not activated with the JAZN Admintool

PermissionClassManager Interface

The `PermissionClassManager` is a repository of all registered Permission classes and a utility to help manage them. Registering a permission class allows access to stored metadata that provides specific information about a given permission's target, action, and / or description. Failure to register a given permission class does not affect the JAAS provider's ability to use the permission class. That is, the JAAS provider does not limit permission grants or revocations to those classes registered with the `PermissionClassManager`.

Works with the JAZN Admintool to perform these functions:

- "Adding and Removing Permissions" on page 6-17
- "Listing Permissions" on page 6-18

See Also:

- "PermissionClassManager" on page A-5 to view the API

PrincipalClassManager Interface

`PrincipalClassManager` represents the repository of all registered Principal classes and a utility to help manage them. Registering a principal class allows access to stored metadata that provides specific information about a given principal's name and description. Failure to register a given principal class will not affect the JAAS provider's ability to use the principal class. That is, the JAAS provider recognizes all principal classes whether or not they've been registered with the `PrincipalClassManager`.

The `PrincipalClassManager` works with the JAZN Admintool to perform these functions:

- "Adding and Removing Principals" on page 6-17
- "Listing Principal Classes" on page 6-18

See Also:

- "PrincipalClassManager" on page A-6 to view the API

LoginModuleManager

`LoginModuleManager` is the JAAS Provider implementation of the JAAS Configuration class and provides login configuration support to applications. The Configuration class is a registry of applications and corresponding login modules used by a given application and the order they are to be used. There are both `LDAPLoginModuleManager` and `XMLLoginModuleManager` implementations of the `LoginModuleManager`.

7

Developing Secure J2SE Applications

This chapter describes how to develop secure Java2 Platform, Standard Edition (J2SE) applications using the Oracle9*i*AS Containers for J2EE (OC4J) JAAS Provider.

This chapter contains these topics:

- Developing Secure J2SE Applications Overview
- Authentication in the J2SE Environment
- Authorization in the J2SE Environment
- Testing and Executing an Application
- Sample J2SE Application

> **Note:** This chapter assumes that you have followed the management instructions in Chapter 7, "Developing Secure J2SE Applications".

Developing Secure J2SE Applications Overview

J2SE application developers develop, deploy, and manage Java applications on local desktops or servers. Using the JAAS provider enables developers to make these applications secure.

After the creation of realms and related components described in Chapter 5, "Integrating the JAAS Provider with Java2 Applications", the JAAS provider can be integrated into J2SE applications to provide the following services:

- Authentication in the J2SE Environment
- Authorization in the J2SE Environment

See Also:

- "JAAS Provider Integration in J2SE Application Environments" on page 5-2
- "Sample J2SE Application" on page 7-5 for a J2SE application demonstration

Authentication in the J2SE Environment

Authentication is the process of verifying the identity of a user in a computing system, often as a prerequisite to granting access to resources in a system. User authentication in the J2SE environment is performed with the following:

- A JAAS `LoginContext` class
- A JAAS Provider `RealmLoginModule` class or another login module that can be configured as the default login module
- A callback handler that you must create, following the JAAS model in `javax.security.auth.callback`

The constructor for the `LoginContext` class requires the name of the client login and a new instance of a callback handler, an object you must implement. The callback handlers, which are described in JAAS documentation, are required by the login module to communicate with users.

The user of the computing service is the `Subject`. The `Subject` is passed to the `LoginContext` class. The `LoginContext.login()` method compares the `Subject` to configuration settings in the JAAS Provider `RealmLoginModule` or other login module. If `login()` is successful, the login module associates the `Principal` (a specific identity) and credentials with the `Subject`.

This authenticates the `Subject`, which can then be retrieved by invoking `LoginContext.getSubject` in the authorization process.

See Also: JAAS documentation at the following Web site for more information about authentication, login modules, and callback handlers:

`http://java.sun.com/products/jaas/`

Authorization in the J2SE Environment

Once a user is successfully authenticated, the authorization policy is enforced upon the user. Authorization is achieved through the following methods and interface based on the Java2 and JAAS Security Model:

- `javax.security.auth.Subject.doAs()` method in the client
- `java.lang.SecurityManager.checkPermission` method in the server
- The `PrivilegedAction` interface of `java.security` in the application

Subject.doAs

After retrieving the authenticated `Subject` from the `LoginContext`, the client invokes `Subject.doAs` with the application as a parameter. The application starts, which activates security checking in the server. An `AccessControlException` is thrown if security checking fails.

SecurityManager.checkPermission

Security checking in J2SE applications requires the use of the JDK 1.3 or greater `java.lang.SecurityManager` in the server.

The security manager determines whether to permit operations. The classes in Java packages cooperate with the security manager by asking the application's security manager for permission to perform certain operations. Each Java application can have its own security manager object that acts as a full-time security guard.

The `SecurityManager.checkPermission` method performs security checking.

PrivilegedAction

The application must implement the interface `PrivilegedAction`.

See Also: Java security architecture at the following Web site:

`http://java.sun.com/j2se/1.3/docs/guide/security/`

Testing and Executing an Application

In order to test or execute the application, you must start the `SecurityManager` at the command line and, if using a login module to start an application, call it.

This is the first real test of the JAAS provider.

Starting with RealmLoginModule

To start the application using the RealmLoginModule:

1. Go to the computer on which the J2SE application is installed.
2. Start the security manager and test the application at the command prompt:

   ```
   java -Djava.security.manager -Djava.security.policy=java2.policy
   -Djava.security.auth.policy=jazn.xml
   -Djava.security.auth.login.config=jaas.config MyApp
   ```

   where the client, `MyApp`, calls your application. The `jazn.xml` file is the property file that identifies the provider type you are using (Oracle Internet Directory or XML-Based Provider Type). The `jaas.config` file indicates that `RealmLoginModule` is required for authentication.

   This command can be used with the sample code shown in "Sample J2SE Application" on page 7-5.

Starting without RealmLoginModule

It is possible to start J2SE applications without using authentication and the `RealmLoginModule` or any login module, but that is not the preferred method. To do so and use the sample code provided in this chapter, you need to modify the `MyApp` code in Example 7–1, "Client Login Code" on page 7-5 so that it does not require the objects described in "Authentication in the J2SE Environment" on page 7-2.

After you have modified the `MyApp` code, you can start it.

To start the application without using the RealmLoginModule:

1. Go to the computer that the J2SE application is installed on.
2. Start the security manager and execute the application at the command prompt:

   For example, to test a sample application, enter:

   ```
   java -Djava.security.manager -Djava.security.policy=java2.policy
   -Djava.security.auth.policy=jazn.xml MyApp
   ```

where the client, MyApp, calls your application. The type of JAAS provider you are using (LDAP-based or XML-based) is identified in the jazn.xml file.

Sample J2SE Application

This section shows a sample client login, MyApp, and a brief test application using the JAAS provider in a J2SE environment.

Table 7–1 Sample Client Login Code

Objects	Names	Comments
CallbackHandler	myCallbackHandler	myCallbackHandler is a callback handler that you must implement.
sample application	AccessTest1	AccessTest1 is the application that the user wants to start. The code for AccessTest1 is show in Example 7–2 on page 7-7.
sample external realm	abcRealm	abcRealm was created in Example 6–1 on page 6-27.
client user	Jane.Smith or unknown	The client user added in Example 6–1 on page 6-27. Since Jane.Smith is the only user added; that is, the only name returned to Principal p.

The following is executed using the commands described in "Testing and Executing an Application" on page 7-4.

Example 7–1 Client Login Code

MyApp Code

```
import java.io.*;
import java.util.*;
import java.security.Principal;
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import javax.security.auth.spi.*;
import com.sun.security.auth.*;

import oracle.security.jazn.*;
import oracle.security.jazn.realm.*;
```

```
public class MyApp {

    public static void main(String[] args) {

  LoginContext lc = null;
  try {
      // you must create a CallbackHandler class
    lc = new LoginContext("MyApp", new myCallbackHandler());
      } catch (LoginException le) {
      le.printStackTrace();
      System.exit(-1);
  }

  try {
      // attempt authentication
      lc.login();
  } catch (AccountExpiredException aee) {
      System.out.println("Your account has expired.  " +
             "Please notify your administrator.");
      System.exit(-1);

 // other exceptions
 //CredentialExpiredException
   // FailedLoginException
  }
  // checking what Principals the user has
  Iterator principalIterator = lc.getSubject().getPrincipals().iterator();
  System.out.println("Authenticated user has the following Principals:");
  while (principalIterator.hasNext()) {
      Principal p = (Principal)principalIterator.next();
      System.out.println("\t" + p.toString());
  }
  System.out.println("User has " +
         lc.getSubject().getPublicCredentials().size() +
         " Public Credential(s)");

  // now try to execute the sample application as the authenticated Subject
  Subject.doAs(lc.getSubject(), new AccessTest1());

  System.exit(0);
    }
}
```

Sample J2SE Application Code

This is the sample application that is executed when a successfully authenticated principal runs MyApp.

Table 7–2 Objects in Sample Application Code

Objects	Names
file	report.data

Example 7–2 Sample Application Code

```
import java.lang.*;
import java.security.*;
import java.io.*;

public class AccessTest1 implements PrivilegedAction {

    public Object run() {

        File f = new File("report.data");

        // Security checking is invoked
        if (f.exists()) {
            System.out.println("*** report.data accessed ***");
        }
        return null;
    }
}
```

Discussion of the J2SE Sample Client Login and Application Code

In the MyApp client, once the authentication process is completed, Subject.doAs starts the sample application AccessTest1.

AccessTest1 starts and requests to read the report.data file. This request invokes security checking in the server, which determines if the user has permission on AccessTest1 to read the report.data file.

Permission has been granted previously to Jane.Smith in Example 6–1 on page 6-27. If Jane.Smith is the user logging in, AccessTest1 runs.

If the user is not Jane.Smith, the authorization fails because no other users have been granted this permission.

8

Developing Secure J2EE Applications

This chapter describes how to develop secure Java2 Platform, Enterprise Edition (J2EE) applications using the JAAS Provider and Oracle9*i*AS Containers for J2EE (OC4J).

This chapter contains these topics:

> **Note:** This chapter assumes that you have followed the management instructions in Chapter 6, "Managing the JAAS Provider".

Developing Secure J2EE Applications Overview

J2EE application developers develop, deploy, and manage Web enabled, server-centric, enterprise level Java applications that are deployed in multiple tier environments. Using the JAAS provider enables developers to make these applications secure.

In J2EE applications, the JAAS provider is integrated with OC4J and provides the `JAZNUserManager`, an implementation of the OC4J `UserManager`.

After the creation of realms and related components described in Chapter 6, "Managing the JAAS Provider", the JAAS Provider can be integrated into J2EE applications to provide the following services:

- Authentication in the J2EE Environment
- Authorization in the J2EE Environment

See Also: "Oracle9iAS Containers for J2EE (OC4J)" on page 5-3

Authentication in the J2EE Environment

Authentication is the process of verifying the identity of a user in a computing system, often as a prerequisite to granting access to resources in a system. User authentication in the J2EE environment is performed with the following:

- Oracle9*i*AS Single Sign-On (for SSO environments) or the JAAS provider `RealmLoginModule` or other login module (for non-SSO environments)
- `JAZNUserManager` for OC4J (Required)

Before HTTP requests can be dispatched to the target servlet, the `JAZNUserManager` gets the authenticated user information (set by mod_osso) from the HTTP request object and sets the JAAS subject in OC4J.

Running with an Authenticated Identity

You can choose to configure the `JAZNUserManager` so that a filter enables the target servlet to run with the permissions and roles associated with an authenticated identity or run-as identify. To do this, configure the `jazn-web-app` element.

See Also: Chapter 7 of the *Oracle9i Application Server Security Guide* and "JAZNUserManager" on page 5-4 for further information on options and configuration of the `JAZNUserManager` filter, including the `jazn-web-app` element

Intercepting Servlet Invocation

The `JAZNUserManager` intercepts calls from Oracle9*i*AS Single Sign-On or the JAAS Provider `RealmLoginModule` and retrieves authentication information to identify the username and role.

Retrieving Authentication Information

The following `javax.servlet.HttpServletRequest` APIs retrieve authentication information within the servlet:

- `getRemoteUser` for the authenticated username
- `getAuthType` for the authentication scheme
- `getUserPrincipal` for the authenticated principal object
- `getAttribute("java.security.cert.X509certificate")` for the SSL client certificate.

(Optional if the Filter Element Has Been Set)

If the filter element has been set, `JAZNUserManager` performs the following when `doFilter(ServletRequest request, ServletResponse response, FilterChain chain)` is invoked:

- For SSO or Basic authentication, the filter relies on `JAZNUserManager` to retrieve the authenticated user and the corresponding principal object.
- For an SSL client certificate, the filter performs the following:
  1. Retrieves SSL client certificate from the request object, if it is available
  2. Instantiates `java.security.cert.X509Certificate` object `x509cert` based on the client certificate
  3. Creates an array of type `java.security.cert.X509Certificate` and adds objects to the array
  4. Sets the attribute on the request object `("java.security.cert.X509Certificate",x509cert)`
  5. Gets the SSL principal name by invoking `oracle.security.jazn.util.CertHash.getHash(x509cert)`
  6. Gets the SSL principal object `sslPrincipal`, a `RealmPrincipal` object, from the default realm using the JAAS Provider API

The filter element constructs an `oracle.security.jazn.oc4j.JAZNServletRequest` *`request`* for the HTTP request.

(End of Optional Section)

Authorization begins with a call to `Subject.doAs()`.

Authorization in the J2EE Environment

Authorization is the process of granting the permissions and privileges entitled to the user.

Once the user is authenticated, the `JAZNUserManager` invokes the target servlet within a `Subject.doAs()` block to enable JAAS-based authorization in the target servlets.

Authorization is achieved through the following:

- `JAZNUserManager`
- Methods based on the Java2 Security Model:
  - `Servlet.service()` in the servlet
  - `Subject.doAs()`in the client
  - `SecurityManager.checkPermission()`in the server

Testing and Executing the J2EE Application

After completing all configuration tasks, follow these steps to test or execute the JAAS Provider within OC4J. These steps assume the following:

- The current directory is `$ORACLE_HOME/j2ee/home`
- `mod_oc4j` is configured

To build and configure your application, a sample application, `callerInfo`, has been provided. Chapter 4, "Quick Start JAAS Provider Demo" describes how to quickly run this sample application. This chapter elaborates on the information in Chapter 4 and discusses available configuration options.

> **See Also:** *Chapter 7 of the Oracle9i Application Server Security Guide* for detailed configuration information

Setting Up

You must perform the following tasks to test and run a J2EE application:

- Task 1: Installing Ant (Optional)
- Task 2: Modifying OC4J Files
- Task 3: Changing Default Configurations

Task 1: Installing Ant (Optional)

You can install Ant, an XML-based build tool (similar to make), from Apache's Jakarta Project or plan to use jar directly. If you do not have Ant installed, you can download it from:

`http://jakarta.apache.org/ant/index.html`

Once you have installed Ant, and before running it, you must configure files as described in the next section, "Task 2: Modifying OC4J Files".

Task 2: Modifying OC4J Files

In order to run a servlet, you need to modify several OC4J Files.

Modifying OC4J Files Where OC4J is Not Running

- Modify the OC4J `server.xml` file in `$ORACLE_HOME/j2ee/home/config/` by adding the following line:

```
<application name="myApp1" path="../jazn/demo/myApp1/
   myApp1.ear" />
```

For the `callerInfo` demo, the line is as follows:

```
<application name="callerInfo" path="../jazn/demo/callerInfo/
   callerInfo.ear" />
```

- Modify the OC4J `default-web-site.xml` file in `$ORACLE_HOME/j2ee/home/ config/` by adding the following line:

```
<web-app application="myApp1" name="myApp1-web" root="/jazn" />
```

For the `callerInfo` demo, the line is as follows:

```
<web-app application="callerInfo" name="callerInfo-web" root="/jazn" />
```

Deploying an Application When the OC4J Server Is Running

If the OC4J server is already up and running, you can use Enterprise Manager to deploy your application; see the *Oracle9iAS Containers for J2EE User's Guide* for details.

For the `callerInfo` demo, specify the following information in the deployment wizard:

- **EAR file:** $J2EE_HOME/jazn/demo/callerInfo/callerInfo.ear
- **Application name:** callerInfo
- **Servlet root context:** /jazn

See Also:

- *Oracle9iAS Containers for J2EE User's Guide* for further information on OC4J configuration
- Chapter 7 of the *Oracle9i Application Server Security Guide* for further information on JAAS Provider configuration

Task 3: Changing Default Configurations

The default realm is set to `sample_subrealm`. To change to another realm, you must modify the `jazn` element of the OC4J `orion-application.xml` (in the directory `jazn/demo/callerinfo/etc/`) as follows:

Using XML-Based Realms (Default)

- Change the realm, `default-realm`, from the default value, `sample_subrealm`, to any realm that you have created.
- Change `location` from the default value, `jazn-data.xml`, to any properly configured data file that you have created. Conversely, you can also use `jazn-data.xml` as a template for your own file.

See Also: "Managing XML-Based Provider Data with the XML Schema" on page 6-33 for further information on the `jazn-data.xml` file

Using LDAP-Based Realms

Since the installation defaults to the XML-based provider type, you need to modify certain files if you are using the LDAP provider type environment.

Note: You must use the Oracle9*i*AS Infrastructure installation type if you use the LDAP provider type environment.

In the `orion-application.xml file` in directory `jazn/demo/callerinfo/etc/`, make the following changes:

- Change the JAAS Provider type to LDAP.
- Enter your LDAP `location` URL (for example, `ldap://myoid.us.oracle.com`)

Using SSL and SSO Integration

If you are using SSO or SSL integration, make the following addition to the `mod_oc4j.conf` file to add redirection information.

```
Oc4jMount /jazn/* ajp13_worker
Oc4jMount /jazn ajp13_worker
```

Assuming that `ajp13_worker` is a defined worker in the `oc4j.conf` file, this directs any request matching `/jazn/*` to be handled by `ajp13_worker`. Any request matching `/jazn/` is to be handled by `ajp13_worker.`

Using SSO

If you are using SSO integration, make the following change in the `orion-web.xml`:

Set the `auth-method` in the `jazn-web-app` element file to "SSO"as in the following example:

```
<jazn-web-app
    auth-method="SSO" (optional - default to null)
    runas-mode="false" (optional - default to false)
    doasprivileged-mode="true" (optional - default to true)
/>
```

Task 4: Building the Directory

To build the directory, either use jar or Ant to create a new directory (`build`) containing the `.ear` and `.war` files for your application.

To build the directory using Ant:

1. Open a command line shell.
2. Go to the `jazn/myApp1/myApp1` directory

   For the `callerInfo` demo, go to `jazn/demo/callerInfo` directory,
3. Type: `ant`

Starting an Application

This is the first real JAAS provider test.

To start your application:

1. Start the Oracle HTTP Server listener as follows:
   - for `mod_osso` (SSO environments), enter `apachectl start`
   - for `mod_ossl` (SSL environments) `apachectl startssl`

   Note: Skip this step if you are using Basic Authentication.
2. Start OC4J with the JAAS provider by entering the following:

   ```
   java -jar oc4j.jar
   ```

   Or start OC4J with the JAAS provider in secure mode (assuming that you have configured your `java2.policy`) with the `SecurityManager:`

   ```
   java -Djava.security.manager.
   -Djava.security.policy=/jazn/config/java2.policy -jar oc4j.jar
   ```
3. Run the servlet from a Web browser using:

   `http://hostname:1234/myApp1/myApp1`

   Or to run the sample application, use:

   `http://hostname:1234/jazn/callerInfo`

   where *1234* is the port configured for your HTTP listener.

   See Also: *Oracle9iAS Containers for J2EE User's Guide*

Sample J2EE Application

This sections shows the sample J2EE application, `callerInfo`, which you can run using the commands described in "Testing and Executing the J2EE Application" on page 8-4 or in Chapter 4, "Quick Start JAAS Provider Demo".

Sample J2EE Application callerInfo

```
package oracle.security.jazn.samples.http;

import java.io.IOException;
import java.util.Date;
import java.util.Properties;
import javax.naming.*;
import javax.servlet.*;
import javax.servlet.http.*;

/**
 * A simple demo that exercises the Servlet security APIs.
 *
 * @author rkng
 */
public class CallerInfo extends HttpServlet {

    public CallerInfo()
    {
  super();
    }
    public void init(ServletConfig config)
  throws ServletException
    {
  super.init(config);
    }
    public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException
    {
  ServletOutputStream out = response.getOutputStream();

  response.setContentType("text/html");
  out.println("<HTML><BODY bgcolor=\"#FFFFFF\">");
  out.println("Time stamp: " + new Date().toString());
  out.println("request.getRemoteUser = " + request.getRemoteUser() + "<br>");
  out.println("request.isUserInRole('FOO') = " + request.isUserInRole("FOO")
    + "<br>");
  out.println("request.isUserInRole('ar_manager') = " +
```

```
request.isUserInRole("ar_manager") + "<br>");
  out.println("request.isUserInRole('ar_developer') = " +
request.isUserInRole("ar_developer") + "<br>");
  out.println("request.getUserPrincipal = " + request.getUserPrincipal() +
"<br>");
  out.println("</BODY>");
  out.println("</HTML>");
    }
}
```

Discussion of the J2EE Sample Application Code

When the call to `callerInfo` is successful, the browser displays a message similar to the following:

```
Time stamp: Fri Aug 24 19:11:37 PDT 2001 request.getRemoteUser =
sample_subrealm/user
request.isUserInRole('FOO') = false
request.isUserInRole('ar_manager') = false
request.isUserInRole('ar_developer') = true
request.getUserPrincipal = ([JAZNUserAdaptor: user=[XMLRealmUser:
sample_subrealm/user])
```

9

Java Message Service

This chapter describes the Java Message Service (JMS) furnished as part of Oracle9*i*AS Containers for J2EE (OC4J). This chapter discusses the following topics:

Overview

Java clients and Java middle-tier services must be capable of using enterprise messaging systems. JMS offers a common way for Java programs to access these systems. JMS is the standard messaging API for passing data asynchronously between application components, allowing business integration in heterogeneous and legacy environments. JMS provides two programming models:

- **Point-to-Point (Queue)** —Messages are sent to one consumer only.
- **Publish and Subscribe (Topics) —**Messages are broadcast to all registered listeners.

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

OC4J provides a `ResourceProvider` interface to transparently plug in third-party JMS implementations. The JMS resources are available under `java:comp/resource` through the resource provider interface delegation.

Resource Providers

The `ResourceProvider` interface enables you to plug in third-party message providers for JMS connections. For Oracle JMS, this allows EJBs, servlets, and OC4J clients to access many different queue implementations. With third-party message providers, only EJBS can access queue implementations. The resources are available under `java:comp/resource/` as the default JMS resources.

Configuring a Custom Resource Provider

To add a custom `<resource-provider>`, add the following to your `orion-application.xml` file:

```
<resource-provider class="providerClassName" name="JNDI name">
     <description>
         description
     </description>
     <property name="name" value="value" />
</resource-provider>
```

In place of the user-replaceable constructs (those in italics) in the preceding code, do the following:

- Replace the value *`providerClassName`* of the `class` attribute with the name of the resource-provider class.
- Replace the value *`JNDI name`* of the `name` attribute with a name by which to identify the resource provider. This name will be used in finding the resource provider in the application's JNDI as `"java:comp/resource/name/"`.
- Replace the value *`description`* of the `description` tag with a description of the specific resource provider.
- Replace the values *`name`* and *`value`* of the corresponding attributes with the same name in any `property` tags that the specific resource provider needs to be given as parameters.

Using a Custom Resource Provider

Use the following lookup syntax to retrieve a resource provider's resources:

`java:comp/resource/`*`providerName/resourceName`*
Where *`providerName`* is the name of the resource provider (as given in the attribute `name` described in the previous section) and *`resourceName`* is the name of a resource this resource provider furnishes.

Using Oracle JMS as a Resource Provider

The `ResourceProvider` interface allows you to plug in Oracle JMS, which enables J2EE code (EJBs, MDBs, JSPs, servlets, application clients, and so on) to access Oracle AQ.

To access Oracle JMS queues through JMS, you must do the following:

1. Create an RDBMS user through which the JMS application will connect to the back-end database. The user must have the necessary privileges to perform Oracle JMS operations. Oracle JMS allows any database user to access queues in any schema, provided the user has the appropriate access privileges.
2. Configure an OC4J resource provider with information about the back-end database. Create data sources or LDAP directory entries, if needed.
3. Access the resource using Oracle JMS resource names, which include the `ResourceName` name component.

Note: For the OC4J 9.0.3 implementation, MDB is integrated with Oracle JMS only through the resource provider interface.

Note: Oracle JMS implements the JMS 1.0.2 specifications and complies with J2EE 1.3.

Configuring the Resource Provider

Identify the JNDI name of the data source to use as the resource provider within the `<resource-provider>` element.

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

The following is an example of how to configure the resource provider using XML syntax for Oracle JMS.

- `class` attribute—The `oracle.jms.OjmsContext` class, which is configured in the `class` attribute, implements the Oracle JMS resource provider.

- `property` attribute—Identify the data source that is to be used as this resource provider in the `property` element. The topic or queue connects to this data source to access the tables and queues that facilitate the messaging.

```
<resource-provider class="oracle.jms.OjmsContext" name="cartojms1">
  <description> OJMS/AQ </description>
  <property name="datasource" value="jdbc/CartEmulatedDS"></property>
</resource-provider>+
```

For details on configuring data sources, see "Defining Data Sources" on page 11-2.

Using Message-Driven Beans

The OC4J message-driven beans (MDB: EJBs that process JMS messages asynchronously) are integrated only with Oracle JMS, through the resource provider interface. MDBs are not integrated with third-party message providers.

An MDB is a JMS message listener that can reliably consume messages from a queue or a subscription of a topic. The advantage of using an MDB instead of a JMS message listener is that you can use the asynchronous nature of a JMS listener with the following EJB container advantages:

- The consumer is created for the listener. That is, the container creates the appropriate `QueueReceiver` or `TopicSubscriber`.
- The MDB is registered with the consumer. The container registers the MDB with the `QueueReceiver` or `TopicSubscriber` and its factory at deployment time.
- The message acknowledgment mode is specified.

Refer to the MDB chapter for details on deploying an MDB accessing Oracle JMS through the resource provider interface.

Download the MDB example from the OC4J sample code page :

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

Note: Message-driven beans are supported only for Oracle JMS.

Using Third-Party Resource Providers

> **Note:** For 9.0.3, OC4J provides only a very limited set of operations, as described in this section, for calling out to third-party message providers through the resource provider interface for JMS applications.

This section discusses the following third-party resource providers:

- MQSeries
- SonicMQ
- SwiftMQ

Here are the operations that the resource provider interface supports:

- Look up queue and topic with `java:comp/resource/`.
- Send a message in EJB.
- Receive a message synchronously in EJB.

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

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

Using MQSeries as a Resource Provider

The Resource Provider interface provides support for plugging in third-party JMS implementations. This example demonstrates how to make MQSeries the default Resource Provider for JMS connections. The MQSeries resources are available in OC4J under `java:comp/resource/MQSeries/`.

Configuring

1. Install and configure MQSeries on your system, then verify the installation by running any examples or tools supplied by the vendor. (See the documentation supplied with your software for instructions.)

2. Use the `<resource-provider>` tag in `orion-application.xml` to add MQSeries as a custom Resource Provider. Here is an example of using this tag for SonicMQ integration:

```
<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:/var/mqm/JNDI-Directory">
  </property>
</resource-provider>
```

3. Add the following MQSeries JMS client jar files to `$J2EE_HOME/lib`:

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

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

Using SonicMQ as a Resource Provider

SonicMQ is a messaging broker with a complete implementation of the JMS 1.0.2 specification. The resource provider interface furnishes support for plugging in third-party JMS implementations. This example describes how to make SonicMQ the default resource provider for JMS connections. The SonicMQ resources are available in OC4J under `java:comp/resource/SonicMQ`.

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

---

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 with your software for instructions.)
2. Use the `<resource-provider>` tag in `orion-application.xml` to add SonicMQ as a custom resource provider. The following example demonstrates using SonicMQ as the message provider and the file system as the JNDI store:

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

3. Add the SonicMQ JMS client JAR files, `Sonic_client.jar` and `Sonic_XA.jar`, to `$J2EE_HOME/lib`.

Using SwiftMQ as a Resource Provider

SwiftMQ is a messaging broker with a complete implementation of the JMS 1.0.1 specification. The Resource Provider interface furnishes support for plugging in third-party JMS implementations. This example describes how to make SwiftMQ the default ResourceProvider for JMS connections. The SwiftMQ resources are available in OC4J under `java:comp/resource/SwiftMQ`.

1. Install and configure SwiftMQ on your system, then verify the installation by running any examples or tools supplied by the vendor. (See the documentation provided with your software for instructions.)
2. Use the `<resource-provider>` tag in `orion-application.xml` to add SwiftMQ as a custom resource provider, as shown in the following:

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

3. Add the SwiftMQ JMS JAR file `swiftmq.jar` to `$J2EE_HOME/lib`.

10

Interoperability and RMI Tunneling

This chapter describes OC4J support for cross-platform distributed EJB interoperation and for using RMI over HTTP (RMI tunneling).

This chapter covers the following topics:

Introduction to EJB Interoperability

Version 2.0 of the Enterprise Java Beans specification adds features that make it easy for EJB-based applications to invoke one another across different containers. You can make your existing EJB interoperable without changing a line of code: simply edit the bean's properties and redeploy. Redeployment details are discussed in "Simple Interoperability" on page 10-3.

EJB interoperability consists of the following:

- **Transport interoperability** through CORBA IIOP
- **Naming interoperability** through the CORBA CosNaming Service
- **Security interoperability** through Common Secure Interoperability Version 2 (CSIv2)
- **Transaction interoperability** through the CORBA Transaction Service (OTS)

OC4J provides all these features.

Naming

OC4J supports the CORBA CosNaming service. OC4J can publish `EJBHome` object references in a CosNaming service. OC4J 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 CSIv2. CSIv2 specifies different conformance levels; OC4J complies with the EJB specification, which requires conformance level 0.

Transactions

The EJB2.0 specification specifies an optional transactional interoperability feature. Conformant implementations must choose one of the following:

- Transactionally interoperable—transactions are supported between beans hosted in different J2EE containers
- Transactionally non-interoperable—transactions are supported only among beans in the same container

This release of OC4J is transactionally non-interoperable. This means that when a transaction spans EJB containers, OC4J raises a specified exception.

Switching to Interoperable Transport

In OC4J, EJBs use RMI/ORMI, a proprietary protocol, to communicate. It is easy to convert an EJB to using RMI/IIOP; this makes it possible for EJBs to invoke one another across EJB containers.

Note: RMI/IIOP support is based on CORBA 2.3.1. Applications compiled using earlier releases of CORBA may not work correctly.

Simple Interoperability

Follow these steps:

1. Restart OC4J with the `-DGenerateIIOP=true` flag.
2. Edit the client's JNDI property `java.naming.provider.url` to use a `corbaname` URL instead of an `ormi` URL. For details on the `corbaname` URL, see "The corbaname URL" on page 10-4.
3. (Client only) Change the client's `classpath` to include the stub JAR file generated by OC4J. This will normally be

   application_deployment_directory/appname/ejb-module/module`_iiopClient.jar`

   If you do not have access to the deployment directory, you can obtain the generated stub JAR file during deployment by running `admin.jar` with the `-iiopClientJar` switch

   > **Note:** IIOP stub and tie class code generation happens at deployment time, unlike ORMI stub generation which happens 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.

4. (Optional) To make the bean accessible to CORBA applications, run `rmic.jar` to generate IDL describing its interfaces. See "Configuring OC4J for Interoperability" on page 10-7 for a discussion of command-line options.
5. Redeploy your application.

Advanced Interoperability

1. Restart OC4J with the `-DGenerateIIOP=true` flag.
2. Specify CSIv2 security policies for the bean in `orion_ejb_jar.xml` and in `internal_settings.xml`. See "CSIv2 Security Properties (orion-ejb-jar.xml)" on page 10-13 and "EJB Server Security Properties (internal-settings.xml)" on page 10-9 for details.
3. Edit the client's JNDI property `java.naming.provider.url` to use a `corbaname` URL instead of an `ormi` URL. For details on the `corbaname` URL, see "The corbaname URL" on page 10-4.

4. (Client only) Change the client's `classpath` to include the stub JAR file generated by OC4J. This will normally be

   application_deployment_directory`/`*appname*`/`*module*`/`*module*`_iiopClient.jar`

   Note: IIOP stub and tie class code generation happens at deployment time, unlike ORMI stub generation which happens at runtime. This is why you must add the JAR file to the `classpath` yourself. If you run in the server, a list of generated classes required by the server and IIOP stubs is made available automatically.

5. (Optional) To make the bean accessible to CORBA applications, run `rmic.jar` to generate IDL describing its interfaces. See "Configuring OC4J for Interoperability" on page 10-7 for a discussion of command-line options.
6. Redeploy your application.

The corbaname URL

In order 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 most used by EJB developers. For a full discussion of the `corbaname` scheme, see section 2.5.3 of the CORBA Naming Service Specification. The `corbaname` scheme is based on the `corbaloc` scheme, which is discussed in section 13.6.10.1 of the CORBA specification.

The most common form of the `corbaname` URL scheme is:

`corbaname::`*host*`[:`*port*`]`

This specifies a conventional DNS hostname 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 a stringified `NamingContext`. The CosNaming service on the specified host is responsible for interpreting the naming context.

`corbaname::`*host*`[:`*port*`]#`*namingcontext*

For example,

```
corbaname::example.com:8000#Myapp
```

The rmic.jar Compiler

In order to invoke or be invoked by CORBA objects, RMI objects must have corresponding stubs, skeletons, and IDL. The `rmic.jar` compiler can used to generate stubs and skeletons from Java classes or to generate IDL. The generated IDL can be used to generate non-Java stubs and skeletons using any CORBA IDL compiler.

`java -jar rmic.jar` *`options classname ...`*
The `rmic.jar` compiler takes the following options:

- `-always` (same as `-alwaysgenerate`)—Forces the compiler to generate new outputs even when the existing stubs, ties, or IDL are newer than the input class. Requires the `-iiop` or `-idl` flags.
- `-classpath` *`classpath`*—Specifies the directories to search for the classes.
- `-d` *`pathname`*— Specifies output directory for generated class files.
- `-g`—Generates debugging information.
- `-idl`—Generates IDL for all classes in the input, as well as any classes they reference. IDL, the Interface Description Language, is Corba's mechanism for describing methods and data in a language-independent way.

  Note: The `rmic.jar` compiler generates IDL that uses the CORBA 2.3 extensions to IDL; compilers that do not support these extensions cannot compile rmic-generated IDL.

- `-idlModule <fromJavaPackage<.class>> <toIDLModule>`—Specifies IDLEntity package mapping. An example:

  ```
  -idlModule foo.bar my::real::idlmod
  ```

- `-idlFile <fromJavaPackage<.class>> <toIDLFile>`—Specifies IDLEntity file mapping. An example:

  ```
  -idlFile test.pkg.X TEST16.idl
  ```

- `-iiop`—Generates IIOP stubs and ties. Stub classes are called by clients to transmit RMI messages over IIOP; tie classes are called by the server to process incoming calls and dispatch them to the implementation class. Each remote interface has a stub; each server implementation class has a tie. Stub classes are

also generated for abstract interfaces. An abstract interface does not extend `java.rmi.Remote`; in addition, it either has no methods or all of its methods throw `java.rmi.RemoteException` or one of its superclasses.

- `-keep` (same as `-keepgenerated`)—Preserves intermediate generated source files; by default, these are deleted.
- `-nolocalstubs` —Does not create stubs optimized to run in the same process as the server. Requires `-iiop`.
- `-nowarn`—Turns off all compiler warnings.
- `-noValueMethods`—Stops generation of IDL for methods and constructors within IDL valuetypes. Requires `-idl`.
- `-v1.1`—Creates stubs and skeletons for 1.1 stub protocol version only. By default. rmic generates stubs and skeletons compatible with both 1.1 and 1.2.
- `-v1.2`—Creates stubs and skeletons for 1.2 stub protocol version only. By default. rmic generates stubs and skeletons compatible with both 1.1 and 1.2.
- `-vcompat` (default)—Creates stubs / skeletons compatible with both 1.1 and 1.2 stub protocol versions.
- `-verbose`—Sends messages about compiler status to `stdout`.

Exception Mapping

When EJBs are invoked over IIOP, OC4J must map system exceptions to CORBA exceptions. Table 10–1, "Java-CORBA Exception Mappings" lists the exception mappings.

Table 10–1 Java-CORBA Exception Mappings

OC4J System Exception	CORBA system exception
`javax.transaction.TransactionRolledbackException`	`TRANSACTION_ROLLEDBACK`
`javax.transaction.TransactionRequiredException`	`TRANSACTION_REQUIRED`
`javax.transaction.InvalidTransactionException`	`INVALID_TRANSACTION`
`java.rmi.NoSuchObjectException`	`OBJECT_NOT_EXIST`
`java.rmi.AccessException`	`NO_PERMISSION`
`java.rmi.MarshalException`	`MARSHAL`
`java.rmi.RemoteException`	`UNKNOWN`

Invoking OC4J-Hosted Beans from a Non-OC4J Container

EJBs that are not hosted in OC4J must add the file `oc4j_interop.jar` to the classpath in order to invoke OC4J-hosted EJBs. OC4J expects the other container to make `HandleDelegate` object available in the JNDI namespace at `java:comp/HandleDelegate`. The `oc4j_interop.jar` file contains the standard portable implementations of home and remote handles and metadata objects.

Configuring OC4J for Interoperability

To add interoperability support to your EJB, you must specify interoperability properties. Some of these properties are specified when starting OC4J and others in bean properties 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

The following files contain entries that specify interoperability information.

Server-wide Files

- `server.xml`

  The `<sep-config>` element in this file specifies the pathname, normally `internal-settings.xml`, for the server extension provider properties.

  ```
  <sep-config path="internal-settings.xml">
  ```

- `internal-settings.xml`

  Specifies server extension provider properties specific to RMI / IIOP. See "EJB Server Security Properties (internal-settings.xml)" on page 10-9 for details.

Application-specific Files

- `orion-ejb-jar.xml`

  (Server) The `<ior-security-config>` sub-entity of the `<session-deployment>` and `<entity-deployment>` entities specifies Common Secure Interoperability Version 2 (CSIv2) security properties. See "CSIv2 Security Properties" on page 10-10 for details.

- `ejb_sec.properties`

  (Client) Specifies client-side security properties for an EJB. See "EJB Client Security Properties (ejb_sec.properties)" on page 10-15 for details.

- `jndi.properties`

  (Client) Specifies the URL of the initial naming context. See "JNDI Properties for Interoperability (jndi.properties)" on page 10-17 for details.

EJB Server Security Properties (internal-settings.xml)

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

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

This file specifies the following properties as values within `<sep-property>` entities. Table 10–2, "EJB Server Security Properties" contains a list of properties.

Table 10–2 EJB Server Security Properties

Property	Meaning
`port`	IIOP port number (defaults to `4444`)
`ssl`	`true` if IIOP / SSL is supported, `false` otherwise
`ssl`-port	IIOP / SSL port number (defaults to `4445`) This port is used for server-side authentication only. If your application uses client and server authentication, OC4J will listen on `ssl-port` + 1 for client-side authentication.
`keystore`	Name of keystore (used only if `ssl` is `true`)
`keystore-password`	the keystore password (used only if `ssl` is `true`)
`trusted-clients`	Comma-separated list of hosts whose identity assertions can be trusted. Each entry in the list can be an IP address, a hostname, a hostname pattern (for instance, `*.example.com`), or `*`; `*` alone means that all clients are trusted. The default is to trust no clients.
`truststore`	Name of truststore. If you do not specify a truststore for a server, OC4J uses the keystore as the truststore. (used only if `ssl` is `true`)
`truststore-password`	Truststore password (can only be set if `ssl` is `true`)

The keystore and truststore files use JDK-specified formats to store keys and certificates. A keystore stores a map of private keys and certificates. A truststore stores trusted certificates for the certificate authorities (CAs).

A typical `internal-settings.xml` looks like:

```
<server-extension-provider name="IIOP"
   class="com.oracle.iiop.server.IIOPServerExtensionProvider">
```

```
  <sep-property name="port" value="4444" />
  <sep-property name="host" value="localhost" />
  <sep-property name="trusted-clients" value="*.example.com" />
  <sep-property name="ssl" value="true" />
  <sep-property name="ssl-port" value="4445" />
  <sep-property name="keystore" value="keystore1" />
  <sep-property name="keystore-password" value="changeit" />
</server-extension-provider>
```

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

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

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

CSIv2 Security Properties

Common Secure Interoperability version 2 (CSIv2) is an OMG standard for a secure interoperable wire protocol that supports authorization and identity delegation. You configure CSIv2 properties in three different locations:

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

These configuration files are discussed in "CSIv2 Security Properties (internal-settings.xml)" on page 10-11, "CSIv2 Security Properties (orion-ejb-jar.xml)" on page 10-13, and "EJB Client Security Properties (ejb_sec.properties)" on page 10-13.

CSIv2 Security Properties (internal-settings.xml)

This section discusses the semantics of the values you set within the `<sep-property>` element in `internal_settings.xml`. For details of syntax, see "EJB Server Security Properties (internal-settings.xml)" on page 10-9.

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

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

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

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

- Truststore only
- Both keystore and truststore.
- Neither

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

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

CSIv2 Security Properties (ejb_sec.properties)

If the client does not use client-side SSL authentication, you must set `client.sendpassword` in the `ejb_sec.properties` file in order for the client runtime to insert a security context and send the username and password. You must also set `server.trustedhosts` to include your server.

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

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

Trust Relationships

There are two types of trust relationships:

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

Clients list trusted servers in the EJB property `oc4j.iiop.trustedServers`. See Table 10–3, "EJB Client Security Properties" on page 10-16 for details. Servers list trusted clients in the `trusted-client` property of the `<sep-property>` element in `internal-settings.xml`. See "EJB Server Security Properties (internal-settings.xml)" on page 10-9 for details.

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

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

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

OC4J provides both kinds of trust; you configure trust using the bean's <ior-security-config> element in orion-ejb-jar.xml. See "CSIv2 Security Properties (orion-ejb-jar.xml)" on page 10-13 for details.

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

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

The <transport-config> element

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

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

Notes: If you set <establish-trust-in-client> to required, this overrides specifying username_password in <as-context>.

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

The <as-context> element

This element specifies the message-level authentication properties.

- <auth-method>—Must be set to either username_password or none. If set to username_password, beans will use user names and passwords to authenticate the caller.
- <realm>—Must be set to default at this release.
- <required>—If set to true, the bean requires the caller to specify a username and password.

The <sas-context> element

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

An example:

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

```
  </sas-context>
</ior-security-config>
```

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

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

EJB Client Security Properties (ejb_sec.properties)

Any client, whether running inside a server or not, has EJB security properties. The following are the EJB client security properties controlled by the `ejb_sec.properties` file. By default, OC4J searches for this file in the current directory when running as a client or in `J2EE_HOME/config` when running in the server. You can specify this file's location explicitly with `-Dejb_sec_properties_location=`*`pathname`*. Table 10–3 lists the properties controlled by the `ejb_sec.properties` file.

Table 10–3 EJB Client Security Properties

Property	Meaning
`# oc4j.iiop.keyStoreLoc`	The pathname for the keystore.
`# oc4j.iiop.keyStorePass`	The password for the keystore.
`# oc4j.iiop.trustStoreLoc`	The pathname for the truststore.
`# oc4j.iiop.trustStorePass`	The password for the truststore.
`# oc4j.iiop.enable.clientauth`	Whether the client supports client-side authentication. If this property is set to `true`, you must specify a keystore location and password.
`oc4j.iiop.ciphersuites`	Which cipher suites are to be enabled. The valid cipher suites are: `TLS_RSA_WITH_RC4_128_MD5` `SSL_RSA_WITH_RC4_128_MD5` `TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA` `SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA` `TLS_RSA_EXPORT_WITH_RC4_40_MD5` `SSL_RSA_EXPORT_WITH_RC4_40_MD5` `TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA` `SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA`
`nameservice.useSSL`	Whether to use SSL when making the initial connection to the server.
`client.sendpassword`	Whether to send username and password in clear (unencrypted) in the service context when not using SSL. If this property is set to `true`, the username and password are sent only to servers listed in the `trustedServer` list.
`oc4j.iiop.trustedServers`	A list of servers that can be trusted to receive passwords sent in clear. Has no effect if `client.sendpassword` is set to false. The list is comma-separated. Each entry in the list can be an IP address, a hostname, a hostname pattern (for instance, `*.example.com`), or `*`; `*` alone means that all servers are trusted.

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

JNDI Properties for Interoperability (jndi.properties)

The following RMI/IIOP properties are controlled by the client's `jndi.properties` file:

- `java.naming.provider.url` must be a `corbaname` URL in order for the bean to be interoperable. For details on `corbaname` URLs, see "The corbaname URL" on page 10-4.
- `contextFactory` can now be either `ApplicationClientInitialContextFactory` or the new class `IIOPInitialContextFactory`.

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

Configuring RMI Tunneling

When EJBs must communicate across firewalls they can use tunneling to transmit RMI across HTTP. This tunneling is supported only with RMI/ORMI; you cannot do HTTP tunneling with RMI/IIOP.

To configure OC4J to support RMI tunneling, do the following:

1. Modify the JNDI provider URL. The JNDI provider URL for accessing the OC4J EJB server takes the form:

   `ormi://`*`hostname`*`:`*`ormi_port`*`/`*`the_app`*
   You should change the URL to:

   `http:ormi://`*`hostname`*`:`*`HTTP_PORT`*`/`*`the_app`*

   > **Note:** If omitted, *`HTTP_PORT`* defaults to 80. The argument port number is your HTTP port, *not* your ORMI port.

2. If your HTTP traffic goes through a proxy server, you must specify the `proxyHost` and (optionally) `proxyPort` in the command line when starting the EJB client. If you do not supply a value for `proxyPort`, it defaults to 80.

   `-Dhttp.proxyHost=`*`proxy_host`* `-Dhttp.proxyPort=`*`proxy_port`*

Configuring RMI in server.xml and rmi.xml

In order to use RMI from OC4J, you must edit the `server.xml` and `rmi.xml` files.

Editing server.xml

Your `server.xml` file must specify the pathname of the RMI configuration file in the `<rmi-config>` element. The syntax is:

```
<rmi-config path="RMI_PATH" />
```

The usual *`RMI_PATH`* is `./rmi.xml`; you can name the file whatever you like.

Editing rmi.xml

The file `rmi.xml` must specify which host, port, and username/password will be used to connect to (and accept connections from) remote RMI servers. Your file must contain an `<rmi-server>` element describing possible connections. An `<rmi-server>` element looks like:

```
<rmi-server host="hostname" port="port">
<server host="hostname" username="username" port="port"
  password="password" http-path="pathname"/>
<log>
  <file path="logfilepathname" /> Okay
</log>
</rmi-server>
```

<rmi-server> has the following attributes:

hostname is the host or IP name from which your server will accept RMI requests. *hostname* can be a particular hostname or "[ALL]". If you specify a *hostname*, the OC4J server will only accept RMI requests from that particular host. If *hostname* is "[ALL]" or you omit the host attribute, the OC4J server will accept RMI requests from any host.

port is the port number on which your server listens for RMI requests. If you omit this attribute, it defaults to 23791.

An <rmi-server> element can contain zero or multiple <server> elements and zero or one <log> elements.

Each <server> element specifies a server that your application can contact over RMI. A <server> element takes the form:

```
<server host="hostname" username="username" port="port"
  password="password"/>
```

The host attribute is required; the remaining attributes are optional.

hostname the name or IP address of the server you will contact over RMI.

username the username of a valid principal on the remote server

port the port number on which the remote server listens for RMI requests

password the password used by the principal *username*

The <log> element contains the pathname of a log file to which the server will write all RMI requests.

11

Data Sources

This chapter describes how to configure and use data sources in your Oracle9*i*AS Containers for J2EE (OC4J) application. A data source is a vendor-independent encapsulation of a connection to a database server. A data source instantiates an object that implements the `javax.sql.DataSource` interface.

This chapter covers the following topics:

Introduction

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

Because data sources are vendor-independent, we recommend that J2EE applications retrieve connections to data servers using data sources.

Defining Data Sources

You define OC4J data sources in an XML file known as `data-sources.xml`.

Defining Location of the Data Source XML Configuration File

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

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

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

Defining Data Sources

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

The default data source is an emulated data source. You can use this data source for applications that access and update only a single data server. If you need to update

more than one database, you must use a non-emulated data source. For a full discussion of emulated versus non-emulated data sources, see "Types of Data Sources" on page 11-5.

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

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

- The `class` attribute defines the type of data source you want to use.
- The `location`, `xa-location`, and `ejb-location` attributes are JNDI names that this data source is bound to within the JNDI namespace. We recommend that you use only the `ejb-location` JNDI name in the JNDI lookup for retrieving this data source.
- The `ejb-location` attribute is the JNDI name that this data source is bound to within the JNDI namespace.
- The `connection-driver` attribute defines the type of connection you expect to be returned to you from the data source.
- The URL, username, and password identify the database, its username, and password.

"Using Data Sources" on page 11-12 describes all data source attributes.

Retrieving a Connection from a Data Source

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

> **Note:** Data sources always return logical connections.

Do the following to modify data within your database:

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

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

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

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

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

The following code represents the preceding steps:

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

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

- `getConnection();`

  The username and password are those defined in the data source definition.

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

  This username and password overrides the username and password defined in the data source definition.

If the data source refers to an Oracle database, you can cast the connection object returned on the `getConnection` method to `oracle.jdbc.OracleConnection` and use all the Oracle extensions. This is shown in the following example:

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

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

Note: We **strongly** recommend that you restart OC4J whenever a database crashes. Because connections obtained through data sources are cached, a connection may become invalid if the database referenced by the data source crashes. This is especially true if you have set `the min-connections` attribute or specified a high `inactivity-timeout`. (See Table 11–1, "Data Source Attributes", for a discussion of these attributes.)

Types of Data Sources

There are several types of data sources. Three types are especially important to understand: emulated data sources, non-emulated data sources, and non-JTA data sources.

- Emulated Data Sources— Emulated data sources support local and global transactions. However, instead of relying on underlying database support for JTA, emulated data sources support JTA by emulating the XA API without relying on the relational database manager's XA implementation. This means that emulated data sources do not support two-phase commit operations. The pre-installed default data source is an emulated data source.
- Non-Emulated Data Sources—Non-emulated data sources support local and global transactions. Non-emulated data sources rely on the underlying database implementation of JTA and XA, and so have full support for two-phase commit.
- Non-JTA Data Sources—Non-JTA data sources support only local transactions; they do not support global transactions. These data sources can be provided by Oracle or by any other JDBC-compliant implementation.

Emulated Data Sources

Connections obtained from emulated data sources are extremely fast, because the connections emulate the XA API without providing full XA global transactional support. In particular, emulated data sources do not support two-phase commit. We recommend that you use emulated data sources for local transactions or when your application uses global transactions without requiring two-phase commit. For

efficiency, any JNDI-retrieved connection to the an emulated data source shares the same connection with the first identified username within the same transaction.

You can use the same emulated data source to obtain connections to different databases by changing the values of `url` and `connection-driver`. The following is a definition of an emulated data source:

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

When looking up a `DataSource` object in the JNDI namespace, use the `ejb-location` logical name, as follows:

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

This connection opens a database session for `SCOTT/TIGER`.

Note: Previous releases supported the `location` and `xa-location` attributes for retrieving data source objects. These are now strongly deprecated; applications, EJBs, servlets, and JSPs should use only the JNDI name `ejb-location` in emulated data source definitions for retrieving the data source.

When using an emulated data source, you cannot use global transactions. The `XAResource` that you enlist with the transaction manager is an emulated `XAResource`, so the underlying database is unaware of global transactions. It provides only local transactional support. If you want to use two-phase commit in global transactions, you must use a non-emulated data source.

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

```
Context ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("jdbc/OracleCMTDS1");
Connection conn1 = ds.getConnection("scott", "tiger");
Connection conn2 = ds.getConnection("scott", "tiger");
```

If you provide different a different username and password for the second connection from this data source, an error condition occurs. You can avoid this problem by using the "`dedicated.connection`" JNDI property. This is described in "Using Different Usernames for Two Connections to a Single Data Source" on page 11-20.

Non-Emulated Data Sources

Non-emulated data sources provide full XA and JTA global transactional support. These are the only data sources that support global two-phase commit transactions.

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

The following is an example of a non-emulated data source definition.

```
<data-source
  class="com.evermind.sql.OrionCMTDataSource"
  name="OracleDS"
  location="jdbc/OracleCMTDS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="scott"
  password="tiger"
  url="jdbc:oracle:thin:@hostname:TTC port number:DB SID"
  inactivity-timeout="30"
/>
```

The following are the expected attribute definitions:

- The `location` attribute is the JNDI name that this data source is bound to within the JNDI namespace. You use the `location` JNDI name in the JNDI lookup for retrieving this data source.
- The `connection-driver` attribute defines the type of connection you expect to be returned to you from the data source.
- The URL, username, and password identifies the database, its username, and password.
- The `class` attribute defines what type of data source class to bind in the namespace. For example, you can define a non-emulated data source with the `com.evermind.sql.OrionCMTDataSource` class, as shown above.

Non-JTA Data Sources

Non-JTA data sources provide no support for global transactions. If you use `OracleDataSource`, no connection pooling is available; if you use `OracleConnectionCacheImpl`, connection pooling is supported.

You can use any of the Oracle `DataSource` objects listed in the *Oracle9i JDBC Developer's Guide*. For example, to define a non-emulated data source with the `OracleXADataSource` class, you would configure the following in the `data-sources.xml` file:

```
<data-source
  class="oracle.xa.client.OracleXADataSource"
  name="OracleXADS"
  location="jdbc/OracleXADS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="scott"
  password="tiger"
  url="jdbc:oracle:thin:@hostname:TTC port number:DB SID"
  inactivity-timeout="30"
/>
```

Non-Emulated Data Sources Cannot Mix Transaction Types

When you are using a non-emulated data source, you cannot mix local and global transactions. You must use either one or the other. The following code shows an invalid mixture of local and global transactions:

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

```
Connection conn1 = ds.getConnection("scott", "tiger");
javax.transaction.UserTransaction txn = (javax.transaction.UserTransaction)
  ic.lookup("java:comp/env/UserTransaction");
conn1.work();     // perform work on conn1 in a local transaction
// start global transaction
txn.start();
conn1.morework(); // perform work on conn1 within a global transaction ERROR!
```

This example mixes transaction types in a different (but also incorrect) way:

```
Context ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("jdbc/OracleCMTDS1");
Connection conn1 = ds.getConnection("scott", "tiger");
javax.transaction.UserTransaction txn = (javax.transaction.UserTransaction)
  ic.lookup("java:comp/env/UserTransaction");
//start global transaction
txn.start();
conn1.work();     // perform work on conn1 in a global transaction
txn.commit();
conn1.morework(); // perform work on conn1 within a local transaction ERROR!
```

Even though you have committed the global transaction, you are still mixing global and local transactional work within the same bean.

Mixing Data Sources

A single application may use several different types of data source. If your application mixes data sources, you should be aware of the following issues:

- Only emulated and non-emulated data sources support JTA transactions. You cannot enlist connections obtained from non-JTA data sources in a JTA transaction.
- Only non-emulated data sources support two-phase commit. To enlist multiple connections in a two-phase commit transaction, all the connections must use non-emulated data sources.
- You cannot use both emulated data sources and non-emulated data sources in the same transaction.
- If your application does not use JTA transactions, you can obtain connections from any data source.
- If your application has opened a `javax.transaction.UserTransaction`, all future transaction work must be performed through that object. If you try to invoke the connection's `rollback()` or `commit()` methods, you will receive

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

Two-Phase Commits and Data Sources

Oracle's two-phase-commit coordinator is a DTC Engine that performs two phase commit with appropriate recovery. The two-phase commit engine is responsible for ensuring that when the transaction ends, all changes to all databases are either totally committed or fully rolled back. The two-phase commit engine can be one of the databases that participates in the global transaction or it can be a separate database. If multiple databases or multiple sessions in the same database participate in a transaction, then you must specify a two-phase commit coordinator. Otherwise you cannot commit the transaction.

You can specify a commit coordinator in the following ways:

- You can specify one commit coordinator for all applications using the global `application.xml` in the `J2EE_HOME/config` directory.
- You can override this commit coordinator for an individual application in the application's `orion-application.xml`.

For example:

```
<commit-coordinator>
    <commit-class class="com.evermind.server.OracleTwoPhaseCommitDriver" />
    <property name="datasource"
              value="jdbc/OracleCommitDS" />
     <property name="username"
               value="system" />
      <property name="password"
               value="manager" />
</commit-coordinator>
```

If you specify a username and password in the global `application.xml`, these values override the values in `datasource.xml`. If these values are null, then the username and password in `datasource.xml` are used to connect to the commit coordinator.

The username and password used to connect to the commit coordinator (for example, System) must have "force any transaction" privilege. By default, during installation, the commit-coordinator is specified in he global `application.xml` with username and password as null.

Each data source participating in a two-phase commit should specify `dblink` information in the `OrionCMTDatasource` data source. This `dblink` should be the name of the `dblink` created in the commit coordinator database to connect to this database.

For example, if db1 is the database for the commit coordinator and db2 and db3 are participating in the global transactions, you would create `link2` and `link3` in the `db1` database as shown in the following example.

```
connect commit_user/commit_user
create database link link2 using "inst1_db2"; // link from db1 to db2
create database link link3 using "inst1_db3"; // link from db1 to db3;
```

Next, you would define a data source called `jdbc/OracleCommitDS` in `application.xml`:

```
<data-source
    class="com.evermind.sql.OrionCMTDataSource"
    name="OracleCommitDS"
    location="jdbc/OracleCommitDS"
    connection-driver="oracle.jdbc.driver.OracleDriver"
    username="system"
    password="manager"
    url="jdbc:oracle:thin:@localhost:5521:db1"
   inactivity-timeout="30"/>
```

Here is the data source description of `db2` which participates in the global transaction. Note that `link2`, which was created in `db1`, is specified as a property here:

```
        <data-source
                class="com.evermind.sql.OrionCMTDataSource"
                name="OracleDB2"
                location="jdbc/OracleDB2"
                connection-driver="oracle.jdbc.driver.OracleDriver"
                username="system"
                password="manager"
                url="jdbc:oracle:thin:@localhost:5521:db2"
                inactivity-timeout="30">
               <property name="dblink"
                         value="LINK2.REGRESS.RDBMS.EXAMPLE.COM"/>
      </data-source>
```

Here is the data source description of `db3` which participates in the global transaction. Note that `link3`, which is created in `db1`, is specified as a property here:

```
<data-source
        class="com.evermind.sql.OrionCMTDataSource"
        name="OracleDB3"
        location="jdbc/OracleDB3"
        connection-driver="oracle.jdbc.driver.OracleDriver"
        username="system"
        password="manager"
        url="jdbc:oracle:thin:@localhost:5521:db3"
        inactivity-timeout="30">
       <property name="dblink"
                 value="LINK3.REGRESS.RDBMS.EXAMPLE.COM"/>
</data-source>
```

Using Data Sources

The following sections describe the data sources that your application can use and how to access them:

- Configuring Data Source Objects
- Configuration Files
- Data Source Attributes
- Data Source Methods
- Portable Data Source Lookup

Configuring Data Source Objects

For most purposes, you can use the data sources that are already defined in the server `data-sources.xml` configuration file.

To define a new data source object, use the Oracle Enterprise Manager.To find out how to use the Administrative tools, see the *Oracle9iAS Containers for J2EE User's Guide*. For Oracle Enterprise Manager information, see *Oracle Enterprise Manager Administrator's Guide*. This chapter explains how to set up and manage data sources by editing the configuration files directly.

Configuration Files

One main configuration file establishes data sources at the OC4J server level: `$J2EE_HOME/config/data-sources.xml`. To edit the information in this file, use the Enterprise Manager and drill down to the Data Source page. OC4J parses the `data-sources.xml` file when it starts, instantiates data source objects, and binds them into the server JNDI namespace. When you add a new data source specification, you must restart the OC4J server to make the new data source available for lookup.

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

Data Source Attributes

A data source can take many attributes. Some are required, but most are optional; the required attributes are marked below. The attributes are specified in a `<data-source>` tag. Table 11–1 lists the attributes and their meaning.

Table 11–1 Data Source Attributes

Attribute Name	Meaning of Value	Default Value
`class`	**Required**. Names the class that implements the data source. For non-emulated, the class attribute can be "`com.evermind.sql.OrionCMTDataSource`". For emulated, the class attribute should be "`com.evermind.sql.DriverManagerDataSource`".	N / A
`location`	(The JNDI logical name for the data source object. OC4J binds the class instance into the application JNDI namespace with this name. This JNDI lookup name is used for non-emulated data sources. In future releases, `ejb-location` will be the only supported attribute for JNDI lookup of emulated data sources.	N / A
`name`	The name of the data source. Must be unique within the application.	If this name is not supplied, `ejb-location` is used as the name.
`connection-driver`	The JDBC-driver class name for this data source, which is needed by some data sources that deal with `java.sql.Connection`. For most data sources, the driver should be `"oracle.jdbc.driver.OracleDriver"`.	None.
`username`	The optional name of the schema to connect to.	None.
`password`	The optional password for the schema.	None.
`URL`	The URL for database connections. Must be supplied for Oracle database connections.	None.
`xa-location`	(*Deprecated*) The logical name of an XA data source. This attribute is supported only for emulated data sources. In future releases, `ejb-location` will be the only supported attribute for JNDI lookup.	None.

Table 11–1 Data Source Attributes (Cont.)

Attribute Name	Meaning of Value	Default Value
`ejb-location`	The logical name of an EJB data source. Use this attribute if you are using JTA for single-phase commit transactions or if you are looking up emulated data sources. If you use it to retrieve the data source, you can map the returned connection to `oracle.jdbc.OracleConnection`.	None.
`inactivity-timeout`	Time (in seconds) to cache unused connections before closing them.	60 seconds
`connection-retry-interval`	The interval to wait (in seconds) before retrying a failed connection attempt.	1 second
`max-connections`	The maximum number of open connections for a pooled data source.	Depends on the data source type.
`min-connections`	The minimum number of open connections for a pooled data source. OC4J does not open these connections until `DataSource.getConnection` method is invoked.	0
`wait-timeout`	The number of seconds to wait for a free connection if the pool is used up (that is, has reached max-connections used).	60
`max-connect-attempts`	The number of times to retry making a connection. This is useful when the network is not stable or the environment is unstable for any other reason that will sometimes make connection attempts fail.	3
`property`	This element is used to specify either a database link for two-phase commit transactions (`dblink`) or a database caching scheme (`cache_scheme`).	None

Data Source Methods

You can call the following methods on a `DataSource` object:

getConnection();
Attempt to establish a database connection.

getConnection(String uid, String password);
Attempt to retrieve a database connection, specifying the username and password.

getLoginTimeout();
Retrieve the maximum time in seconds that this data source can wait while attempting to connect to a database

setLoginTimeout(int seconds);
Set the maximum time in seconds that this data source will wait while attempting to connect to a database.

getLogWriter();
Retrieve the log writer for this data source. Returns a java.io.Printwriter object.

setLogWriter(PrintWriter out);
Set the log writer for this data source.

Portable Data Source Lookup

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

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

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

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

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

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

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

You would retrieve this data source without using "`java:comp/env`" as shown by the following JNDI lookup:

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

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

Example 11–1 Mapping Logical JNDI Name to Actual JNDI Name

The following demonstrates option #2 above. If you want to choose a logical name of "`jdbc/OracleMappedDS`" to be used within your code for the JNDI retrieval. Then you would have the following in your `web.xml` or `ejb-jar.xml` files:

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

In order for the actual JNDI name to be found, you must have a `<resource-ref-mapping>` element that maps the "`jdbc/OracleMappedDS`" to the actual JNDI name in the `data-sources.xml` file. If we are using the default emulated data source, then the `ejb-location` would be defined with "`jdbc/OracleDS`" as the actual JNDI name. Thus, the following line would be contained in the OC4J-specific XML file:

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

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

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

Using Oracle JDBC Extensions

To use Oracle JDBC extensions, cast the returned connection to `oracle.jdbc.OracleConnection`, as follows:

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

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

You can use any of the Oracle extensions on the returned connection, "conn".

```
// you can create oracle.jdbc.* objects using this connection
oracle.jdbc.Statement orclStmt =
  (oracle.jdbc.OracleStatement)conn.createStatement();
// assume table is varray_table
oracle.jdbc.OracleResultSet rs =
  orclStmt.executeQuery("SELECT * FROM " + tableName);
while (rs.next())
{
  oracle.sql.ARRAY array = rs.getARRAY(1);
...
}
```

Behavior of a Non-Emulated Data Source Object

The physical behavior of a non-emulated data source object changes depending on whether you retrieve a connection off the data source within a global transaction or not. The following discusses these differences:

- Retrieving a Connection Outside a Global Transaction
- Retrieving a Connection Within a Global Transaction

Retrieving a Connection Outside a Global Transaction

If you retrieve a connection from a non-emulated data source and you are not involved in a global transaction, every `getConnection` method returns a logical handle. When the connection is used for work, a physical connection is created for each connection created. Thus, if you create two connections outside of a global transaction, both connections use a separate physical connection. When you close each connection, it is returned to a pool to be used by the next connection retrieval.

Retrieving a Connection Within a Global Transaction

If you retrieve a connection from a non-emulated data source and you are involved in a global JTA transaction, all physical connections retrieved from the same `DataSource` object by the same user within the transaction share the same physical connection.

For example, if you start a transaction and retrieve two connections from the "jdbc/OracleCMTDS1" DataSource with the "scott" user, both connections share the physical connection. In the following example, both conn1 and conn2 share the same physical connection.

```
Context ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("jdbc/OracleCMTDS1");
//start txn
txn.start();
Connection conn1 = ds.getConnection("scott", "tiger");
Connection conn2 = ds.getConnection("scott", "tiger");
```

However, separate physical connections are retrieved for connections retrieved from separate DataSource objects. The following example shows both conn1 and conn2 retrieved from different DataSource objects—"jdbc/OracleCMTDS1" and "jdbc/OracleCMTDS2". Both conn1 and conn2 will exist upon a separate physical connection.

```
Context ic = new InitialContext();
DataSource ds1 = (DataSource) ic.lookup("jdbc/OracleCMTDS1");
DataSource ds2 = (DataSource) ic.lookup("jdbc/OracleCMTDS2");
//start txn
txn.start();
Connection conn1 = ds1.getConnection();
Connection conn2 = ds2.getConnection();
```

Using Database Caching Schemes

You can define the database caching scheme to use within the data source definition. There are three types of caching schemes: DYNAMIC_SCHEME, FIXED_WAIT_SCHEME, and FIXED_RETURN_NULL_SCHEME. To specify a caching scheme, you specify an integer value for a <property> element named cacheScheme. The supported values are shown in Table 11–2.

Table 11–2 Database Caching Schemes

Value	Cache Scheme
1	DYNAMIC_SCHEME
2	FIXED_WAIT_SCHEME
3	FIXED_RETURN_NULL_SCHEME

The following example is a data source using the DYNAMIC_SCHEME.

```
<data-source
  class="com.evermind.sql.OrionCMTDataSource"
  name="OracleDS"
  location="jdbc/OracleCMTDS1"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="scott"
  password="tiger"
  url="jdbc:oracle:thin:@<hostname>:<TTC port number>:<DB SID>"
  inactivity-timeout="30">
  <property name="cacheScheme" value="1" />
</data-source>
```

Connection Retrieval Error Conditions

The following mistakes can create an error condition:

- Using Different Usernames for Two Connections to a Single Data Source
- Using the OCI JDBC Drivers

Using Different Usernames for Two Connections to a Single Data Source

When you retrieve a connection from the a `DataSource` object with a username and password, this username and password is used on all subsequent connection retrievals within the same transaction. This is true for all data source types. For example, suppose an application retrieves a connection from the "`jdbc/OracleCMTDS1`" data source with the "`scott`" user. When the application retrieves a second connection from the same data source with a different username, such as "`adams`", the username provided is ignored. Instead, the "`scott`" user is used.

```
Context ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("jdbc/OracleCMTDS1");
//start txn
txn.start();
Connection conn1 = ds.getConnection("scott", "tiger"); //uses scott/tiger
Connection conn2 = ds.getConnection("adams", "wood"); //uses scott/tiger also
```

Thus, you cannot authenticate using two different users to the same data source. If you try to access the tables as "`adams/wood`", you enter into an error condition.

Using the OCI JDBC Drivers

The examples of Oracle data source definitions in this chapter use the Oracle JDBC thin driver. However, you can use the Oracle JDBC OCI (thick) driver as well. Set the following before you start the OC4J server:

- install the Oracle Client on the same machine on which OC4J is installed
- set the `ORACLE_HOME` variable
- set `LD_LIBRARY_PATH` (or the equivalent environment variable for your OS) to `$ORACLE_HOME/lib`
- set `TNS_ADMIN` to a valid Oracle administration directory with a valid `tnsnames.ora` file

The URL to use in the `url` attribute of the `<data-source>` element definition can have any of these forms:

- `jdbc:oracle:oci8:@`: this TNS entry is for a database on the same system as the client, and the client connects to the database in IPC mode
- `jdbc:oracle:oci8:@<TNS service name>`: where the TNS service name is an entry in the instance `tnsnames.ora` file
- `jdbc:oracle:oci8:@<full_TNS_listener_description>`: the complete TNS service specification, as described in the *Oracle Net Administrator's Guide*

Using DataDirect Drivers

When your application must connect to heterogeneous databases, use DataDirect JDBC drivers. DataDirect JDBC drivers are not meant to be used with an Oracle database but for connecting to non-Oracle databases, such as Microsoft, SQLServer, Sybase and DB2. If you want to use DataDirect drivers with OC4J, add corresponding entries for each database in the `data-sources.xml` file.

Please see the DataDirect documentation for information on installing the DataDirect JDBC drivers.

The following is an example of a data source entry for SQLServer. For more detailed information, see the *DataDirect Connect JDBC User's Guide and Reference*.

```
<data-source
   class="com.evermind.sql.DriverManagerDataSource"
   name="MerantDS"
   location="jdbc/MerantCoreSSDS"
```

```
   xa-location="jdbc/xa/MerantSSXADS"
   ejb-location="jdbc/MerantSSDS"
   connection-driver="com.merant.datadirect.jdbc.sqlserver.SQLServerDriver"
   username="test"
   password="secret"
   url="jdbc:sqlserver//hostname:port;User=test;Password=secret"
   inactivity-timeout="30"
 />
```

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

```
<data-source
  class="com.evermind.sql.DriverManagerDataSource"
  name="MerantDS"
  location="jdbc/MerantDB2DS"
  xa-location="jdbc/xa/MerantDB2XADS"
  ejb-location="jdbc/MerantDB2DS"
  connection-driver="com.merant.datadirect.jdbc.db2.DB2Driver"
  username="test"
  password="secret"
  url="jdbc:sqlserver//hostname:port;LocationName=jdbc;CollectionId=default;
  inactivity-timeout="30"
/>
```

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

```
<data-source
  class="com.evermind.sql.DriverManagerDataSource"
  name="MerantDS"
  location="jdbc/MerantCoreSybaseDS"
  xa-location="jdbc/xa/MerantSybaseXADS"
  ejb-location="jdbc/MerantSybaseDS"
  connection-driver="com.merant.datadirect.jdbc.sybase.SybaseDriver"
  username="test"
  password="secret"
  url="jdbc:sqlserver//hostname:port;User=test;Password=secret"
  inactivity-timeout="30"
/>
```

You can also use vendor-specific data sources in the class attribute directly. That is, you do not need to use an OC4J-specific data source in the class attribute.

12

Java Transaction API

This chapter describes the Oracle9*i*AS Containers for J2EE (OC4J) Transaction API.

This chapter covers the following topics:

Introduction

Enterprise Java Beans use Java Transaction API (JTA) 1.0.1 for managing transactions. This chapter discusses the method for using JTA in OC4J. It does not cover JTA concepts—you must understand how to use and program global transactions before reading this chapter. See the Sun Microsystems Web site for more information. Code examples are available for download from the OTN OC4J sample code site:

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

JTA involves enlisting resources and demarcating the transaction.

Enlisting Resources The complexity of your transaction is determined by how many resources your application enlists.

- Single-Phase Commit (1pc): If only a single resource (database) is enlisted in the transaction, you can use single-phase commit.
- Two-Phase Commit (2pc): If more than one resource is enlisted, you must use two-phase commit, which is more difficult to configure.

Demarcating Transactions Your application demarcates the transaction through either bean-managed or container-managed transactions.

- Bean-managed transactions are programmatically demarcated within your bean implementation. The transaction boundaries are completely controlled by the application.
- Container-managed transactions are controlled by the container. That is, the container either joins an existing transaction or starts a new transaction for the application—as defined within the deployment descriptor—and ends the newly created transaction when the bean method completes. It is not necessary for your implementation to provide code for managing the transaction.

Note: Not all data sources support JTA transactions. (See "Types of Data Sources" on page 11-5 for details.)

Single-Phase Commit

Single-phase commit (1pc) is a transaction that involves only a single resource. JTA transactions consist of enlisting resources and demarcating transactions.

Enlisting a Single Resource

To enlist the single resource in the single-phase commit, you must do the following:

1. Configure the `DataSource` in `data-sources.xml`. For single-phase commit, use an emulated data source.
2. Retrieve a connection to this `DataSource` in your bean implementation after the transaction has begun.
   a. After the transaction has begun (demarcated), lookup the `DataSource` from the JNDI name space.

b. Retrieve a connection off this `DataSource` object using the `getConnection` method.

Configuring the Data Source

Use an emulated data source for a single phase commit. Refer to Chapter 11, "Data Sources" for information on emulated and non-emulated data source types.

Use the default `DataSource` object if you can for the single-phase commit JTA transaction. After modifying this data source `url` attribute with your database URL information, retrieve the data source in your code using a JNDI lookup with the JNDI name configured in the `ejb-location` attribute. Configure a `DataSource` for each database involved in the transaction.

```
<data-source
  class="com.evermind.sql.DriverManagerDataSource"
  name="OracleDS"
  location="jdbc/OracleCoreDS"
  xa-location="jdbc/xa/OracleXADS"
  ejb-location="jdbc/OracleDS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="scott"
  password="tiger"
  url="jdbc:oracle:thin:@myhost:myport:mySID"
  inactivity-timeout="30"
/>
```

The following are the expected attribute definitions:

- The `ejb-location` attribute is the JNDI name that this data source is bound to within the JNDI namespace. You use the `ejb-location` JNDI name in the JNDI lookup for retrieving this data source.
- The `connection-driver` attribute defines the type of connection you expect to be returned to you from the data source.
- The URL, username, and password identify the database, its username, and password. Modify this example with the URL, username, and password of your intended database. These are used to retrieve the data source session and database schema that will be used to access and modify the database.
- The `class` attribute defines what type of data source class to bind in the namespace. The emulated data sources are defined using the `com.evermind.sql.DriverManagerDataSource` class, as shown above.

Retrieving the Data Source Connection

Before executing any SQL statements against tables in the database, you must retrieve a connection to that database. For these updates to be included in the JTA transaction, you must do one of the following:

1. After the transaction has begun (demarcated), lookup the `DataSource` from the JNDI name space. You can use one of two methods for the retrieval.
2. Retrieve a connection off this `DataSource` object using the `getConnection` method.

There are two methods for retrieving the `DataSource` out of the JNDI namespace, as follows:

- Performing JNDI Lookup on Data Source Definition
- Performing JNDI Lookup Using Environment

Performing JNDI Lookup on Data Source Definition You can perform a lookup on the JNDI name bound to the `DataSource` definition in the `data-sources.xml` file and retrieve a connection, as follows:

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

Performing JNDI Lookup Using Environment You can perform a lookup on a logical name defined in the environment of the bean container. For more information, see Chapter 11, "Data Sources". Basically, define the logical name in the J2EE deployment descriptor as follows:

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

Map the `<res-ref-name>` in the OC4J-specific deployment descriptor to the JNDI name bound in the `data-sources.xml` file as follows:

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

where "`jdbc/OracleDS`" is the JNDI name defined in the `data-sources.xml` file.

Then retrieve the data source using the environment JNDI lookup and create a connection, as shown below:

```
InitialContext ic = new InitialContext();
DataSource ds = ic.lookup("java:comp/env/jdbc/OracleMappedDS");
Connection conn = ds.getConnection();
```

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

Example 12–1 shows a small portion of an employee session bean that uses container-managed transactions and uses SQLJ for updating the database.

Example 12–1 Retrieving a Connection Using Portable JNDI Lookup

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

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

//Lookup the DataSource using the <resource-ref> definition
remoteDS = (DataSource)ic.lookup ("java:comp/env/jdbc/OracleMappedDS");

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

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

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

Demarcating the Transaction

With JTA, you can demarcate the transaction yourself by specifying that the bean is bean-managed transactional, or designate that the container should demarcate the transaction by specifying that the bean is container-managed transactional. Container-managed transaction is available only to entity beans and stateful beans.

> **Note:** Currently, the client cannot demarcate the transaction. Propagation of the transaction context cannot cross OC4J instances. Thus, neither a remote client nor a remote EJB can initiate or join the transaction.

You specify the type of demarcation in the bean deployment descriptor. The following shows a session bean that is declared as container-managed transactional by defining the `<transaction-type>` element as "`Container`". To configure the bean to use bean-managed transactional demarcation, define this element to be "`Bean`".

```
<session>
  <description>no description</description>
  <ejb-name>myEmployee</ejb-name>
  <home>cmtxn.ejb.EmployeeHome</home>
  <remote>cmtxn.ejb.Employee</remote>
  <ejb-class>cmtxn.ejb.EmployeeBean</ejb-class>
  <session-type>Stateful</session-type>
  <transaction-type>Container</transaction-type>
  <resource-ref>
   <res-ref-name>jdbc/OracleMappedDS</res-ref-name>
   <res-type>javax.sql.DataSource</res-type>
   <res-auth>Application</res-auth>
  </resource-ref>
</session>
```

Container-Managed Transactional Demarcation

If you define your bean to use container-managed transactions (CMT), then you must specify how the container manages the JTA transaction for this bean in the `<trans-attribute>` element in the deployment descriptor. Table 12–1 briefly describes the transaction attribute types that you should specify in the deployment descriptor.

Table 12–1 Transaction Attributes

Transaction Attribute	Description
`NotSupported`	The bean is not involved in a transaction. If the bean invoker calls the bean while involved in a transaction, the invoker's transaction is suspended, the bean executes, and when the bean returns, the invoker's transaction is resumed.
`Required`	The bean must be involved in a transaction. If the invoker is involved in a transaction, the bean uses the invoker's transaction. If the invoker is not involved in a transaction, the container starts a new transaction for the bean.
`Supports`	Whatever transactional state that the invoker is involved in is used for the bean. If the invoker has begun a transaction, the invoker's transaction context is used by the bean. If the invoker is not involved in a transaction, neither is the bean.
`RequiresNew`	Whether or not the invoker is involved in a transaction, this bean starts a new transaction that exists only for itself. If the invoker calls while involved in a transaction, 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, a `RemoteException` is thrown.

The following `<container-transaction>` portion of the deployment descriptor demonstrates how this bean specifies the `RequiresNew` transaction attribute for all (*) methods of the `myEmployee` EJB.

```
<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 rollback the transaction. The container handles all of these functions based on the transaction attribute specified in the deployment descriptor.

Bean-Managed Transactions

If you declare the bean as bean-managed transactional (BMT) within the `<transaction-type>`, then the bean implementation must demarcate the start, commit, or rollback for the global transaction. In addition, you must be careful to retrieve the `DataSource` connection after you start the transaction and not before.

Programmatic Transaction Demarcation For programmatic transaction demarcation, the bean writer can use either the JTA user transaction interface or the JDBC connection interface methods. The bean writer must explicitly start and commit or roll back transactions within the timeout interval.

Programmatic transaction demarcation must be used by Web components (JSP, Servlets) and Stateless Session beans; Stateful Session beans may use it; entity beans must use declarative transaction demarcation.

Client-side Transaction Demarcation This form of transaction demarcation is not required by the J2EE specification, and is not recommended for performance and latency reasons. OC4J does not support client-side transaction demarcation.

JTA Transactions

The Web component or bean writer must explicitly issue begin, commit and rollback methods of the `UserTransaction` interface as follows:

```
Context initCtx = new Initial Context();
ut = (UserTransaction)  initCtx.lookup("java:comp/env/UserTransaction");
…
ut.begin();
// Commit the transaction started in ejbCreate.
Try {
        ut.commit();
} catch (Exception ex) { …..}
```

JDBC Transactions

The `javax.sql.Connection` class provides commit and rollback methods. JDBC transactions implicitly begin with the first SQL statement that follows the most recent commit, rollback, or connect statement.

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

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

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

1. Retrieves the `UserTransaction` object from the bean context.
2. Starts the transaction with the `begin` method.
3. Enlists the database.

This example is the same as in "Retrieving the Data Source Connection" on page 12-4, but it is surrounded by `UserTransaction` `begin()` and `commit()` methods.

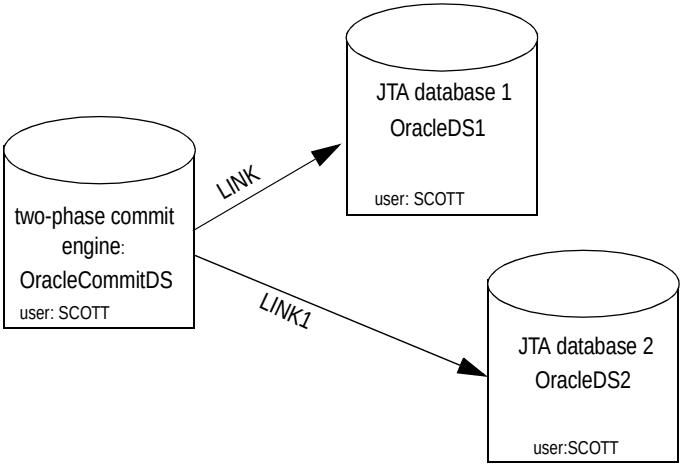
```
DataSource remoteDS;
Context ic;
int empno = 0;
double salary = 0.0;
//Retrieve the UserTransaction object. Its methods are used for txn demarcation
UserTransaction ut = ctx.getUserTransaction ();

//Start the transaction
ut.begin();

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

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

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

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

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

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

Two-Phase Commit

The main focus of JTA is to declaratively or programmatically start and end simple and global transactions. When a global transaction is completed, all changes are either committed or rolled back. The difficulty in implementing a two-phase commit transaction is in the configuration details. To understand this section, you must understand non-emulated data sources. See "Non-Emulated Data Sources" on page 11-7.

Figure 12–1 shows an example of a two-phase commit engine—`jdbc/OracleCommitDS`—coordinating two databases in the global transaction—`jdbc/OracleDS1` and `jdbc/OracleDS2`. Refer to this example when going through the steps for configuring your JTA two-phase commit environment.

Figure 12–1 Two-Phase Commit Diagram



Configuring Two-Phase Commit Engine

When a global transaction involves multiple databases, the changes to these resources must all be committed or rolled back at the same time. That is, when the transaction ends, the transaction manager contacts a coordinator—also known as a two-phase commit engine—to either commit or roll back all changes to all included databases. The two-phase commit engine is an Oracle9*i* database that is configured with the following:

- Fully-qualified database links from itself to each of the databases involved in the transaction. When the transaction ends, the two-phase commit engine communicates with the included databases over their fully-qualified database links.
- A user that is designated to create sessions to each database involved and is given the responsibility of performing the commit or rollback. The user that performs the communication must be created on all involved databases and be given the appropriate privileges.

To facilitate this coordination, you must configure the following:

1. Designate and configure an Oracle9*i* database as the two-phase commit engine. When you have defined the database that is to act as the two-phase commit engine, configure it as follows:

   a. Define a non-emulated data source, using `OrionCMTDataSource`, for the two-phase commit engine database in the `data-sources.xml` file. The following code defines the two-phase commit engine `OrionCMTDataSource` in the `data-sources.xml` file.

   ```
   <data-source
     class="com.evermind.sql.OrionCMTDataSource"
     name="OracleCommitDS"
     location="jdbc/OracleCommitDS"
     connection-driver="oracle.jdbc.driver.OracleDriver"
     username="coordusr"
     password="coordpwd"
     url="jdbc:oracle:thin:@mysun:5521:jis"
     inactivity-timeout="30"
   />
   ```

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

   Configure the two-phase commit engine in the `orion-application.xml` as follows:

   ```
   <commit-coordinator>
    <commit-class class="com.evermind.server.OracleTwoPhaseCommitDriver" />
    <property name="datasource" value="jdbc/OracleCommitDS" />
    <property name="username" value="coordusr" />
    <property name="password" value="coordpwd" />
   </commit-coordinator>
   ```

The parameters are as follows:

* Specify the JNDI name of "**jdbc/OracleCommitDS**" for the OrionCMTDataSource defined in the data-sources.xml. This identifies the DataSource to use as the two-phase commit engine.
* Specify the two-phase commit engine username and password. This step is optional, because you could also specify it in the DataSource configuration. This is the username and password to use as the login authorization to the two-phase commit engine. This user must have the privileges previously mentioned in step 4.

Note: The container prioritizes the username and password defined in the orion-application.xml file over the username and password defined in the data-sources.xml file.

---

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

The following example defines the two-phase commit engine in the <commit-coordinator> element in the application.xml file.

* The OracleTwoPhaseCommitDriver class is defined in the <commit-class> element.
* The JNDI name for the OrionCMTDataSource is identified in the <property> element whose name is "datasource".
* The username is identified in the <property> element "username".
* The password is identified in the <property> element "password".

2. Create the user on the two-phase commit engine that facilitates the transaction. First, the user opens a session from the two-phase commit engine to each of the involved databases. Second, it must be granted the CONNECT, RESOURCE, CREATE SESSION privileges to be able to connect to each of these databases. The FORCE ANY TRANSACTION privilege allows the user to commit or roll back the transaction.

   Additionally, create this user and grant these permissions on all databases involved in the transaction.

   For example, if the user that is needed for completing the transaction is COORDUSR, you would do the following on the two-phase commit engine and EACH database involved in the transaction:

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

3. Configure fully-qualified public database links (using the `CREATE PUBLIC DATABASE LINK` command) from the two-phase commit engine to each database that may be involved in the global transaction. This is necessary for the two-phase commit engine to communicate with each database at the end of the transaction. The `COORDUSR` must be able to connect to all participating databases using these links.

   This example has two databases involved in the transaction. The database link from the two-phase commit engine to each database is provided on each `OrionCMTDataSource` definition in a `<property>` element in the `data-sources.xml` file. See the next step for the "`dblink`" `<property>` element.

4. Configure non-emulated data source objects of type `OrionCMTDataSource` for each database involved in the transaction with the following information:

   a. The JNDI bound name for the object.

   b. The URL for creating a connection to the database.

   c. The fully-qualified database link from the two-phase commit engine to this database. This is provided in a `<property>` element within the `DataSource` definition in the `data-sources.xml` file.

   The following `OrionCMTDataSource` objects specify the two databases involved in the global transaction. Notice that each of them has a `<property>` element named "`dblink`" that denotes the database link from the two-phase commit engine to itself.

```
<data-source
  class="com.evermind.sql.OrionCMTDataSource"
  name="OracleCMTDS1"
  location="jdbc/OracleDS1"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="scott"
  password="driver"
  url="jdbc:oracle:thin:@mysun:5521:jis"
  inactivity-timeout="30"
  <property name="dblink"
   value="LINK.REGRESS.RDBMS.DEV.US.ORACLE.COM"/>
</data-source>
```

```
<data-source
  class="com.evermind.sql.OrionCMTDataSource"
  name="OracleCMTDS2"
  location="jdbc/OracleDS2"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="scott"
  password="driver"
  url="jdbc:oracle:thin:@mysun:6521:jis"
  inactivity-timeout="30"
  <property name="dblink"
   value="LINK.REGRESS.RDBMS.DEV.US.ORACLE.COM"/>
</data-source>
```

Note: If you change the two-phase commit engine, you must update all database links—both within the new two-phase commit engine as well as within the `OrionCMTDataSource` `<property>` definitions.

Once the two-phase commit engine and all the databases involved in the transaction are configured, you can start and stop a transaction in the same manner as the single-phase commit. See "Single-Phase Commit" on page 12-2 for more information.

Two-Phase Commit Elements in the orion-application.xml DTD

The following code example contains the elements in the `orion-application.xml` file that are relevant to the two-phase commit engine:

```
<!ELEMENT orion-application
(ejb-module*,web-module*,client-module*,security-role-mapping*,
persistence?, library*, principals?, mail-session*, user-manager?,
log?, data-sources?, commit-coordinator?, namespace-access?)>
<!-- Transaction co-ordinator for the server. -->
<!ELEMENT commit-coordinator (commit-class, property*)>

<!ELEMENT commit-class (#PCDATA)>
<!ATTLIST class name CDATA #IMPLIED>

<!-- A property to set when using a custom/3rd-party DataSource. -->
<!ELEMENT property (#PCDATA)>
<!ATTLIST property name CDATA #IMPLIED
value CDATA #IMPLIED >
```

13

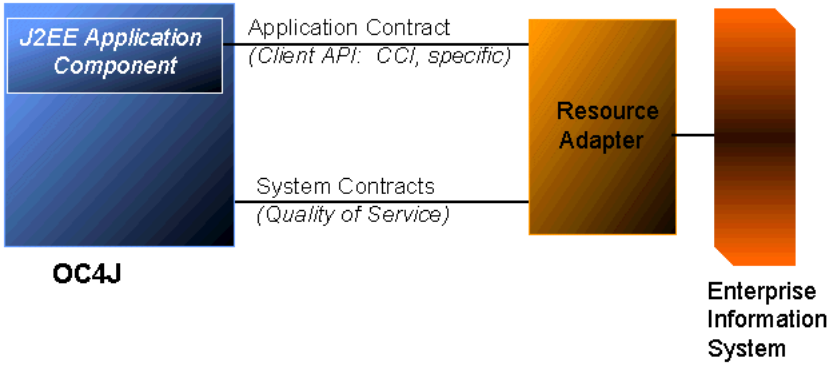
J2EE Connector Architecture

This chapter describes how to use the J2EE Connector Architecture (J2EE Connector) in an Oracle9*i*AS Containers for J2EE (OC4J) application. This chapter covers the following topics:

Introduction

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

Figure 13–1 Java Connector Architecture



Resource Adapters

A *resource adapter* is a driver that an application server or an application client uses to connect to a specific EIS. Examples of resource adapters are JDBC or SQLJ drivers to connect to a relational database, an ERP resource adapter to connect to an ERP system, and a TP resource adapter to connect to a TP monitor. J2EE 1.3 requires application servers to support both standalone and embedded resource adapters.

- *Standalone resource adapters* are available to all applications deployed in the application server instance. These adapters are stored in standalone Resource Adapter Archive (RAR) files. Here is an example:

```
/META-INF/ra.xml
/META-INF/oc4j-ra.xml
/howto.html
/images/icon.jpg
/ra.jar
/cci.jar
/win.dll
/solaris.so
```

> **Note:** The JAR files referred to in the RAR file can be located in any directory within the archive.

- *Embedded resource adapters* are available only to the J2EE application(s) with which they are bundled in an enterprise application archive (EAR) file.

Classes that are defined within a resource adapter are available to all application components, including EJBs, that reference the resource adapter. Classes defined by standalone resource adapters are available to all applications deployed within OC4J; classes defined by embedded resource adapters are available only to applications within their own EAR file.

Application Contracts

The client API furnished by a resource adapter can be either the standard Common Client Interface (CCI), or a client API specific to the type of a resource adapter and its underlying EIS. For example, the JDBC API is the client API specific to relational database accesses. The EIS side of the contract is implemented by the resource adapter, transparently to the application components.

Quality of Service Contracts

Java Connector Architecture also defines three Quality of Service (QoS) contracts between an application server and an EIS.

- *Connection Pooling* enables an application server to pool connections to an underlying EIS, and enables application components to connect to an EIS.

> **Note:** The J2EE Connector connection-pooling interface differs from the JDBC interface; J2EE Connector connection pools are not shared with JDBC connection pools, nor do properties set for one connection pool affect the other.

- *Transaction Management* enables an application server to use a transaction manager (JTA `XAResource`) to manage transactions across multiple resource managers.
- *Security management* provides authentication, authorization, and secure communication between the J2EE server and the EIS.

All resource adapters must support their side of the QoS contracts to be pluggable into application servers.

Support for Optional Features

OC4J does not support the optional connection sharing (section 6.9 in the J2EE Connector Architecture 1.0 specification) and local transaction optimization (section 6.12) features.

Deploying Resource Adapters

This section discusses deployment descriptors, deploying standalone resource adapters, and deploying embedded resource adapters.

OC4J supports three deployment descriptors: `ra.xml`, `oc4j-ra.xml`, and `oc4j-connectors.xml`. The `ra.xml` descriptor is normally supplied with the resource adapter. Whenever you deploy a resource adapter within an EAR file, OC4J generates `oc4j-connectors.xml` and `oc4j-ra.xml`; you should manually edit the second file.

The ra.xml Descriptor

The `ra.xml` descriptor is the standard J2EE deployment descriptor for resource adapters. For details, see the J2EE Connector Architecture 1.0 specification.

The oc4j-ra.xml Descriptor

The `oc4j-ra.xml` descriptor provides OC4J-specific deployment information (JNDI pathname and connector properties) for resource adapters. For each resource adapter, `oc4j-ra.xml` contains one or more `<connector-factory>` elements specifying a JNDI name corresponding to a set of configuration parameter values. OC4J binds each connection into the proper JNDI namespace location as a `ConnectionFactory` instance.

A `<connector-factory>` element can contain any combination of the following elements (all are optional):

- `<description>`—Text description of the connector. This element is not interpreted by OC4J.
- `<config-property>`—Value for a property defined in `ra.xml`. All `ra.xml` files define the properties `ServerName`, `PortNumber`, `UserName`, `Password`, and `ConnectionURL`, although an adapter does not need to support them. Values defined in `oc4j-ra.xml` override any values defined in `ra.xml`.

- `<connection-pooling>`— Parameters describing how J2EE Connector pooled connections are to be handled. This element is discussed in "The <connection-pooling> Element" on page 13-5.
- `<security-config>`—Parameters describing how to supply usernames and passwords to the EIS. This element is discussed in "The <security-config> Element" on page 13-6.
- `<log>`— Pathname of a log file for a connection property set. The syntax is:

```
<log>
  <file path="pathname" />
</log>
```

The <connection-pooling> Element

Connection pooling is a J2EE 1.3 feature that allows a set of connections to be reused within an application. Because the J2EE Connector specification is intended to be general rather than database-specific, the J2EE Connector connection-pooling interface differs significantly from the JDBC interface.

To set a connection pooling property, specify a `<property>` element within the `<connection-pooling>` element. The syntax is:

`<property name="propname" value="propvalue" />`.

The *propname* must be one of:

- `maxConnections`—Maximum number of connections permitted within pool. Defaults to infinity.
- `minConnections`—Minimum number of connections. Defaults to 0. If `minConnections` is greater than 0, the specified number of connections will be opened when OC4J is initialized. OC4J may not be able to open the connections if necessary information is unavailable at initialization time. For instance, if the connection requires a JNDI lookup, it cannot be created, because JNDI information is not available until initialization is complete.
- `scheme`—Specifies how OC4J handles connection requests after maximum permitted number of connections is reached. You must specify one of the following values:
  - `dynamic`—OC4J always creates a new connection and returns it to the application, even if this violates the maximum limit. When these limit-violating connections are closed, they are destroyed instead of being returned to the connection pool.

  - `fixed`—OC4J raises an exception when the application requests a connection and the maximum limit has been reached.
  - `fixed_wait`—OC4J blocks the application's connection request until an in-use connection is returned to the pool. If `waitTimeout` is specified, OC4J throws an exception if no connection becomes available within the specified time limit.
- `waitTimeout`—Maximum number of seconds OC4J will wait for an available connection if `maxConnections` has been exceeded and the `fixed_wait` scheme is in effect. Defaults to infinity.

The <security-config> Element

The `<security-config>` element specifies the user name and password for container-managed sign-ons.

There are two ways of supplying this information in the `<security-config>` element of the `oc4j-ra.xml` file:

- Specifying mapping subelements explicitly (`<principal-mapping-entries>` subelement)
- Specifying the name of a user-created mapping class that either implements `oracle.j2ee.connector.PrincipalMapping` or inherits from `oracle.j2ee.AbstractPrincipalMapping` (`<principal-mapping-interface>` subelement)

Authentication issues are discussed in detail in "Authentication in Container-Managed Sign-On" on page 13-15. This section discusses only the syntax for the `<security-config>` element.

A `<security-config>` element contains either a `<principal-mapping-entries>` element, specifying user names and passwords explicitly; a `<principal-mapping-interface>` element, specifying the name of the mapping class; or a `<jaas-module>` element, specifying the JAAS module to be used for authentication.

```
<security-config>
  <principal-mapping-entries>                        // 1
     <default-mapping>                               // 2
        <res-user>username</res-user>                // 3
        <res-password>password</res-password>        // 4
     </default-mapping>
     <principal-mapping-entry>                       // 5
        <initiating-user>iuname</initiating-user>    // 6
```

```
        <res-user>username</res-user>
        <res-password>password</res-password>
     </principal-mapping-entry>
  </principal-mapping-entries>

  <principal-mapping-interface>                    // 7
    <impl-class>classname</impl-class>             // 8
    <property name="propname"
              value="propvalue" />                 // 9
  </principal-mapping-interface>

  <jaas-module>                                    // 10
      <jaas-application-name>                      // 11
         appname
      </jaas-application-name>
  </jaas-module>
</security-config>
```

1. `<principal-mapping-entries>`— Provides a declarative specification for resource mapping. This element begins with an optional `<default-mapping>` element; it continues with one or more `<principal-mapping-entry>` elements.
2. `<default-mapping>`— Specifies the user name and password for the default resource principal.
3. `<res-user>`— Specifies user name.
4. `<res-password>`— Specifies password.
5. `<principal-mapping-entry>`— Specifies a mapping from a single initiating principal to a resource principal and password.
6. `<initiating-user>`— Specifies the initiating principal.
7. `<principal-mapping-interface>`— Specifies information necessary to employ user-created classes to provide mappings.
8. `<impl-class>`— Specifies the name of the user-provided `PrincipalMapping` implementation.
9. `<property name="`*name*`" value="`*value*`">` (optional; can be repeated)— Specifies information specific to your `PrincipalMapping` implementation: for instance, the path of the principal mapping file, or LDAP server connection information.

10. <jaas-module>— Specifies the JAAS module used for authentication. Has only one element, <jaas-application-name>.
11. <jaas-application-name>— Specifies the name of the JAAS module used for authentication.

The oc4j-ra.xml DTD

The XML DTD for the resource adapter descriptor is:

```
<!ENTITY % JNDIPATH "CDATA">
<!-- Define a property set for a Connector Architecture
   compliant resource adapter. -->
<!ELEMENT config-property (#PCDATA)>
<!ATTLIST config-property name CDATA
   #REQUIREDvalue CDATA #REQUIRED>
<!-- Define a property set for a Connector Architecture c
     ompliant resource adapter. -->
<!ELEMENT connector-factory (description?, config-property*,
connection-pooling?, security-config?, log?)>
<!ATTLIST connector-factory connector-name CDATA #REQUIRED
  location %JNDIPATH; #REQUIRED>
<!ELEMENT connection-pooling (property*)>
<!ELEMENT default-mapping (res-user, res-password)>
<!-- A short description. -->
<!ELEMENT description (#PCDATA)>
<!-- A relative/absolute path to log events to. -->
<!ELEMENT file (#PCDATA)>
<!ATTLIST file path CDATA #IMPLIED>
<!-- name of the class which implements the
  oracle.j2ee.connector.PrincipalMapping interface -->
<!ELEMENT impl-class (#PCDATA)>
<!-- logged in user name of J2EE application -->
<!ELEMENT initiating-user (#PCDATA)>
<!ELEMENT jaas-application-name (#PCDATA)>
<!ELEMENT jaas-module (jaas-application-name)>
<!-- Logging settings. -->
<!ELEMENT log (file)>
<!-- This file contains the definition of property sets
  configuration for an installed Connector Architecture
  compliant resource adapters. -->
<!ELEMENT oc4j-connector-factories (connector-factory*)>
<!ELEMENT principal-mapping-entries (description?,
   default-mapping?, principal-mapping-entry*)>
<!ELEMENT principal-mapping-entry
  (initiating-user, res-user, res-password)>
```

```
<!ELEMENT principal-mapping-interface (impl-class, property*)>
<!-- Contains a name/value pair initialization param. -->
<!ELEMENT property (#PCDATA)><!ATTLIST property name
  CDATA  #IMPLIEDvalue CDATA #IMPLIED>
<!-- password of the EIS resource -->
<!ELEMENT res-password (#PCDATA)>
<!-- user name of the EIS resource -->
<!ELEMENT res-user (#PCDATA)>
<!-- principal mapping configurations -->
<!ELEMENT security-config (
  principal-mapping-entries | principal-mapping-interface
  | jaas-module )>
```

The oc4j-connectors.xml Descriptor

The `oc4j-connectors.xml` descriptor configures the resource adapters deployed by `oc4j-ra.xml`. The `oc4j-connectors.xml` descriptor lists the standalone resource adapters deployed in this OC4J instance, as well as the resource adapters embedded within an application.

> **Note:** The name and pathname of the connectors descriptor are defined by the `<connectors>` element under the `<orion-application>` element in the file `orion-application.xml`. If no `<connectors>` element is specified in `orion-application.xml`, then the default path is `$OC4J_HOME/connectors/`*`rarname`*`./oc4j-connectors.xml`.

The root element is `<oc4j-connectors>`. Each individual connector is represented by a `<connector>` element that specifies the name and pathname for the connector. Each `<connector>` element contains the following elements:

- `<description>` *Optional*—Text description of the connector. Not interpreted by OC4J.
- `<native-library path="`*`pathname`*`">` *Optional*—Directory containing native libraries. If you do not specify this element, OC4J expects the libraries to be located in the directory containing the decompressed RAR. OC4J interprets the *`pathname`* attribute relative to the decompressed RAR directory.
- `<security-permission enabled="`*`booleanvalue`*`">`—Permissions to be granted to each resource adapter. Each `<security-permission>` contains a

<security-permission-spec> that conforms to the Java 2 Security policy file syntax.

OC4J automatically generates a <security-permission> element in oc4j-connectors.xml for each <security-permission> element in ra.xml. Each generated element has the enabled attribute set to false. Setting the enabled attribute to true grants the named permission.

Example:

```
<oc4j-connectors>
  <connector name="myEIS" path="eis.rar">
    <native-library> path="mylibrary"</native-library>
    <security-permission>
      <security-permission-spec enabled="false">
        grant {permission java.lang.RuntimePermission "LoadLibrary", *’};
      </security-permission-spec>
    </security-permission>
  </connector>
</oc4j-connectors>
```

The oc4j-connectors.xml DTD

The XML DTD for the connectors descriptor is:

```
<!-- An installed Connector Architecture compliant resource adapter. -->
<!ELEMENT connector
   (description?, native-library?, security-permission*)>
<!ATTLIST connector name CDATA #REQUIRED path CDATA #REQUIRED>
<!-- A short description. -->
<!ELEMENT description (#PCDATA)>
<!-- Relative path of native libraries in the resource adatper. -->
<!ELEMENT native-library (#PCDATA)>
<!ATTLIST native-library path CDATA #IMPLIED>
<!-- This file contains the configuration for the installed
      Connector Architecture compliant resource adapters
      of an application-server. -->
<!ELEMENT oc4j-connectors (connector*)>
<!-- Java security permissions for
     resource adapter jar files. -->
<!ELEMENT security-permission (security-permission-spec)>
<!ATTLIST security-permission enabled (true|false) "false">
<!--The element permission-spec specifies a security permission
basedon the Security policy file syntax
[reference: Java 2, Security  architecture specification]
```

```
http://java.sun.com/products/jdk/1.3/docs/guide/security/PolicyFiles.html#FileSy
ntax-->
<!ELEMENT security-permission-spec (#PCDATA)>
```

Deploying Standalone Resource Adapter Archives

You can deploy standalone resource adapter archives in OC4J. During deployment, give each standalone resource adapter a unique name for future operations, such as undeployment of the resource adapter. You deploy standalone resource adapters in one of the following ways:

- Deploying Using admin.jar
- Deploying Manually

Deploying Using admin.jar

To deploy standalone resource adapters you use the `-deployconnector` switch of the command-line tool `admin.jar`. The syntax is:

`-deployconnector -file` *`mypath.rar`* `-name` *`myname`* `-nativeLibPath` *`libpathname`* `-grantAllPermissions`

The `-deployconnector` switch is supported by additional command-line switches:

- `-file` *`mypath`* *(required)* — pathname of the resource adapter's RAR file
- `-name` *`myname`* *(required)*— resource adapter's name
- `-nativelibpath` *`libpathname`* — pathname for native libraries within the RAR file (see also the `<native-library>` element in "The oc4j-connectors.xml Descriptor" on page 13-9.)
- `-grantallpermissions`—grants all runtime permissions requested within the RAR (see also the `<security-permission>` element in "The oc4j-connectors.xml Descriptor" on page 13-9)

The `admin.jar` tool decompresses the RAR file into `$OC4J_HOME/`*`connectordirectory`*`/myname`, creating the directory if it does not exist.

The default *`connectordirectory`* is `$OC4J_HOME/connectors`. To specify a different connector directory, edit the `server.xml` file, setting the `connector-directory` attribute of the `<application-server>` element to the correct pathname.

```
<application-server connector-directory="my_connectors">
```

The `admin.jar` tool then creates (or updates) `oc4j-connectors.xml` and `oc4j-ra.xml` in *`my_connectors`*. See "The oc4j-ra.xml Descriptor" on page 13-4 and "The oc4j-connectors.xml Descriptor" on page 13-9 for a discussion of these files. If the deployment descriptor specifies transaction level and authentication mechanisms that are not supported by OC4J, the `admin.jar` tool prints an error message.

Example:

```
java -jar admin.jar ... -deployconnector -name accounts -file ./accounts.rar
```

Deploying Manually

If you prefer to deploy your connector manually, you must:

1. Create a *`connectorname`* directory under $`OC4J_HOME`/*`connectordirectory`*,.
2. Copy the connector's RAR file into $`OC4J_HOME`/*`connectordirectory`*,/*`connectorname`*.
3. Create an `oc4j-connectors.xml` file in *`connectordirectory`* for the new resource adapter, or add a `<connector>` element to the file if it already exists.
4. Restart OC4J. OC4J generates a new `oc4j-ra.xml` in $`OC4J_HOME/application-deployments/default/`*`connectorname`* for the adapter. You must modify the generated file to contain a `<connector_factory>` element appropriate for your connector.

> **Note:** See "The oc4j-ra.xml Descriptor" on page 13-4 and "The oc4j-connectors.xml Descriptor" on page 13-9 for details on `oc4j-connectors.xml` and `oc4j-ra.xml`.

Removing Resource Adapters

To remove a deployed resource adapter, use the `-undeployconnector` switch of `admin.jar`. The syntax is:

```
-deployconnector -name myname
```

The required `-name` argument specifies which adapter is being removed. This command removes all `<connector>` entries that use the specified resource adapter

from `oc4j-connectors.xml` and deletes the `$OC4J_HOME/connector_directory/myname` directory.

If you prefer, you can remove an adapter manually by deleting all `<connector>` entries that refer to the adapter from `oc4j-connectors.xml` and deleting the `$OC4J_HOME/connector_directory/myname` directory.

Deploying Embedded Resource Adapters

Each application deployed in an OC4J instance that contains resource adapter(s) has a corresponding `oc4j-connectors.xml` file under `$OC4J_HOME/application-deployments/app-name/`.

The `oc4j-connectors.xml` file contains the list of resource adapters for the Web application packaged within an EAR file (one entry for each resource adapter). For details on this file, see "The oc4j-ra.xml Descriptor" on page 13-4 and "The oc4j-connectors.xml Descriptor" on page 13-9. Applications with embedded RARs are deployed in the same fashion (either using the `admin.jar` tool or manually) as applications without RARs.

A resource adapter archive, `myPackaged.rar`, is packaged in the EAR file `myApp.ear`. The application is then deployed with OC4J under `$OC4J_HOME/applications/myapp/myPackaged`.

If the EAR file includes an `oc4j-connectors.xml` file specifying the deployment name `myRA`, the generated `oc4j-ra.xml` file is located in `$OC4J_HOME/application-deployment/myapp/myRA/`. An `oc4j-connectors.xml` file is created under `$OC4J_HOME/application-deployment/myapp/`.

Example:

Assume that a standalone resource adapter connection is configured in `oc4j-ra.xml` to be bound to the location `eis/myEIS`. An application component looks up its connection factory using the JNDI name `"java:comp/env/eis/myEIS"`. The application component must have the `<resource-ref>` element defined in its deployment descriptor in `web.xml` or `ejb-jar.xml`, which may look like the following example:

```
<resource-ref>
  <res-ref-name>eis/myEIS</res-ref-name>
  <res-type>javax.resource.cci.ConnectionFactory</res-type>
  <res-auth>Application</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

```
try
{
 Context ic = new InitialContext();
 cf = (ConnectionFactory)
   ic.lookup("java:comp/env/eis/myEIS");
 } catch (NamingException ex) {
ex.printStackTrace();
}
```

Specifying Container-Managed or Component-Managed Sign-On

Applications can use either application components or the OC4J application server to manage resource-adapter sign-on to the EIS system. You specify the manager using the `<res-auth>` deployment descriptor element for EJB or Web components. If `<res-auth>` is set to `Application`, the application component signs on to the EIS programmatically. The application component is responsible for providing explicit security information for the sign-on. If `<res-auth>` is set to `Container`, OC4J provides the resource principal and credentials required for signing on to the EIS.

Example:

```
Context initctx = new InitialContext();
// perform JNDI lookup to obtain a connection factory
javax.resource.cci.ConnectionFactory cxf =

(javax.resource.cci.ConnectionFactory)initctx.lookup("java:com/env/eis/MyEIS");
  // For container-managed sign-on, no security information is passed in the
getConnection call
  javax.resource.cci.Connection cx = cxf.getConnection();
  // If component-managed sign-on is specified, the code should instead provide
explicit security
  // information in the getConnection call
  // We need to get a new ConnectionSpec implementation instance for setting
login
  // attributes
  com.myeis.ConnectionSpecImpl connSpec = ...
  connSpec.setUserName("EISuser");
  connSpec.setPassword("EISpassword");
  javax.resource.cci.Connection cx = cxf.getConnection(connSpec);
```

In either case, the `createManagedConnection` method in the resource adapter's implementation of `javax.resource.spi.ManagedConnectionFactory` interface is called to create a physical connection to the EIS.

If you specify component-managed sign-on, OC4J invokes the `createManagedConnection` method with a null `Subject` and the `ConnectionRequestInfo` object passed in from the application component code. If you specify container-managed sign-on, OC4J provides a `javax.security.auth.Subject` object to the `createManagedConnection` method. The content of the `Subject` object depends on the value in the `<authentication-mechanism-type>` and `<credential-interface>` elements in the resource adapter deployment descriptor.

If `<authentication-mechanism-type>` is `BasicPassword` and `<credential-interface>` is `javax.resource.spi.security.PasswordCredential`, then the `Subject` object must contain `javax.resource.spi.security.PasswordCredential` objects in the private credential set.

On the other hand, if `<authentication-mechanism-type>` is `Kerbv5` or any other non-password-based authentication mechanism, and `<credential-interface>` is `javax.resource.spi.security.GenericCredential`, then the `Subject` object must contain credentials represented by instances of implementers of the `javax.resource.spi.security.GenericCredential` interface. The `GenericCredential` interface is used for resource adapters that support non-password-based authentication mechanisms, such as Kerberos.

Authentication in Container-Managed Sign-On

When using container-managed sign-on, OC4J must provide a resource principal and its credentials to the EIS. The principal and credentials can be obtained in one of the following ways:

- Configured Identity - The resource principal is independent of the initiating / caller principal and can be configured at deployment time in a deployment descriptor.
- Principal Mapping - The resource principal is determined by a mapping from the identity and / or security attributes of the initiating / caller principal.
- Caller Impersonation - The resource principal acts on behalf of an initiating / caller principal by delegating the caller's identity and credentials to the EIS.

- Credentials Mapping - The resource principal is the same as the initiating / caller principal, but with its credential mapped from the authentication type used by OC4J to the authentication type used by the EIS. An example would be to map a public key certificate-based credential associated with a principal to a Kerberos credential.

OC4J supports all these methods with three authentication mechanisms:

- JAAS Pluggable Authentication
- User-Created Authentication Classes
- Modifying oc4j-ra.xml

The following sections discuss these mechanisms in detail.

JAAS Pluggable Authentication

OC4J furnishes a JAAS pluggable authentication framework that conforms to Appendix C in the Connector Architecture 1.0 specification. With this framework, an application server and its underlying authentication services remain independent from each other, and new authentication services can be plugged in without requiring modifications to the application server.

Authentication services can obtain resource principals and credentials using any of the following modules:

- Principal Mapping JAAS module
- Credential Mapping JAAS module
- Kerberos JAAS module (used for Caller Impersonation)

The JAAS login modules can be provided by resource adapter vendors, the EIS vendors, or by the customer. Login modules must implement the `javax.security.auth.spi.LoginModule` interface, as documented in the Sun JAAS specification.

OC4J provides initiating user subjects to login modules by passing an instance of `javax.security.auth.Subject` containing any public certificates and an instance of `oracle.j2ee.connector.InitiatingPrincipal` representing the OC4J user. OC4J may pass a null `Subject` if there is no authenticated user (that is, an anonymous user). The JAAS login module's login method must, based on the initiating user, find the corresponding resource principal and create new `PasswordCredential` or `GenericCredential` instances for the resource principal. The resource principal and credential objects are then added to the initiating `Subject` in the `commit` method. The resource credential is passed to the

createManagedConnection method in the javax.resource.spi.ManagedConnectionFactory implementation provided by the resource adapter. If a null Subject is passed, the JAAS login module is responsible for creating a new javax.security.auth.Subject containing the resource principal and the appropriate credential.

The InitiatingPrincipal and InitiatingGroup Classes

The classes oracle.j2ee.connector.InitiatingPrincipal and oracle.j2ee.connector.InitiatingGroup are used to represent OC4J users to the JAAS login modules. OC4J creates instances of oracle.j2ee.connector.InitiatingPrincipal and incorporates them into the Subject that is passed to the initialize method of the login modules. The oracle.j2ee.connector.InitiatingPrincipal class implements the java.security.Principal interface, and adds the method getGroups().

```
/**
 * Returns a Set of groups (or roles in JAZN terminology) that this
 * principal is a member of.
 *
 * @return A set of InitiatingGroup objects representing the groups
 *      that this pricipal belongs to.
 */
public Set getGroups()
```

The getGroups method returns a java.util.Set of oracle.j2ee.connector.InitiatingGroup objects, representing the OC4J groups or JAZN roles for this OC4J user. The group membership is defined in OC4J-specific descriptor files such as principals.xml or jazn-data.xml, depending on the user manager. The oracle.j2ee.connector.InitiatingGroup class implements but does not extend the java.security.Principal interface.

Login modules can use getGroups() to provide mappings between OC4J groups and EIS users. The java.security.Principal interface methods support mappings between OC4J users and EIS users. Login modules do not need to refer to the oracle.j2ee.connector.InitiatingPrincipal and oracle.j2ee.connector.InitiatingGroup classes if they do not provide mappings between OC4J groups and EIS users.

JAAS and the <connector-factory> Element

Each <connector-factory> element in oc4j-ra.xml can specify a different JAAS login module. You specify a name for the connector factory configuration in

the <jaas-module> element. Here is an example of a <connector-factory> element in oc4j-ra.xml that uses JAAS login modules for container-managed sign-on:

```
  <connector-factory connector-name="myBlackbox" location="eis/myEIS1">
    <description>Connection to my EIS</description>
    <config-property name="connectionURL"
value="jdbc:oracle:thin:@localhost:5521:orcl" />
    <security-config>
      <jaas-module>
        <jaas-application-name>JCADemo</jaas-application-name>
      </jaas-module>
    </security-config>
  </connector-factory>
```

In JAAS you must specify which LoginModule to use for a particular application, and in what order to invoke the LoginModules. JAAS uses the value specified in the <jaas-application-name> element to look up LoginModules.

User-Created Authentication Classes

OC4J provides the oracle.j2ee.connector.PrincipalMapping interface for principal mapping.

```
package oracle.j2ee.connector;

public interface PrincipalMapping
{
/**
* Initializes the various settings for the PrincipalMapping implementation
class.
* Implementation class may use the properties for setting default user name and
* password, LDAP connect info, or default mapping.
*
* OC4J will pass the properties specified in the <principal-mapping-interface>
* element in oc4j-ra.xml to this method.
*
* @param prop A Properties object containing the set up information required
*             by the implementation class.
*/
  public void init(Properties prop);

/**
* The ManagedConnectionFactory instance that can be used in creating a
   * PasswordCredential.
```

```
   *
 * @param mcf The ManagedConnectionFactory instance that is needed when
  *creating a PasswordCredential instance
  */
  public void setManagedConnectionFactory(ManagedConnectionFactory mcf);

  /**
* Passes the authentication mechanism(s) supported by the resource
* adapter to the PrincipalMapping implementation class.
   * The key of the map passed is a String containing the supported mechanism
* type, such as "BasicPassword", or "Kerbv5". The value is a String
  * containig the corresponding credentials interface as declared in ra.xml,
* such as "javax.resource.spi.security.PasswordCredential".
  *
 * The map may contain multiple elements if the resource adatper supports
  * multiple authentication mechanisms.
   *
 * @param authMechanisms The authentication mechanisms and their corresponding
   *  credentials intereface supported by the resource adapter
  */
  public void setAuthenticationMechanisms(Map authMechanisms);

    /**
* This is the method that performs the principal mapping. An application user
   * subject is passed, and the implemetation of this method should return
   * a subject for use by the resource adapter to log in to the EIS resource
* per the JCA specifications.
   *
 * OC4J will only called this method for container-managed sign on.
*
 * @param initiatingSubject A Subject containing the application server logged
   *     in principals and public credentials.
     *
    * @return A Subject for use by resource adapter to log in to the remote EIS.
    *      It may return null if the proper resource principal cannot be
determined.
  */
  public Subject mapping(Subject initiatingSubject);
}
```

The `mapping` method must return a `Subject` containing the resource principal and credential. The `Subject` returned must adhere to either option A or option B in section 8.2.6 of the Connector Architecture 1.0 specification. OC4J invokes the `mapping` method with the initiating user as the `initiatingPrincipal`.

OC4J also provides the abstract class `oracle.j2ee.connector.AbstractPrincipalMapping`. This class provides a default implementation of the `setManagedConnectionFactory()` and `setAuthenticationMechanism()` methods, as well as utility methods to determine whether the resource adapter supports the `BasicPassword` or Kerbv5 authentication methods, and a method for extracting the `Principal` from the application server user `Subject`. By extending the `oracle.j2ee.connector.AbstractPrincipalMapping` class, developers need only implement the `init` and `mapping` methods.

Here are the utility methods provided by the `oracle.j2ee.connector.AbstractPrincipalMapping` class:

```
/**
   *  Utility method provided by this abstract class to return
  *  the ManagedConnectionFactory instance for use to create a
    *  PasswordCredentials object
   *
    *  @return The ManagedConnectionFactory instance that is needed when
    *    creating a PasswordCredential instance
    */
  public ManagedConnectionFactory getManagedConnectionFactory()

   /**
   * Utility method provided by this abstract class to return the Map
   * of all authentication mechanisms supported by this resource adapter.
  * The key of the map passed is a String containing the supported mechanism
   * type, such as "BasicPassword", or "Kerbv5". The value is a String
  * containig the corresponding credentials interface as declared in ra.xml,
   * such as "javax.resource.spi.security.PasswordCredential".
  *
    * @return The authentication mechanisms and their corresponding
     *      credentials intereface supported by the resource adpater
   */
    public Map getAuthenticationMechanisms()

     /**
   * Utility method provided by this abstract class to return whether
   * BasicPassword authention mechanism is supported by this resource
   * adapter.
   *
    * @return true if BasicPassword authentication mechanism is supported
   *      by the resource adapter, false otherwise.
  */
```

```
  public boolean isBasicPasswordSupported()

  /**
 * Utility method provided by this abstract class to return whether
 * Kerbv5 authention mechanism is supported by this resource
* adapter.
 *
  * @return true if Kerbv5 authentication mechanism is supported
*      by the resource adapter, false otherwise.
*/
  public boolean isKerbv5Supported()

 /**
 * Utility method provided by this abstract class to extract the
    * Principal object from the given application server user subject
  * passed from OC4J.
*
  * @param subject The application server user subject passed from
   *       OC4J.
*
  * @return The principal extracted from the given subject
   */
  public Principal getPrincipal(Subject subject)
```

After you create your implementation class, copy a JAR file containing the class into the directory containing the decompressed RAR file. This directory is typically `$OC4J_HOME/applications/application_name/`*`rar-name`*. After copying the file, edit `oc4j-ra.xml` to contain a `<principal-mapping-interface>` element for the new class; see "The <security-config> Element" on page 13-6 for details.

Extending AbstractPrincipalMapping

This simple example demonstrates how to extend the `oracle.j2ee.connector.AbstractPrincipalMapping` abstract class to provide a principal mapping that always maps the user to the default user and password. You specify the default user and password by using properties under the `<principal-mapping-interface>` element in `oc4j-ra.xml`.

The `PrincipalMapping` class is called `MyMapping`. It is defined as follows:

```
package com.acme.app;

import java.util.*;
import javax.resource.spi.*;
```

```
import javax.resource.spi.security.*;
import oracle.j2ee.connector.AbstractPrincipalMapping;
import javax.security.auth.*;
import java.security.*;

public class MyMapping extends AbstractPrincipalMapping
{
  String m_defaultUser;

   String m_defaultPassword;

    public void init(Properties prop)
    {
      if (prop != null)
      {
           // Retrieves the default user and password from the properties
         m_defaultUser = prop.getProperty("user");
           m_defaultPassword = prop.getProperty("password");
      }
    }

    public Subject mapping(Subject initiatingSubject)
    {
      // This implementation only supporst BasicPassword authentication
      // mechanism. Return if the resource adapter does not support it.
      if (!isBasicPasswordSupported())
            return null;

      // Use the utility method to retrieve the Principal from the
      // OC4J user. This code is included here only as an example.
      // The principal obtained is not being used in this method.
       Principal principal = getPrincipal(initiatingSubject);

    char[] resPasswordArray = null;
      if (m_defaultPassword != null)
         resPasswordArray = m_defaultPassword.toCharArray();

       // Create a PasswordCredential using the default user name and
      // password, and add it to the Subject per option A in section
    // 8.2.6 in the JCA 1.0 spec.
     PasswordCredential cred = new PasswordCredential(m_defaultUser,
resPasswordArray);
        cred.setManagedConnectionFactory(getManagedConnectionFactory());
       initiatingSubject.getPrivateCredentials().add(cred);
      return initiatingSubject;
```

```
    }
}
```

You add a <principal-mapping-interface> entry to oc4j-ra.xml that specifies com.acme.app.MyMapping for the principal mapping mechanism:

```
<connector-factory name="..." location="...">
  ...
  <security-config>
 <principal-mapping-interface>
   <impl-class>com.acme.app.MyMapping</impl-class>
   <property name="user" value="scott" />
   <property name="password" value="tiger" />
 </principal-mapping-interface>
  </security-config>
  ...
</connector-factory>
```

Modifying oc4j-ra.xml

If you prefer, you can create default principal mappings in the oc4j-ra.xml file. To use the default principal mappings mechanism, use the <principal-mapping-entries> subelement under the <security-config> element. For syntax details, see "The <security-config> Element" on page 13-6.

You use the <default-mapping> element to specify the user name and password for the default resource principal. This principal is used to log on to the EIS if there is no <principal-mapping-entry> element whose initiating user corresponds to the current initiating principal. If no default mapping is specified, OC4J uses the values of the configuration properties UserName and Password from the deployment descriptor (either in ra.xml or oc4j-ra.xml), assuming these defaults are acceptable to the resource adapter. If neither configuration properties nor a default mapping is specified, OC4J may not be able to log in to the EIS.

Each <principal-mapping-entry> element contains a mapping from initiating principal to resource principal and password.

For example, if the OC4J principal `scott` should be logged in to a certain EIS, `myEIS1`, as user name `scott` and password `tiger`, while all other OC4J users should be logged in to the EIS using user name `guest` with password `guestpw`, the `<connector-factory>` element in `oc4j-ra.xml` should look like this:

```
<connector-factory name="..." location="...">
    ...
  <security-config>
    <principal-mapping-entries>
      <default-mapping>
        <res-user>guest</res-user>
        <res-password>guestpw</res-password>
      </default-mapping>
      <principal-mapping-entry>
        <initiating-user>scott</initiating-user>
        <res-user>scott</res-user>
        <res-password>tiger</res-password>
      </principal-mapping-entry>
    </principal-mapping-entries>
  </security-config>
    ...
</connector-factory>
```

14

Working with Java Object Cache

This chapter describes the Oracle9*i*AS Containers for J2EE (OC4J) Java Object Cache, including its architecture and programming features.

This chapter covers the following topics:

Java Object Cache Concepts

Oracle9*i*AS 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 Oracle9*i*AS.

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 Oracle9*i*AS cache architecture includes the following cache components:

- **Oracle 9iAS 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 very 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 post-processing 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 may contain generated pages or may 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 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 allows 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 Messaging Service (JMS), Advanced Queueing (AQ), or Simple Object Access Protocol (SOAP).

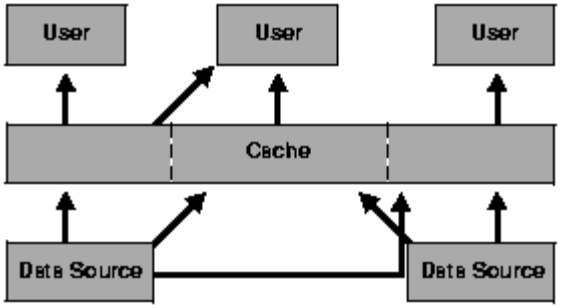
Note: This chapter focuses on the Java Object Cache. For a full discussion of all three caches and their differences, see the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference.*

Java Object Cache Basic Architecture

For a programmer using the Java Object Cache, information has one of three characteristics:

1. Static information that never changes. The programmer handles the data efficiently using a Java `Hashtable`.
2. Dynamic information that is unique. The programmer must generate data each time the information is requested.
3. Variable information that is sometimes static and sometimes is generated. The programmer uses the Java Object Cache.
4. Figure 14–1 shows the basic architecture for the Java Object Cache. The cache delivers information to a user process. The process could be a servlet application that generates HTML pages or any other Java application.

Figure 14–1 Java Object Cache Basic Architecture

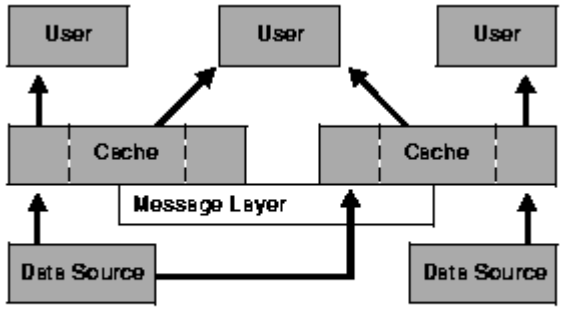


Distributed Object Management

For simplicity, availability, and performance, the Java object cache is specific to each process (object creation is not centrally controlled). However, using distributed object management, the Java Object Cache provides coordination of updates and invalidations between processes. If an object is updated or invalidated in one process, 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 14–2 shows the architecture for the Java Object Cache, using distributed object management. The cache delivers information to a user process. The user process could be a servlet application that generates HTML pages or any other Java application. Using the distributed object management message layer, the application uses the Java Object Cache to share the information across processes and between caches.

Figure 14–2 Java Object Cache Distributed Architecture



How the Java Object Cache Works

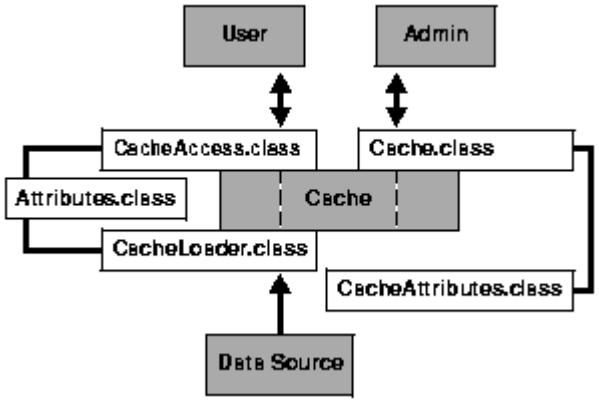
The Java Object Cache manages Java objects within a process, across processes, or on a local disk. The Java Object Cache provides a powerful, flexible, and easy-to-use service that significantly improves Java performance by managing local copies of Java objects. There are very few restrictions on the 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 application programming interfaces (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 14–3 shows the basic Java Object Cache APIs. Figure 14–3 does not show distributed cache management.

Figure 14–3 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 14–1 provides a summary of the constructs in the cache environment and the cache object types.

See Also:

- Java Object Cache Object Types on page 14-8
- Java Object Cache Environment on page 14-10

Table 14–1 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 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 can be accessed from memory or disk, depending on the size of the object and the cache capacity.

Java Object Cache Features

The Java Object Cache provides the following features:

- Objects can be updated or invalidated.
- Objects can be invalidated either explicitly, or with an attribute specifying the expiration time or the idle time.
- Objects can be coordinated between processes.
- Object loading and creation can be automatic.
- Object loading can be coordinated between processes.
- Objects can be associated in cache regions or groups with similar characteristics.
- Cache event notification provides for event handling and special processing.
- Cache management attributes can be specified for each object or applied to cache regions or groups.

Java Object Cache Object Types

This section describes the object types that the Java Object Cache manages, including:

- Memory Objects
- Disk Objects
- StreamAccess Objects
- Pool Objects

Restriction on Identifying Objects:

Objects are identified by a name that can be any Java object. Usually, the name is represented with a `String`. The Java object used for the identifying name must override the default Java object `equals` method, and the default Java object `hashcode` method. The `String` class provides implementations for both of these methods.

If you provide an object to use as the Java Object Cache name, you need to provide implementations for the `equals` and `hashcode` methods for the object. If the object is distributed, then the `Serializable` interface must also be implemented.

Memory Objects

Memory objects are Java objects that the Java Object Cache manages. Memory objects are stored in the Java VM's 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 may be controlled externally (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 memory objects 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 `CacheAccess.replace()`).

The `CacheAccess.defineObject()` method associates attributes with an object. If attributes are not defined, 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 usually 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 you to re-create objects when the system or the Java VM is restarted after the process fails or shuts down.

There are very few restrictions on Java Object Cache memory objects. Memory objects can contain any Java object.

See Also: "Developing Applications Using Java Object Cache" on page 14-17

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 may be shared by all Java Object Cache processes, or they may be local to a particular process, depending on 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. Disk objects can be updated by obtaining a private copy of the disk object (file). When the Java Object Cache requires additional space, disk objects that are not being referenced may be removed from the cache.

There are very few restrictions on disk objects in the Java Object Cache.

See Also: "Developing Applications Using Java Object Cache" on page 14-17

StreamAccess Objects

StreamAccess objects are objects that are accessed as a stream, and are automatically loaded to the disk cache. The object is loaded as an `OutputStream` and read as an

`InputStream`. 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, while larger objects are streamed directly from 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 provide 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 very large objects.

The Java Object Cache places some restrictions on StreamAccess objects.

Pool Objects

A *pool object* is a special class of object that the Java Object Cache manages. A pool object contains a set of identical object instances. The pool object itself is a shared object, while 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` may be associated with a pool object. These attributes apply to the pool object as a whole, or they can be applied to the objects within the pool individually.

The Java Object Cache instantiates objects within a pool using an application-defined factory object. The size of a pool decreases or increases based on demand and on the values of the `TimeToLive` or `IdleTime` attributes. A minimum size for the pool is specified when the pool is created. The minimum-size value is interpreted as a request rather than a guaranteed minimum value. Objects within a pool object are subject to removal from the cache due to lack of space, so the pool may decrease below the requested minimum value. A maximum pool size value can be set that puts 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
- Cache Object Attributes

This section describes these Java Object Cache environment constructs.

Cache Regions

Objects that use the Java Object Cache service are managed 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 only require one region. The Java Object Cache provides a default region; when a region is not specified, objects are placed in the default region.

Attributes may be defined for a region and are then inherited by the objects, subregions, and groups within the region.

See Also: "Cache Object Attributes" on page 14-12 and "Developing Applications Using Java Object Cache" on page 14-17

Cache Subregions

Objects that use the Java Object Cache are managed 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 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.

See Also: "Cache Object Attributes" on page 14-12 and "Developing Applications Using Java Object Cache" on page 14-17

Cache Groups

A *cache group* creates an association between objects within the Java Object Cache. Cache groups allow related objects to be manipulated together. Objects are typically associated in a cache group because they need to be invalidated together or they use common attributes. Any set of cache objects within the same region or subregion can be associated using a cache group, which may in turn, include other cache groups.

An Java Object Cache object can only belong to 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 may have its own attributes, or it may inherit its attributes from its parent region, subregion, or group.

Group names are not used to identify individual objects. A group defines a set or collection of objects that have something in common. A group does not define a hierarchical name space. Object type does not distinguish objects for naming purposes; therefore, a region cannot include a group and a memory object with the same name. Use subregions to define a hierarchical name space within a region.

Groups can contain groups, with the groups having a parent and child relationship. The child group inherits attributes from the parent group.

Cache Object Attributes

Cache object *attributes* affect how the Java Object Cache manages objects. Each object type, region, subregion, and group has a set of associated attributes. An object's applicable attributes contain either the default attribute values; the attribute values inherited from the object's parent region, subregion, or group; or the attribute values that you select for the object.

Attributes fall into two categories:

1. Attributes that must be defined before an object is loaded into the cache. Table 14–2 summarizes these attributes. Each of the attributes shown in Table 14–2 does not have corresponding set or get methods, except the `LOADER` attribute. Use the `Attributes.setFlags()` method to set these attributes.
2. Attributes that can be modified after an object is stored in the cache. Table 14–3 summarizes these attributes.

Note: Some attributes do not apply to certain types of objects. See Object Types sections in the descriptions in Table 14–2 and Table 14–3.

Using Attributes Defined Before Object Loading

The attributes shown in Table 14–2 must be defined on an object before the object is loaded. These attributes determine an object's basic management characteristics.

The following list shows the methods you can use to set the attributes shown in Table 14–2 (by setting the values of an `Attributes` object argument).

- `CacheAccess.defineRegion()`
- `CacheAccess.defineSubRegion()`
- `CacheAccess.defineGroup()`
- `CacheAccess.defineObject()`
- `CacheAccess.put()`
- `CacheAccess.createPool()`
- `CacheLoader.createDiskObject()`
- `CacheLoader.createStream()`
- `CacheLoader.SetAttributes()`

Note: You cannot reset the attributes shown in Table 14–2 by using the `CacheAccess.resetAttributes()`method.

Table 14–2 Java Object Cache Attributes–Set at Object Creation

Attribute Name	Description
DISTRIBUTE	This attribute 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. Pool objects are always local, so this attribute does not apply to pool objects. **Default Value:** All objects are local.
GROUP_TTL_ DESTROY	This attribute 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:** By default only group member objects are invalidated when the TimeToLive expires.
LOADER	This attribute specifies the CacheLoader associated with the object. **Object Types:** When set on a region or a group, the specified CacheLoader becomes the default loader for the region, subregion, or group, the LOADER attribute is individually specified on objects within the region or the group. **Default Value:** By default, no LOADER is set.
ORIGINAL	This attribute indicates that the object was created by the application 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 destroyed when they are no longer useful. **Object Types:** When set on a region or a 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**: By default, this attribute is not set.
REPLY	This attribute specifies whether objects can expect to receive a reply from remote caches after a request for an object update or invalidation has completed. This attribute should be set when a high level of consistency is required between cached objects. 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 a 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 only applies when the DISTRIBUTE attribute is set to the value DISTRIBUTE. Pool objects are always local, so this attribute does not apply for pool objects. **Default Value:** By default no reply is sent. When DISTRIBUTE is set to local the REPLY attribute is ignored.

Table 14–2 Java Object Cache Attributes–Set at Object Creation (Cont.)

Attribute Name	Description
SPOOL	This attribute 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 only applies to memory objects. If the object is also distributed, the object can survive the death of the process that spooled it. Local objects are only accessible 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. If this attribute is set on a region, subregion, or group, all associated objects must implement the `java.io.Serializable` interface. **Object Types:** When set on a region, subregion, or a 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:** By default, memory objects are not stored to disk.
SYNCHRONIZE	This attribute is used to synchronize updates within multiple threads or at multiple locations within a site. Updates are synchronized by obtaining ownership for objects. Use the `CacheAccess.getOwnership()` method to obtain ownership of an 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 a group, ownership is applied to the region, subregion, or group as a whole. Pool objects do not use this attribute. **Default Value:** By default updates are not synchronized.
SYNCHRONIZE_ DEFAULT	This attribute 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 a group, ownership is applied to individual objects within the region, subregion, or group. Pool objects do not use this attribute. **Default Value:** By default updates are not synchronized.

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 14–3 lists these attributes. These attributes can be set using the methods listed in the list shown before Table 14–2, and can be reset using the `CacheAccess.resetAttributes()` method.

Table 14–3 Java Object Cache Attributes

Attribute Name	Description
`DefaultTimeToLive`	The `DefaultTimeToLive` applies only to regions, subregions, and groups. This attribute establishes a default value for the `TimeToLive` that is applied to all objects individually within the region, subregion, or group. This value can be overridden be 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`	The `IdleTime` attribute specifies the amount of time an object may 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 objects explicitly set `IdleTime`. **Default Value:** no automatic `IdleTime` invalidation.
`CacheEventListener`	This attribute 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 the group. **Default Value:** By default, no `CacheEventListener` is set.
`TimeToLive`	The `TimeToLive` attribute establishes the maximum amount of time 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 on 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 may 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`.

Developing Applications Using Java Object Cache

This section describes how to develop applications that use Java Object Cache. This section covers the following topics:

Importing the Java Object Cache

The Oracle installer installs the Java Object Cache jar file `cache.jar` in the directory `$ORACLE_HOME/javacache/lib` on UNIX or in `%ORACLE_HOME%\javacache\lib` on Windows NT.

To use the Java Object Cache, you need to import `oracle.ias.cache`.

```
import oracle.ias.cache.*;
```

Defining a Cache Region

All access to the Java Object Cache is through a `CacheAccess` object. A `CacheAccess` object provides access to the cache through 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, `defineRegion()` initializes the Java Object Cache.

When you define the region, you can also set attributes and create a `CacheLoader` object. Attributes specify how the Java Object Cache manages objects. The `Attributes.setLoader()` method sets the name of `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.

> **See Also:** "Java Object Cache Environment" on page 14-10 and "Implementing a CacheLoader" on page 14-20

Defining a Cache Group

When you want to create an association between two or more objects within the cache, create a cache group. Objects are typically associated in a cache group because they need to 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 following code defines a cache group within the region named "`Test Application`":

```
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. A subregion's name space 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 a subregion's parent.

In the following example, cache subregion is defined within the region named `"Test Application"`.

```
final static String APP_NAME_ = "Test Application";
final static String SUBREGION_NAME_ = "Test SubRegion";
// obtain an instance of CacheAccess object to a named region
CacheAccess caccess = CacheAccess.getAccess(APP_NAME_);
// Create a SubRegion
caccess.defineSubRegion(SUBREGION_NAME_);
// Close the CacheAccess object
caccess.close();
```

Defining and Using Cache Objects

You may sometimes want to describe to the Java Object Cache how an individual object should be managed within the cache before the object is loaded. Management options can be specified 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, the Java Object Cache uses the default attributes set for the region, subregion, or group with which the object is associated.

Example 14–1 shows how to set attributes for a cache object.

Example 14–1 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

Generally, you should use the Java Object Cache to load objects automatically, as needed rather than using the application to directly manage objects in the cache. When an application directly manages objects, it uses the `CacheAccess.put()` method to insert objects into the cache. To take advantage of automatic loading, you use a `CacheLoader` object and implement a `load()` method to insert objects into the cache.

A `CacheLoader` can be associated with a region, subregion, a group, or an 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."

When an object is not in the cache, when an application calls `CacheAccess.get()` or `CacheAccess.preLoad()`, the `CacheLoader` executes the `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 `CacheAccess.get()` 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` that is defined for an individual object is used only to load the object.

> **Note:** A `CacheLoader` that is defined for a region, subregion, or group or for more than one cache object needs to be written with concurrent access in mind. The implementation should be thread-safe, since the `CacheLoader` object is shared.

Using CacheLoader Methods Within the Load Method

The Java Object Cache supports several `CacheLoader` methods that you can use within a `load()` method implementation. Table 14–4 summarizes the available `CacheLoader` methods.

Table 14–4 CacheLoader Methods Used in load()

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

Example 14–2 shows a `CacheLoader` using the `cacheLoader.netSearch()` method to check if the object being loaded is available in distributed Java Object Cache caches. If the object is not found using `netSearch()`, the load method uses a more expensive call to retrieve the object (an expensive call might involve an HTTP

connection to a remote Web site or a connection to the Oracle9*i* Database Server). For this example, the Java Object Cache stores the result as a `String`.

Example 14–2 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 a 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, destroy the object using the `CacheAccess.destroy()` method.

An object may be invalidated automatically based on the `TimeToLive` or `IdleTime` attributes. 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 `CacheAccess.invalidate()`.

```
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 access
```

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 may be destroyed automatically based on the `TimeToLive` or `IdleTime` attributes. By default, objects are invalidated and are not destroyed. If the objects need to 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.

```
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
```

Setting Cache Configuration Properties

During initialization, the Java Object Cache sets values for configuration properties. Table 14–5 lists the configuration properties for Java Object Cache. By default, the first time a region is created, or the default region is accessed, the Java Object Cache initializes the configuration properties. When the Java Object Cache is installed, the installer updates values for certain administrative properties and places the updated values in the `javacache.properties` configuration file, in the directory `$ORACLE_HOME/javacache/admin` on UNIX or in `%ORACLE_HOME\javacache\admin` on Windows NT.

You can modify the `javacache.properties` file to use values other than the default configuration property values. For configuration property values that are not specified in `javacache.properties`, the Java Object Cache uses the default values included in Table 14–5.

When the Java Object Cache is initialized, it uses either the default administration property values, or values specified in `javacache.properties`. No explicit method calls are required to configure the administrative properties using this initialization technique. The Java Object Cache also supports other initialization techniques (see the `Cache` object methods in the Javadoc for details).

The format for the values in the properties `javacache.properties` file is:

`property=value`

A # character in a configuration file starts a comment. When the # is in the first column, the entire line is a comment. When the # is occurs after a property value specification, it applies to the remainder of the line.

Table 14–5 lists the valid property names and lists the valid types for each property.

Table 14–5 Java Object Cache Configuration Properties

Configuration Property	Description	Type
`cleanInterval`	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 associated with the object. **Default value:** `60`	`int`
`discoveryAddress`	Specifies the address that the Java Object Cache initially contacts to join the caching system, when using distributed caching. The value is in the form, *`hostname:port`*. If the *`hostname`* is omitted, `localhost` is used. If the Java Object Cache spans systems, a comma-separated list of host names and ports should be included, with one *`hostname`*:*`port`* pair specified for each node. **Default Value:** `:12345` (this is equivalent to `localhost:12345`).	`String`
`diskPath`	Specifies the absolute path to the root for the disk cache (a directory). If this attribute is not set, disk caching is not available. **Default value:** `null`	`String`
`distribute`	Indicates whether the cache is distributed. Updates and invalidation for objects that have the `distribute` property set are propagated to other caches known to the Java Object Cache. If the `distribute` property is set to `false`, all objects are treated as local, even when the attributes set on objects are set to distribute. **Default value:** `false`	`boolean`
`logFileName`	Specifies the log file name for the default logger implementation. **Default value:** `$ORACLE_HOME/javacache/admin/logs/javacache.log` on UNIX or `%ORACLE_HOME%\javacache\admin\logs\javacache.log` on Windows NT	`String`
`logger`	Specifies the class name for the object that implements the `CacheLogger` interface. The object is instantiated when the Java Object Cache is initialized. **Default value:** `oracle.ias.cache.DefaultCacheLogger`	`String`

Table 14–5 Java Object Cache Configuration Properties (Cont.)

Configuration Property	Description	Type
`logSeverity`	Specifies the logging severity level used for initializing the logger. The valid values are: ▪ `-1 CacheLogger.OFF` ▪ `0 CacheLogger.FATAL` ▪ `3 CacheLogger.ERROR` ▪ `4 CacheLogger.DEFAULT` ▪ `6 CacheLogger.WARNING` ▪ `7 CacheLogger.TRACE` ▪ `10 CacheLogger.INFO` ▪ `15 CacheLogger.DEBUG` **Default value:** `CacheLogger.DEFAULT`	`int`
`maxObjects`	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`	`int`
`maxSize`	Specifies the maximum size of the memory, in megabytes, available to the Java Object Cache. **Default value:** `10`	`int`

Note: Configuration properties are distinct from the Java Object Cache attributes that you specify using the `Attributes` class.

Implementing a Cache Event Listener

There are a number of events that can occur in the life cycle of a cached object, including object creation and object invalidation. This sections shows how an application can be notified when cache events occur.

To receive notification of an object's creation, 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, the listener only runs when the group, region, or subregion itself is invalidated. Invalidating a member does not trigger the event. `Attributes.setCacheEventListener()` takes a boolean argument, that if `true`, applies the event listener to each member of the region, subregion, or group, rather than to the region, subregion, or group itself. In this case, the invalidation of an object within the region, subregion, or group triggers the event.

The `CacheEventListener` interface has one method, `handleEvent()`. This method takes a single argument, a `CacheEvent` object that extends `java.util.EventObject`. This object has two methods `getID()`,which returns the type of event (`OBJECT_INVALIDATION` or `OBJECT_UPDATED`), and `getSource()`, which returns the object being invalidated. For group objects, the `getSource()` method returns the name of the group.

The `handleEvent()` method is executed in the context of a background thread that the Java Object Cache manages. Avoid using JNI code in this method, as the expected thread context may not be available.

Example 14–3 shows how a `CacheEventListener` is implemented and associated with an object or a group.

Example 14–3 Implementing a CacheEventListener

```
import oracle.ias.cache.*;
   // A CacheEventListener for a cache object
   class MyEventListener implements
   CacheEventListener  {

       public void handleEvent(CacheEvent ev)
       {
          MyObject obj = (MyObject)ev.getSource();
          obj.cleanup();
        }

       // A CacheEventListener for a group object
       class MyGroupEventListener implements CacheEventListener {
       public void handleEvent(CacheEvent ev)
       {
          String groupName = (String)ev.getSource();
          app.notify("group " + groupName + " has been invalidated");
```

```
      }
   }
```

Use the `Attributes.listener` attribute to specify the `CacheEventListener` for a region, subregion, group, or object.

Example 14–4 shows how to set a cache event listener on an object. Example 14–5 shows how to set a cache event listener on a group.

Example 14–4 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 14–5 Setting a Cache Event Listener on a Group

```
import oracle.ias.cache.*;
try
{
   CacheAccess cacc = CacheAccess.getAccess(myRegion);
   Attributes attr = new Attributes ();

   MyGroupEventListener listener = new MyGroupEventListener();
   attr.setCacheEventListener(CacheEvent.OBJECT_INVALIDATED, listener);

   cacc.defineGroup("myGroup", attr);
   //....
```

```
   cacc.close();

}catch(CacheException ex)
{
   // handle exception
}
```

Restrictions and Programming Pointers

This section covers restrictions and programming pointers to keep in mind when using the Java Object Cache.

1. The `CacheAccess` object should not be shared between threads. This object represents a user to the caching system. The `CacheAccess` object contains the current state of the user's access to the cache: what object is currently being accessed, what objects are currently owned, and so on. Trying to share the `CacheAccess` object is unnecessary and can result in nondeterministic behavior.
2. A `CacheAccess` object only holds a reference to one cached object at a time. If multiple cached objects are being accessed concurrently, multiple `CacheAccess` objects should be used. For objects stored in memory, the consequences of not doing this are minor since Java prevents the cached object from being garbage collected even if the cache believes it is not being referenced. For disk objects, if the cache reference is not maintained, the underlying file could be removed by another user or by time-based invalidation, causing unexpected exceptions. To optimize resource management, you should keep the cache reference open as long as the cached object is being used.
3. A `CacheAccess` object should always be closed when it is no longer being used. The `CacheAccess` objects are pooled. They acquire other cache resources on behalf of the user. If the access object is not closed when it is not being used, these resources are not returned to the pool and are not cleaned up until they are garbage collected by the Java VM. If `CacheAccess` objects are continually allocated and not closed, available resources and a consequent degradation in performance may occur.
4. 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 only visible to the cache instance where they were loaded. If that cache instance goes away for any reason, the objects it manages, including on disk, are lost. If an object needs to survive process termination, both the object and the cache need to be defined `DISTRIBUTE`.

5. The cache configuration, also called the cache environment, is local to a cache, this includes the region, subregion, group, and object definitions. The cache configuration is not saved to disk or propagated to other caches. The cache configuration should be defined during the initialization of the application.
6. If a `CacheAccess.waitForResponse()` or `CacheAccess.releaseOwnership()` method call times out, it must be called again until it returns successfully. Call these methods with a `-1 timeout` value to free up resources, and eliminate waits.
7. When a group is destroyed or invalidated, distributed definitions take precedence over local definitions. That is, if the group is distributed, all objects in the group will be invalidated or destroyed across the entire cache system even if the individual objects or associated groups are defined as local. If the group is defined as local, local objects within the group are invalidated locally, while distributed objects are invalidated throughout the entire cache system.
8. 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.
9. In general, objects stored in the cache should be loaded by the system class loader defined in the `CLASSPATH` when the Java VM is initialized, rather than by a user defined class loader. Specifically, any objects that are shared between applications or may be saved or spooled to disk need to be defined in the system `CLASSPATH`. Failure to do so may result in `ClassNotFoundExceptions` or `ClassCastExceptions`.
10. On some systems, the open file descriptors may 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 may be an appropriate value for the number of open file descriptors.
11. When configured in either local or distributed mode, at startup, one active Java Object Cache cache is created in a Java VM process (that is, in the program running in the Java VM 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:

Configuring Properties for Using the Disk Cache

To configure the Java Object Cache to use a disk cache, set the value of the `diskPath` configuration property in the `javacache.properties` file.

Setting the diskPath Configuration Property

To configure the Java Object Cache to use a disk cache, the `diskPath` property in the configuration properties file should be set to the path of the root directory for the disk cache. The default value for `diskPath` is null, which specifies that the Java Object Cache should not enable the disk cache.

Note: when operating in distributed mode. To share disk cache files, all caches cooperating in the same cache system must specify values for the `diskPath` property that represent the same physical disk. However, the values specified for the `diskPath` do not need to be the same.

If you configure the `diskPath` properties to represent different locations on the same or different physical disks, the disk cache objects are not shared.

See Also: "Setting Cache Configuration Properties" on page 14-24

Local and Distributed Disk Cache Objects

This section covers the following topics:

Local Objects

When operating in local mode, 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 only visible 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 dies.

Distributed Objects

When operating in distributed mode, disk cache objects are shared by all caches that have access to the file system hosting the disk cache. This configuration allows for better utilization of disk resources and allows disk objects to persist beyond the life of the Java Object Cache process. Distributed memory objects are not shared by all caches since individual copies of each memory object reside in the individual caches across the system.

Objects stored in the disk cache are identified using the concatenation of the path 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.

> **See Also:** "Automatically Adding Objects" on page 14-32 for information on using the `SPOOL` attribute

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 may 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. The Java Object Cache automatically moves objects from the memory cache to the disk cache in two cases.

- When space is running out in the memory cache, the Java Object Cache searches through the cache, looking for memory objects that are not currently accessed. These memory objects may be removed from the cache. If the memory object is defined with the `SPOOL` attribute set, the memory object is written to disk before it is removed. Spooling saves the memory object to the disk cache, and avoids

re-creating the object when or if it is needed again. You should set the SPOOL attribute for objects that are expensive to create, especially if the time required to create the object is greater than the cost of loading the object from disk.

- StreamAccess objects are automatically loaded to disk cache. StreamAccess objects give the Java Object Cache latitude as to how the object is accessed. Smaller StreamAccess objects can be accessed from memory or the disk cache, while larger StreamAccess objects are streamed directly from disk. The Java Object Cache determines how to store the StreamAccess object based on the size of the object and the capacity of the cache.

See Also: "Cache Object Attributes" on page 14-12 and "Working with StreamAccess Objects" on page 14-35

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 synchronously 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, the event is recorded in the Java Object Cache log and the save operation continues with the next object.

Using Objects that Reside Only in Disk Cache

Objects that you only access 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 disk object's attributes are not defined for the disk object, set them using the `createDiskObject()` 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 specified attributes.

> **Note:** If you want to share a disk cache object between distributed caches in the same cache system, you must define the `DISTRIBUTE` attribute when the disk cache object is created. This attribute cannot be changed for the disk cache object 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, appropriate to its needs.

Disk objects are stored on a local disk and accessed directly from the disk by the application using the Java Object Cache. Disk objects may be shared by all Java Object Cache processes, or they may be local to a particular process, depending on the setting for the `DISTRIBUTE` attribute (and the mode the Java Object Cache is running in, either distributed, or local).

Example 14–6 shows a loader object that loads a disk object into the cache.

See Also: "Implementing a CacheLoader" on page 14-20 and "Java Object Cache Environment" on page 14-10

Example 14–6 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
      {
         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 14–7 shows application code that uses an Java Object Cache disk object. This example assumes the region named "Stock-Market" is already defined with the "YourObjectLoader" loader set up in Example 14–6 as the default loader for the region.

Example 14–7 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

StreamAccess objects are objects that are accessed as a stream and are 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, while larger StreamAccess objects are streamed directly from disk. The Java Object Cache automatically determines where to access the StreamAccess object based on the size of the object and the capacity of the cache.

The user is always presented with a stream object, an `InputStream` for reading and an `OutputStream` for writing, regardless of whether the object is in a file or in memory. The StreamAccess object allows the Java Object Cache user to always access the object in a uniform manner, without regard to object size or resource availability.

Creating a StreamAccess Object

To create a StreamAccess object, call the `CacheLoader.createStream()` method from the `CacheLoader.load()` method when the object is loaded into the cache. The `createStream()` method returns an `OutputStream` object. The `OutputStream` object can be used to load the object into the cache.

If the attributes have not already been defined for the object, they should be set using the `createStream()` method. The system manages local and distributed disk objects differently; the determination of local or distributed is made when the system creates the object, based on the attributes.

Note: If you want to share a StreamAccess object between distributed caches in the same cache system, you must define the `DISTRIBUTE` attribute when the StreamAccess object is created. This attribute cannot be changed after the object is created.

Example 14–8 shows a loader object that loads a StreamAccess object into the cache.

Example 14–8 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, stored as a static across the entire cache instance, while the objects within the pool object are private objects that the Java Object Cache manages. Users access individual objects within the pool with a check out, using a pool access object, and then return the objects to the pool when they are no longer needed.

This section covers the following topics:

- Creating Pool Objects
- Using Objects from a Pool
- Implementing a Pool Object Instance Factory

Creating Pool Objects

To create a pool object, use `CacheAccess.createPool()`. The `CreatePool()` method takes as arguments a `PoolInstanceFactory`, and an `Attributes` object, plus two integer arguments. The integer arguments specify the maximum pool size and the minimum pool size. By supplying a group name as an argument to `CreatePool()`, a pool object is associated with a group.

Attributes, including `TimeToLive` or `IdleTime` may 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. The minimum is specified 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 may 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.

See Also:

- "Implementing a Pool Object Instance Factory" on page 14-39
- "Java Object Cache Environment" on page 14-10

Example 14–9 shows how to create a pool object.

Example 14–9 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 14–10 shows the calls 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 14–10 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 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 14–11 shows the calls required when implementing a `PoolInstanceFactory`.

Example 14–11 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();
    }
}
```

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 machine or remotely across the network. Local mode provides a decentralized architecture that supports a very efficient cache system, with very limited overhead. 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 machine 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. Running in distributed mode has possible disadvantages. Specifically, significant system resources may be required when a large number of distributed objects need to be invalidated, when very large objects are updated, or when updates must be performed rapidly.

This section covers the following topics:

- Configuring Properties for Distributed Mode
- Using Distributed Objects, Regions, Subregions, and Groups
- Cached Object Consistency Levels

Configuring Properties for Distributed Mode

To configure the Java Object Cache to run in distributed mode, set the value of the `distribute` and `discoveryAddress` configuration properties in the `javacache.properties` file.

Setting the Distribute Configuration Property

To start the Java Object Cache in distributed mode, the `distribute` property should be set to `true` in the configuration file.

See Also: "Setting Cache Configuration Properties" on page 14-24

Setting the DiscoveryAddress Configuration Property

In distributed mode, invalidations, destroys, and replaces are propagated through the cache's messaging system. The messaging system requires a known hostname and port address to allow a cache to join the cache system when it is first initialized. Use the `discoveryAddress` property in the `javacache.properties` file to specify a list of hostname and port addresses.

By default, Java Object Cache sets the `discoveryAddress` to the value `:12345` (this is equivalent to `localhost:12345`). To eliminate conflicts with other software on the site, you should have your system administrator set the `discoveryAddress`.

If the Java Object Cache spans systems, a comma separated list of host name and port pairs should be included as the value for `discoveryAddress`, with one *`hostname:port`* pair specified for each node. This avoids any dependency on a particular machine being available or on the order the processes are started.

See Also: "Setting Cache Configuration Properties" on page 14-24

Note: All caches cooperating in the same cache system must specify the same set of hostname and port addresses. The address list, set with the `discoveryAddress` property defines the caches that make up a particular cache system. If the address lists vary, 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 may 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.

See Also: "Cached Object Consistency Levels" on page 14-46

Using the REPLY Attribute with Distributed Objects

When updating, invalidating, or destroying objects across multiple caches, it is 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 sender when a requested action has completed for the object with the `REPLY` attribute set. This also enables the wait for response feature for object updates, invalidates, or destroys, and requires the use of the blocking method `CacheAccess.waitForResponse()`.

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 accessed by the `CacheAccess` object. This allows the application to update a number of objects, then wait for all the replies.

Example 14–12 illustrates how to set an object as distributed and handle replies when the `REPLY` attribute is set. In this example, the attributes may also be set for the entire region. Attributes could also be set for a group or individual object, as appropriate for your application.

Example 14–12 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 may coordinate the update action. Setting the SYNCHRONIZE attribute enables synchronized updates and requires an application to obtain ownership of an object before the object is loaded or updated.

The SYNCHRONIZE attribute also applies to regions, subregions, and groups. When the SYNCHRONIZE attribute is applied to a region, subregion, or group, ownership of the region, subregion, or group must be obtained before an object can be loaded or replaced in the region, subregion, or group.

Setting the SYNCHRONIZE_DEFAULT attribute on a region, subregion, or group applies the SYNCHRONIZE attribute to all of the objects within the region, subregion, or group. Ownership must be obtained for the individual objects within the region, subregion, or group before they can be loaded or replaced.

> **Note:** You can also use the SYNCHRONIZE and SYNCHRONIZE_DEFAULT attributes with objects that are not distributed to control updates for the objects from multiple threads, where each thread uses the Java Object Cache.

To obtain ownership of an object, use CacheAccess.getOwnership(). Once 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.

Once 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 time out value is -1, ownership is released immediately without waiting for the responses from the other caches.

Example 14–13 Distributed Caching Using SYNCRHONIZE and SYNCHRONIZE_DEFAULT

```
import oracle.ias.cache.*;

CacheAccess cacc;
String     obj;
Attributes attr = new Attributes ();
MyLoader   loader = new MyLoader();

// mark the object for distribution and set synchronize attribute
attr.setFlags(Attributes.DISTRIBUTE|Attributes.SYNCHRONIZE);
attr.setLoader(loader);

//create region
CacheAccess.defineRegion("testRegion");
cacc = CacheAccess.getAccess("testRegion");
cacc.defineGroup("syncGroup", attr); //define a distributed synchronized group
cacc.defineObject("syncObject", attr); // define a distributed synchronized object
attr.setFlagsToDefaults()  // reset attribute flags

// define a group where SYNCHRONIZE is the default for all objects in the group
attr.setFlags(Attributes.DISTRIBUTE|Attributes.SYNCHRONIZE_DEFAULT);
cacc.defineGroup("syncGroup2", attr);
```

```
try
{
// try to get the ownership for the group don't wait more than 5 seconds
   cacc.getOwnership("syncGroup", 5000);
   obj = (String)cacc.get("testObject", "syncGroup"); // get latest object
   // replace the object with a new version
   cacc.replace("testObject", "syncGroup", obj + "new version");
   obj = (String)cacc.get("testObject2", "syncGroup"); // get a second object
   // replace the object with a new version
   cacc.replace("testObject2", "syncGroup", obj + "new version");
}

catch (TimeoutException ex)
{
   System.out.println("unable to acquire ownership for group");
   cacc.close();
   return;
}
try
{
   cacc.releaseOwnership("syncGroup",5000);
}
catch (TimeoutException ex)
{
   // tired of waiting so just release ownership
   cacc.releaseOwnership("syncGroup", -1));
}
try
{
   cacc.getOwnership("syncObject", 5000); // try to get the ownership for the object
   // don't wait more than 5 seconds
   obj = (String)cacc.get("syncObject");  // get latest object
   cacc.replace("syncObject", obj + "new version"); // replace the object with a new version
}
catch (TimeoutException ex)
{
   System.out.println("unable to acquire ownership for object");
   cacc.close();
   return;
}
try
{
   cacc.releaseOwnership("syncObject", 5000);
}
catch (TimeoutException ex)
```

```
{
   cacc.releaseOwnership("syncObject", -1)); // tired of waiting so just release ownership
}
try
{
   cacc.getOwnership("Object2", "syncGroup2", 5000); // try to get the ownership for the object
   // where the ownership is defined as the default for the group don't wait more than 5 seconds
   obj = (String)cacc.get("Object2", "syncGroup2"); // get latest object
   // replace the object with new version
   cacc.replace("Object2", "syncGroup2", obj + "new version");
}

catch (TimeoutException ex)
{
   System.out.println("unable to acquire ownership for object");
   cacc.close();
   return;
}
try
{
   cacc.releaseOwnership("Object2", 5000);
}
catch (TimeoutException ex)
{
   cacc.releaseOwnership("Object2", -1)); // tired of waiting so just release ownership
}
   cacc.close();
}
```

Cached Object Consistency Levels

Within the Java Object Cache, each cache manages its own objects locally within its Java VM process. In distributed mode, when using multiple processes or when the system is running on multiple sites, a copy of an object may exist in more than one cache.

The Java Object Cache allows you to specify the consistency level required between copies of objects that are available in multiple caches. The consistency level 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, an object should be defined 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 only visible to the local cache.

Propagating Changes Without Waiting for a Reply

To distribute object updates across distributed caches, an object should be defined 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. Using these attributes, notification occurs when a modification has completed in all caches. When `Attributes.REPLY` is set 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 Java Object Cache's highest level of consistency set the appropriate attributes on the region, subregion, group, or object to make objects act as synchronized objects.

On a region, subregion, or group, setting `Attributes.SYNCHRONIZE_DEFAULT` sets the `SYNCHRONIZE` attribute for all of the objects within the region, subregion, or group.

On an object, setting `Attributes.SYNCHRONIZE` 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. Using 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 does not effectively synchronize. With `Attributes.SYNCHRONIZE` set, the Java Object Cache forces the cache to synchronize its updates of the object, but does not prevent the Java programmer from obtaining a reference to the object and then modifying the object.

When using this level of consistency, with `Attributes.SYNCHRONIZE`, the `CacheLoader.load()` method should call `CacheLoader.netSearch()` before loading the object from an external source. Calling `CacheLoader.netSearch()` in the load method tells the Java Object Cache to search all other caches for a copy of the object. This prevents different versions of the object from being loaded into the cache from an external source.

Sharing Cached Objects in an OC4J Servlet

To take advantage of the Java cache's distributed functionality or to share a cached object among servlets, some minor modification to an applications deployment may be necessary. Any user-defined objects that will be shared among servlets or distributed among JVMs must be loaded by the system class loader. By default, objects loaded by a servlet are loaded by the context class loader. These objects are only visible 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 will be available to other servlets and to the cache on other JVMs.

With Jserv, this was accomplished by including the cached object in the classpath definition available when the Jserv process was started.

With OC4J, the system classpath is derived from the manifest of the `oc4j.jar` file and any associated JAR files, including `cache.jar`. The classpath in the environment is ignored. To include a cached object in the classpath for OC4J, the class file should be copied to `$ORACLE_HOME/javacache/sharedobjects/classes`

or added to the JAR file `$ORACLE_HOME/javacache/cachedobjects/share.jar`. Both the `classes` directory and the `share.jar` file have been included in the manifest for `cache.jar`.

15

Oracle HTTPS for Client Connections

This chapter describes the Oracle9*i*AS Containers for J2EE (OC4J) implementation of HTTPS that provides SSL functionality to client HTTP connections. The following topics are included:

Prerequisites

Please perform the following tasks before you attempt to use Oracle HTTPS:

- Install JDK version 1.2 or later.
- Ensure that the `CLASSPATH` environment variable includes the following jar files:
  - `javax-ssl-1_1.jar`
  - `jssl-1_1.jar`
- Add the Java SSL shared library to the shared library path:
  - **For UNIX:** `libnjss18.so` must be included in the library path specified by the `LD_LIBRARY_PATH` environment variable.
  - **For Windows NT:** `njss18.dll` must be included in the path specified by the `PATH` environment variable.

    See Also: Platform-specific documentation.

- Set the following Java security property so Oracle HTTPS can use Oracle Java SSL sockets:

  ```
  ssl.SocketFactory.provider=oracle.security.ssl.OracleSSLSocketFactoryImpl
  ```

  See Also: Sun Microsystems, Inc., JSSE (Java Secure Socket Extension) documentation for more information about setting system properties at:

  `http://www.java.sun.com`

Audience

To effectively use Oracle HTTPS, application developers should understand the basics of Java sockets programming and JSSE (Java Secure Socket Extension). They should also be familiar with the Sun Microsystems, Inc., `java.net` package, which supports network programming and the open source `HTTPClient` package that Oracle HTTPS is based on.

In addition, it is important for developers who use Oracle HTTPS to understand the fundamental concepts of public key infrastructure digital certificates and keys.

See Also:

- *Oracle9iAS Security Guide* for information about Oracle Wallet Manager, PKI, and security fundamentals.
- Documentation for the open source `HTTPClient` package which is available at `http://www.innovation.ch/java/HTTPClient`
- Documentation for JSSE and the `java.net` packages which is available at `http://www.java.sun.com`

About Oracle HTTPS

HTTPS is vital to securing client-server interactions. For many server applications HTTPS is handled by the Web server. However, any application that acts as a client, such as servlets that initiate connections to other Web servers, needs its own HTTPS implementation to make requests and to receive information securely from the server. Java application developers who are familiar with either the HTTP package, `HTTPClient`, or who are familiar with the Sun Microsystems, Inc., `java.net` package can easily use Oracle HTTPS to secure client interactions with a server.

Oracle HTTPS extends the `HTTPConnection` class of the open source `HTTPClient` package, which provides a complete HTTP client library. To support client HTTPS connections, several methods have been added to the `HTTPConnection` class that use the Oracle Java SSL class, `OracleSSLCredential`.

The following sections describe these components in further detail:

- HTTPConnection Class
- OracleSSLCredential Class

See Also: "Oracle HTTPS APIs" on page 15-11 for a description of the methods that have been added to the `HTTPConnection` class.

HTTPConnection Class

The `HTTPConnection` class is used to create new connections that use HTTP and related protocols such as HTTPS. To provide support for PKI (Public Key Infrastructure) digital certificates and wallets, the methods described in "Oracle HTTPS APIs" on page 15-11 have been added to this class.

> **See Also:** Documentation for the open source `HTTPClient` package which is available at:
>
> `http://www.innovation.ch/java/HTTPClient`

OracleSSLCredential Class

Security credentials are used to authenticate the server and the client to each other. Oracle HTTPS uses the Oracle Java SSL package, `OracleSSLCredential`, to load user certificates, trusted certificates (trust points), and private keys from base64 or DER-encoded certificates. (DER, part of the X.690 ASN.1 standard, stands for Distinguished Encoding Rules.)

The API for Oracle Java SSL requires that security credentials be passed to the HTTP connection before the connection is established. The OracleSSLCredential class is used to store these security credentials. Typically, a wallet generated by Oracle Wallet Manager is used to populate the OracleSSLCredential object. Alternatively, individual certificates can be added by using an OracleSSLCredential class API. After the credentials are complete, they are passed to the connection with the `setCredentials` method.

> **See Also:** "Oracle HTTPS APIs" on page 15-11 for a description of the `OracleSSLCredential` class.

Overview of Oracle HTTPS Features

Oracle HTTPS, based on the open source HTTP package, HTTPClient 3.2, supports HTTP 1.0 and HTTP 1.1 connections between a client and a server. To provide SSL functionality, new methods have been added to the `HTTPConnection` class of this package. These methods are used in conjunction with Oracle Java SSL to support cipher suite selection, security credential management with Oracle Wallet Manager, security-aware applications, and other features that are described in the following sections.

In addition to the functionality included in the `HTTPClient` package, Oracle HTTPS supports the following:

- Multiple cryptographic algorithms
- Certificate and key management with Oracle Wallet Manager
- Limited support for the `java.net.URL` framework

In addition, Oracle HTTPS uses the `HTTPClient` package to support

- HTTP tunneling through proxies
- HTTP proxy authentication

The following sections describe Oracle HTTPS features in detail:

- SSL Cipher Suites Supported by Oracle HTTPS
- Certificate and Key Management with Oracle Wallet Manager
- Access Information About Established SSL Connections
- Security-Aware Applications Support
- java.net.URL Framework Support

SSL Cipher Suites Supported by Oracle HTTPS

Before data can flow through an SSL connection, both sides of the connection must negotiate common algorithms to be used for data transmission. A set of such algorithms combined to provide a mix of security features is called a *cipher suite*. Selecting a particular cipher suite lets the participants in an SSL connection establish the appropriate level for their communications.

Oracle HTTPS supports cipher suites with the following options:

- Key exchange of 512, 768, or 1024 bit asymmetric keys using the following algorithms:
  - RSA
  - Diffie-Hellman
- NULL encryption, or symmetric key encryption with 40 and 128 bit symmetric keys using the following algorithms:
  - RC4 stream cipher
  - DES, DES40, and 3DES-EDE, in *Cipher Block Chaining (CBC)* mode

  Note: With NULL encryption, SSL is only used for authentication and data integrity purposes.

- Message Authentication Code using MD5 or SHA1 data integrity.

Table 15–1 lists all of the cipher suites that are supported by Oracle HTTPS.

Table 15–1 Cipher Suites Supported By Oracle HTTPS

Cipher Suite	Authentication	Encryption	Data Integrity
SSL_RSA_WITH_3DES_EDE_CBC_SHA	RSA	3DES EDE CBC	SHA1
SSL_RSA_WITH_RC4_128_SHA	RSA	RC4 128	SHA1
SSL_RSA_WITH_RC4_128_MD5	RSA	RC4 128	MD5
SSL_RSA_WITH_DES_CBC_SHA	RSA	DES CBC	SHA1
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA	DH anon	3DES EDE CBC	SHA1
SSL_DH_anon_WITH_RC4_128_MD5	DH anon	RC4 128	MD5
SSL_DH_anon_WITH_DES_CBC_SHA	DH anon	DES CBC	SHA1
SSL_RSA_EXPORT_WITH_RC4_40_MD5	RSA	RC4 40	MD5
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA	RSA	DES40 CBC	SHA1
SSL_DH_anon_EXPORT_WITH_RC4_40_MD5	DH anon	RC4 40	MD5
SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA	DH anon	DES40 CBC	SHA1
SSL_RSA_WITH_NULL_SHA	RSA	NULL	SHA1
SSL_RSA_WITH_NULL_MD5	RSA	NULL	MD5

Certificate and Key Management with Oracle Wallet Manager

You can use Oracle Wallet Manager to generate public/private key pairs and certificate requests. A signed certificate request and the appropriate trusted certificates must be added to produce a complete Oracle wallet.

You can export a complete wallet with a certificate in *Ready* status, in a BASE64-formatted file, using the menu option *Operation ->Export Wallet*. This file can be used to add SSL credentials in a Java SSL-based program.

> **See Also:** *Oracle9i Application Server Security Guide* for information about Oracle Wallet Manager.

Access Information About Established SSL Connections

Users can access information about established SSL connections using the `getSSLSession` method of Oracle HTTPS. After a connection is established, users can retrieve the cipher suite used for the connection, the peer certificate chain, and other information about the current connection.

> **See Also:** "Oracle HTTPS APIs" on page 15-11 for a description of the `getSSLSession` method.

Security-Aware Applications Support

Oracle HTTPS uses Oracle Java SSL to provide security-aware applications support. When security-aware applications do not set trust points, Oracle Java SSL allows them to perform their own validation letting the handshake complete successfully only if a complete certificate chain is sent by the peer. With Oracle HTTPS, the connection completes successfully when no trust points are set if the server sends the client a complete certificate chain that starts from the root CA (Certifying Authority) and ends with the server certificate. This feature is useful when there is a large number of trust points stored in a database, and the application is constrained from passing all of them to the SSL layer.

After the handshake is complete, the application must obtain the SSL session information and perform any additional validation for the connection.

Security-unaware applications that need the trust point check must ensure that trust points are set in the application.

> **See Also:** *Oracle Advanced Security Administrator's Guide* for information about Oracle Java SSL.

java.net.URL Framework Support

The `HTTPClient` package provides basic support for the `java.net.URL` framework with the `HTTPClient.HttpUrlConnection` class. However, many of the Oracle HTTPS features are supported through system properties only.

Features that are only supported through system properties are

- cipher suites selection option
- confidentiality only option
- server authentication option
- mutual authentication option
- security credential management with Oracle Wallet Manager

> **Note:** If the `java.net.URL` framework is used, then set the `java.protocol.handler.pkgs` system property to select the `HTTPSConnection` package as a replacement for the JDK client as follows:
>
> ```
> java.protocol.handler=HTTPClient
> ```
>
> **See Also:**
>
> - "Specifying Default System Properties" on page 15-9 for information about setting Java system properties.
> - Documentation for the `java.net.URL` framework at
>
> `http://java.sun.com`

Specifying Default System Properties

For many users of HTTPS it is desirable to specify some default properties in a non-programmatic way. The best way to accomplish this is through Java system properties which are accessible through the `java.lang.System` class. These properties are the only way for users of the `java.net.URL` framework to set security credential information. Oracle HTTPS recognizes the following properties:

- javax.net.ssl.KeyStore
- javax.net.ssl.KeyStorePassword
- Oracle.ssl.defaultCipherSuites

The following sections describe how to set these properties.

> **See Also:** Documentation that describes setting Java system properties at
>
> `http://www.java.sun.com`

javax.net.ssl.KeyStore

This property can be set to point to the text wallet file exported from Oracle Wallet Manager that contains the credentials that are to be used for a specific connection. For example:

```
javax.net.ssl.KeyStore=/etc/ORACLE/WALLETS/Default/default.txt
```

where *default.txt* is the name of the text wallet file that contains the credentials.

If no other credentials have been set for the HTTPS connection, then the file set by this property is opened when a handshake first occurs. If any errors occur while reading this file, then the connection fails and an `IOException` is thrown.

javax.net.ssl.KeyStorePassword

This property can be set to the password that is necessary to open the wallet file. For example:

```
javax.net.ssl.KeyStorePassword=welcome1
```

where *welcome1* is the password that is necessary to open the wallet file.

Potential Security Risk with Storing Passwords in System Properties

Storing the wallet file password as a Java system property can result in a security risk in some environments. To avoid this risk, use one of the following alternatives:

- If mutual authentication is not required for the application, then a text wallet that contains no private key should be used instead. To open these wallets, no password is necessary.
- If a password is necessary, then do not store it in a clear text file. Instead, load the property dynamically before the `HTTPConnection` is started by using `System.setProperty()`. Unset the property after the handshake is completed.

Oracle.ssl.defaultCipherSuites

This property can be set to a comma-delimited list of cipher suites. For example:

```
Oracle.ssl.defaultCipherSuites=
            SSL_RSA_WITH_DES_CBC_SHA,\
            SSL_RSA_EXPORT_WITH_RC4_40_MD5,\
            SSL_RSA_WITH_RC4_128_MD5
```

The cipher suites that you set this property to are used as the default cipher suites for new HTTPS connections.

> **See Also:** Table 15–1 on page 15-7 for a complete list of the cipher suites that are supported by Oracle HTTPS.

Oracle HTTPS APIs

This section describes the public classes and interfaces used by Oracle HTTPS. Oracle HTTPS uses the Oracle Java SSL class, `OracleSSLCredential`, and it extends the `HTTPConnection` class of the open source `HTTPClient` package. The following sections describe these packages:

- Public Class: HTTPConnection
- Public Class: OracleSSLCredential

Public Class: HTTPConnection

Because Oracle HTTPS extends the `HTTPConnection` class, only the methods that are added to that package for SSL support are described in the following. The fully qualified name of this class is `HTTPClient.HTTPConnection`.

`public void connect()`
Initiates a connection with the host, but does not perform any data transfer.

`public String[] getSSLEnabledCipherSuites()`
Returns a list of cipher suites enabled for this connection.

`public javax.net.ssl.SSLSession getSSLSession()`
Returns an `SSLSession` containing the information about the current connection.

`public javax.net.ssl.SSLSocketFactory getSSLSocketFactory()`
Returns the `SSLSocketFactory` used by the `HTTPConnection` to create `SSLSockets`.

`public oracle.security.ssl.OracleSSLCredential get SSLCredential()`
Returns the SSL credentials used by this connection.

`public void setSSLCredential (oracle.security.ssl.OracleSSLCredential)`
Sets the authentication context for the connection.

Parameters: `credential` - Authentication context contains the private key, certificate chains, and trusted certificates that are to be used in the SSL connection.

`public void setSSLEnabledCipherSuites(String[] suites) throws IllegalArgumentException`

Controls which particular cipher suites are enabled for use on this connection. The cipher suites must have been listed by `SSLSocketFactory.getSupportedCipherSuites()` as being supported. The method throws an `IllegalArgumentException` when one of the ciphers named by the parameter is not supported.

Parameters: `suites` - List of cipher suites.

Public Class: OracleSSLCredential

This public class extends `java.lang.Object`. The fully qualified name of this class is `oracle.security.ssl.OracleSSLCredential`.

Credentials are used to authenticate the server and the client to each other. `OracleSSLCredential` is used to load user certificates, trusted certificates (trust points), and private keys from base64 or DER-encoded certificates.

Constructor

`public OracleSSLCredential()`

Creates an empty OracleSSLCredential. An empty credential lets the socket connect to any peer that sends a complete certificate chain during the handshake.

Methods

`public void addTrustedCert(java.lang.String b64TrustedCert)`

Adds a trusted certificate to the credential.

Parameters: `b64TrustedCert` - A Base64 encoded X509 certificate.

`public void addTrustedCert(byte[] trustedCert)`

Adds a trusted certificate to the credential.

Parameters: `trustedCert` - A DER-encoded X509 trusted certificate.

`public void setPrivateKey(java.lang.String b64PvtKey, java.lang.String password)`

Adds a private key to the credential.

Parameters: `b64PvtKey` - A Base64 encoded X509 Private Key

`password` - The password needed to decipher the private key.

`public void setPrivateKey(byte[] pvtKey, java.lang.String password)`

Adds a private key to the credential.

Parameters: `b64PvtKey` - A DER-encoded X509 Private Key

`password` - The password needed to decipher the private key.

`public void addCertChain(java.lang.String b64certChainCert)`

Adds a certificate to the certificate chain. The certificate chain is sent along with the user certificate during the SSL handshake. It is used by the peer to verify the user certificate. The first certificate added to the certificate chain must be the Root CA certificate. Each subsequent certificate added must be signed by its immediate predecessor.

Parameters: `b64certChainCert` - A Base64 encoded X509 certificate.

`public void addCertChain(byte[] certChainCert)`

Adds a certificate to the certificate chain.

Parameters: `certChainCert` - A DER-encoded X509 certificate.

`public void setWallet(java.lang.String wltPath, java.lang.String password) throws java.io.IOException`

If Oracle Wallet Manager is used to create a wallet, the wallet can be exported in text format and used by JavaSSL. The text file must contain the user certificate, followed by the private key, the certificate chain, and any other trusted certificates. The method throws a `java.io.IOException` if the wallet cannot be opened.

Parameters: `wltPath` - The path name of the wallet

`password` - The password needed to decrypt the private key

Oracle HTTPS Example

The following is a simple program that uses Oracle HTTPS to connect to a Web server, send a GET request, and fetch a Web page. The complete code for this program is presented here followed by sections that explain how Oracle HTTPS is used to set up secure connections.

```
import HTTPClient.HTTPConnection;
import HTTPClient.HTTPResponse;
import oracle.security.ssl.OracleSSLCredential;
import java.io.IOException;

public class HTTPSConnectionExample
{
    public static void main(String[] args)
    {
        if(args.length < 4)
        {
            System.out.println(
            "Usage: java HTTPSConnectionTest [host] [port] " +
            "[wallet] [password]");
            System.exit(-1);
        }

        String hostname = args[0].toLowerCase();
        int port = Integer.decode(args[1]).intValue();
        String walletPath = args[2];
        String password = args[3];

        HTTPConnection httpsConnection = null;
        OracleSSLCredential credential = null;

        try
        {
            httpsConnection = new HTTPConnection("https", hostname, port);
        }
        catch(IOException e)
        {
            System.out.println("HTTPS Protocol not supported");
            System.exit(-1);
        }
```

```
try
{
    credential = new OracleSSLCredential();
    credential.setWallet(walletPath, password);
}
catch(IOException e)
{
    System.out.println("Could not open wallet");
    System.exit(-1);
}
httpsConnection.setSSLCredential(credential);

try
{
    httpsConnection.connect();
}
catch (IOException e)
{
    System.out.println("Could not establish connection");
    e.printStackTrace();
    System.exit(-1);
}

javax.security.cert.X509Certificate[] peerCerts = null;
try
{
    peerCerts =
      (httpsConnection.getSSLSession()).getPeerCertificateChain();
}
catch(javax.net.ssl.SSLPeerUnverifiedException e)
{
    System.err.println("Unable to obtain peer credentials");
    System.exit(-1);
}

String peerCertDN =
  peerCerts[peerCerts.length -1].getSubjectDN().getName();
peerCertDN = peerCertDN.toLowerCase();
if(peerCertDN.lastIndexOf("cn="+hostname) == -1)
{
    System.out.println("Certificate for " + hostname + " is issued to "
      + peerCertDN);
    System.out.println("Aborting connection");
    System.exit(-1);
}
```

```
        try
        {
            HTTPResponse rsp = httpsConnection.Get("/");
            System.out.println("Server Response: ");
            System.out.println(rsp);
        }
        catch(Exception e)
        {
            System.out.println("Exception occured during Get");
            e.printStackTrace();
            System.exit(-1);
        }
    }
}
```

Initializing SSL Credentials

This program example uses a wallet created by Oracle Wallet Manager to set up credential information. First the credentials are created and the wallet is loaded using

```
credential = new OracleSSLCredential();
credential.setWallet(walletPath, password);
```

After the credentials are created, they are passed to `HTTPSConnection` using

```
httpsConnection.setSSLCredential(credential);
```

The private key, user certificate, and trust points located in the wallet can now be used for the connection.

Verifying Connection Information

Although SSL verifies that the certificate chain presented by the server is valid and contains at least one certificate trusted by the client, that does not prevent impersonation by malicious third parties. An HTTPS standard that addresses this problem requires that HTTPS servers have certificates issued to their host name. Then it is the responsibility of the client to perform this validation after the SSL connection is established.

To perform this validation in this sample program, `HTTPSConnectionExample` establishes a connection to the server without transferring any data using the following:

```
httpsConnection.connect();
```

After the connection is established, the connection information, in this case the server certificate chain, is obtained with the following:

```
peerCerts = (httpsConnection.getSSLSession()).getPeerCertificateChain();
```

Finally the server certificate's common name is obtained with the following:

```
String peerCertDN = peerCerts[peerCerts.length -1].getSubjectDN().getName();
peerCertDN = peerCertDN.toLowerCase();
```

If the certificate name is not the same as the host name used to connect to the server, then the connection is aborted with the following:

```
if(peerCertDN.lastIndexOf("cn="+hostname) == -1)
{
    System.out.println("Certificate for " + hostname + " is issued to " +
        peerCertDN);
    System.out.println("Aborting connection");
        System.exit(-1);
}
```

Transferring Data

It is important to verify the connection information before data is transferred from the client or from the server. The data transfer is performed in the same way for HTTPS as it is for HTTP. In this sample program a `GET` request is made to the server using the following:

```
HTTPResponse rsp = httpsConnection.Get("/");
```

A

JAAS Provider APIs

This appendix describes the JAAS Provider public packages.

This appendix contains these topics:

JAAS Provider API Overview

This appendix provide brief descriptions of the JAAS Provider APIs. For detailed information on these APIs, see the JAAS Provider Javadoc available in the OC4J section of the Oracle9*i* Application Server Documentation Library.

Package oracle.security.jazn

Package `oracle.security.jazn` provides the classes and interfaces for Oracle's authorization / policy provider for the Java Authentication and Authorization Service (JAAS).

Besides providing a full implementation of `javax.security.auth.Policy`, the JAAS provider enhances JAAS in the following ways:

- Defines a realm-based user and role management API
- Defines an administrative API for administering the following aspects of the authorization policy:
  - Permission-to-user assignment
  - Permission-to-role assignment
  - User-to-role assignment
- Provides role-based access control (RBAC) support through the realm framework, with full support for role hierarchies.

Interfaces

Persistable

`Persistable` defines the basic behavior for a persistable object.

Classes

JAZNConfig

`JAZNConfig` provides a starting point for obtaining JAAS Provider-related objects and a centralized place for managing JAAS Provider properties

`JAZNConfig` enables you to run multiple JAAS provider instances. You can deploy several different applications using JAAS provider in the same Java virtual machine (JVM), each with different configurations. For example, you can have one application using JAAS provider with LDAP-based Oracle Internet Directory as the provider type and another application using JAAS provider with XML-Based Provider Type as the provider type in the same JVM.

JAZNContext

`JAZNContext` provides a starting point for obtaining JAAS Provider-related objects and a centralized place for managing the JAAS provider properties. `JAZNContext` is essentially a single-instance version of `JAZNConfig`.

JAZNPermission

`JAZNPermission` is for authorization permissions. A `JAZNPermission` contains a name (also referred to as a target name), but no actions list; you either have the named permission or you do not.

The target name is the name of the JAAS provider permission.

Table A–1 lists the possible target names for a `JAZNPermission`, describes what the permission allows, and describes the risks of granting the permission.

Table A–1 JAZNPermission Target Names

Permission Name	The Permission Allows	Risks of Allowing this Permission
`getPolicy`	The caller to retrieve the `JAZNPolicy` object	This enables someone to retrieve a `JAZNPolicy` object. Since the `JAZNPolicy` object can modify the JAAS Provider type, grant this permission only to the administrators.
`getRealmManager`	The caller to retrieve the `RealmManager` object	This enables someone to retrieve a `RealmManager` object. Since the `RealmManager` object can create, drop, and modify realms, grant this permission only to the administrators.
`getProperty.{propertyName}`	The caller to retrieve the value of the JAAS provider property named `{propertyName}`	Depending on the particular key for which access has been granted, the code may have access to the location of the backend server as well as security credentials used to access the backend server. Carefully protect this permission and grant it only to administrators.
`setProperty.{propertyName}`	The caller to set the value of the JAAS provider property named `{propertyName}`	This can include setting a new backend server and new credentials to access the backend server. Since this can bypass the enterprise policy, carefully protect this permission and grant it only to administrators.

JAZNWebAppConfig

`JAZNWebAppConfig` represents a `<jazn-web-app>` Configuration instance.

Exceptions

JAZNConfigException

`JAZNConfigException` represents an authorization exception.

JAZNException

`JAZNException` represents an authorization exception.

JAZNInitException

`JAZNInitException` is thrown when an initialization error occurs.

JAZNNamingException

`JAZNNamingException` is used to wrap a `javax.naming.NamingException`.

JAZNObjectExistsException

`JAZNObjectExistsException` is thrown when an attempt is made to create an object that already exists.

JAZNObjectNotFoundException

`JAZNObjectNotFoundException` is thrown when an attempt is made to access an object that does not already exist.

JAZNRuntimeException

`JAZNRuntimeException` represents an authorization exception.

Package oracle.security.jazn.login

Package `oracle.security.jazn.login` provides the classes and interfaces for administering Login Modules.

Classes

LoginModuleManager

`LoginModuleManager` extends `javax.security.auth.login. Configuration` by defining management methods (`add` / remove `AppConfigurationEntry`).

Package oracle.security.jazn.policy

Package `oracle.security.jazn.policy` provides the classes and interfaces for administering the authorization policy.

Interfaces

GlobalPolicy

`GlobalPolicy` represents the Global JAAS Provider Policy.

JAZNPolicy

`JAZNPolicy` represents the repository of authorization policies. More specifically, `JAZNPolicy` deals with the assignment of permissions or privileges to grantees (these can be users or roles or any valid grantee).

In order for a grant or revocation to succeed, the grantor or revoker (represented by the current subject) must have the relevant permissions granted to them.

In general, the methods that return a list or set represent a snapshot of a `JAZNPolicy` provider at the time of the query. If the JAAS provider is further modified, the returned set of permissions and roles may no longer be valid.

In general, `JAZNPolicy` implementation should cache the policy information, so that repeated calls using the same parameters do not result in repeated network round trips to the backing store.

`JAZNPolicy` also defines methods that change the persistent state of the JAAS Provider type (for example, grant or revoke `xx` methods). The implementation must ensure that whenever a grant or revoke is attempted, the relevant cache entries are invalidated.

PermissionClassManager

The `PermissionClassManager` is an utility to help manage permission classes.

PermissionClassManager represents the repository of all registered Permission classes. Registering a permission class allows access to stored metadata that provides specific information about a given permission's target, action, and/or description. Failure to register a given permission class will not affect JAAS provider's ability to use the permission class. That is, JAAS does not limit permission grants or revocations to those classes registered with the PermissionClassManager.

PolicyManager

PolicyManager defines basic methods for managing JAAS Provider policies.

PrincipalClassManager

The PrincipalClassManager is an utility to help manage principal classes.

PrincipalClassManager represents the repository of all registered Principal classes. Registering a principal class allows access to stored metadata that provides specific information about a given principal's name and description. Failure to register a given principal class will not affect the JAAS provider's ability to use the principal class. That is, the JAAS provider recognizes all principal classes whether or not they have been registered with the PrincipalClassManager.

RealmPolicy

RealmPolicy is a Realm-specific Policy.

Classes

AdminPermission

AdminPermission represents the right to administer a permission. Given a Permission p, the grantee of AdminPermission(p) is granted the right to:

- Grant or revoke permissions implied by p (say p')
- Grant or revoke AdminPermission(p')

For example:

```
p = java.io.FilePermission("/home/frank/-","read,write");
```

If grantee frank is granted AdminPermission(p), then frank is granted the following rights:

- The right to further grant or revoke `p'` (that is, read and write privileges for any file in the file system under `/home/frank`) to and from other grantees
- The right to further grant or revoke `AdminPermission(p')`

Consider the following information:

- An `AdminPermission` embedding another `AdminPermission` is not supported. There is no need to do so, since granting a grantee `AdminPermission(p)` implies that the grantee can further grant/revoke `AdminPermission(p')`
- Granting a grantee `AdminPermission(p)` does not imply granting the grantee. That must be granted separately.

Grantee

`Grantee` represents a grantee in a policy entry.

PermissionClassDesc

`PermissionClassDesc` defines the descriptor (metadata) for a Permission class.

PrincipalClassDesc

`PrincipalClassDesc` defines the descriptor (metadata) of a Principal class.

RoleAdminPermission

The grantee of `RoleAdminPermission` is granted the right to further grant or revoke the target role.

Package oracle.security.jazn.realm

Package `oracle.security.jazn.realm` provides the classes and interfaces for the realm framework.

Interfaces

InitRealmInfo.RealmType

`InitRealmInfo.RealmType` defines the different realm types supported by JAAS Provider.

Realm

`Realm` provides access to a store of roles and users. The JAAS provider separates role management from user management by providing each realm instance with its own `UserManager` for user management and `RoleManager` for role management.

`Realm` defines methods for managing realm's metadata (properties) and getting its `UserManager` and `RoleManager`.

Realm.LDAPProperty

`Realm.LDAPProperty` defines the LDAP properties applicable for creating a realm (user manager and role manager) using an LDAP directory as a backing store.

RealmPrincipal

`RealmPrincipal` extends from `java.security.Principal`. It is a principal associated with a realm instance.

RealmRole

`RealmRole` is a role associated with a realm. It can be associated with a group of privileges or roles.

RealmUser

`RealmUser` is a user associated with a realm. This is an empty interface for tagging objects as being `RealmUser` objects. It differs from `RealmRole` in that it cannot contain other roles.

RoleManager

`RoleManager` defines the APIs for managing roles in a realm.

UserManager

`UserManager` defines the APIs for managing users in a realm.

Classes

InitRealmInfo

`InitRealmInfo` is a placeholder for specifying realm properties when creating a new realm.

RealmLoginModule

`RealmLoginModule` is a realm-based *login module*.

RealmManager

`RealmManager` manages realms.

RealmPermission

`RealmPermission` is defined to represent permissions for a realm. It extends from `java.security.Permission`, and is used like any regular Java permission. `RealmPermission` consists of the name of the realm (also known as permission target name) and a set of actions specifying *privileges* applicable to that realm. The target name of a `RealmPermission` instance is the name of the realm in question. The individual action name is specific to the realm in question and is system-defined.

Table A–2 lists all the system-defined `RealmPermission` action names.

Table A–2 RealmPermission Action Names

Permission Action	Enables User To
`createRealm`	Create realms
`dropRealm`	Drop realms
`createUser`	Create users in the target realm
`dropUser`	Drop users in the target realm
`createRole`	Create roles in the target realm
`dropRole`	Drop roles in the target realm
`modifyRole`	Modify roles in the target realm
`grantRole`	Grant roles in the target realm
`revokeRole`	Revoke roles from the target realm

B

JAAS Provider Standards and Samples

This appendix provides supplemental samples and standards.

This appendix contains these topics:

- Sample jazn-data.xml Code
- Supplemental Code Samples

Sample jazn-data.xml Code

This section presents a sample `jazn-data.xml` file which illustrates the specific DTD standards that XML files must conform to. This `jazn-data.xml` file contains one realm, `jazn.com`, four users (three with obfuscated passwords) and three roles.

See Also:

- "DTD for jazn-data.xml" on page 6-34
- "Realm Management in XML-Based Environments" on page 3-25
- "Managing XML-Based Provider Data with the XML Schema" on page 6-33 for further information on managing JAAS Provider in XML-based provider environment
- "Other Utilities" on page 6-36 for further information on the `PermissionClassManager`, `PrincipalClassManager`, and `LoginModuleManager`

Example B–1 Sample jazn-data.xml File

```
<jazn-data>

<!--JAZN Realm Data -->

   <jazn-realm>
        <realm>
             <name>jazn.com</name>
             <users>
                  <user>
                       <name>admin</name>
                       <displayName>Realm Administrator</displayName>
                       <description>Administrator for this realm</description>
                       <credentials>Qj+w7NJulLM=</credentials>
                  </user>
                  <user>
                       <name>user</name>
                       <description>The default guest</description>
                       <credentials>wEE6aA==</credentials>
                  </user>
```

```
     <user>
          <name>anonymous</name>
          <description>The default guest/anonymous
                    user</description>
     </user>
     <user>
          <name>SCOTT</name>
          <displayName>SCOTT</displayName>
          <credentials>DppF6Lo4</credentials>
     </user>
</users>
<roles>
     <role>
          <name>guests</name>
          <members>
               <member>
                    <type>user</type>
                    <name>admin</name>
               </member>
               <member>
                    <type>user</type>
                    <name>user</name>
               </member>
               <member>
                    <type>user</type>
                    <name>anonymous</name>
               </member>
          </members>
     </role>
     <role>
          <name>administrators</name>
          <displayName>Realm Admin Role</displayName>
          <description>Administrative role for this
                       realm</description>
          <members>
               <member>
                    <type>user</type>
                    <name>admin</name>
               </member>
          </members>
     </role>
```

```
                    <role>
                         <name>users</name>
                         <members>
                              <member>
                                   <type>user</type>
                                   <name>admin</name>
                              </member>
                              <member>
                                   <type>user</type>
                                   <name>user</name>
                              </member>
                         </members>
                    </role>
               </roles>
          </realm>
</jazn-realm>

<!--JAZN Policy Data -->
<jazn-policy>
          <grant>
               <grantee>
                    <principals>
                         <principal>
                              <realm>jazn.com/realm>
                              <type>role/type>
                              <class>oracle.security.jazn.spi.xml.XMLRealmRole
                                  </class>
                                  <name>jazn.com/administrators/name>
                         </principal>
                    </principals>
               </grantee>
               <permissions>
                   <permission>
                        <class>oracle.security.jazn.realm.RealmPermission</class>
                        <name>jazn.com</name>
                        <actions>modifyrealmmetadata</actions>
                   </permission>
                   <permission>
                        <class>com.evermind.server.AdministrationPermission
                                  </class>
                        <name>administration</name>
                        <actions>administration</actions>
                   </permission>
```

```
<permission>
     <class>oracle.security.jazn.policy.AdminPermission</class>
     <name>oracle.security.jazn.realm.
           com$modifyrealmmetadata</name>
</permission>
<permission>
     <class>oracle.security.jazn.policy.AdminPermission</class>
     <name>oracle.security.jazn.realm.
         RealmPermission$jazn.com$droprealm</name>
</permission>
<permission>
     <class>oracle.security.jazn.policy.RoleAdminPermission
           </class>
     <name>jazn.com/*</name>
</permission>
<permission>
     <class>oracle.security.jazn.policy.AdminPermission</class>
     <name>oracle.security.jazn.policy.
           RoleAdminPermission$jazn.com/*$</name>
</permission>
<permission>
     <class>oracle.security.jazn.policy.AdminPermission</class>
     <name>oracle.security.jazn.realm.
           RealmPermission$jazn.com$droprole</name>
</permission>
<permission>
     <class>com.evermind.server.rmi.RMIPermission</class>
     <name>login</name>
</permission>
<permission>
     <class>oracle.security.jazn.realm.RealmPermission</class>
     <name>jazn.com</name>
     <actions>droprealm</actions>
</permission>
<permission>
     <class>oracle.security.jazn.policy.AdminPermission</class>
     <name>oracle.security.jazn.realm.RealmPermission$jazn.
           com$createrole</name>
</permission>
<permission>
     <class>oracle.security.jazn.policy.AdminPermission</class>
     <name>oracle.security.jazn.realm.RealmPermission$jazn.
          com$createrealm</name>
</permission>
```

```
                    <permission>
                         <class>oracle.security.jazn.realm.RealmPermission</class>
                         <name>jazn.com</name>
                         <actions>createrealm</actions>
                    </permission>
               </permissions>
          </grant>
     </jazn-policy>

<!-- Permission Class Data -->
     <jazn-permission-classes>
          <permission-class>
               <name>JAZNPermission</name>
               <description>To govern access to JAZN API</description>
               <type>jdk</type>
               <class>oracle.security.jazn.JAZNPermission</class>
                    <target-descriptors>
                         <target-descriptor>
                              <name>*</name>
                              <description>Access to ALL of JAZN API</description>
                         </target-descriptor>
                    </target-descriptors>
                    <action-descriptors>
                    </action-descriptors>
          </permission-class>
     </jazn-permission-classes>

<!-- Principal Class Data -->
     <jazn-principal-classes>
          <principal-class>
               <name>SolarisPrincipal</name>
               <description>Solaris Principal</description>
               <type>jdk</type>
               <class>com.sun.security.auth.SolarisPrincipal</class>
               <name-description-map>
                    <name-description-pair>
                         <name>*</name>
                         <description>All Principals</description>
                    </name-description-pair>
               </name-description-map>
          </principal-class>
     </jazn-principal-classes>
```

```
<!-- Login Module Data -->
    <jazn-loginconfig>
        <application>
             <name>TestRealmLogin</name>
             <login-modules>
                 <login-module>
                      <class>oracle.security.jazn.realm.RealmLoginModule</class>
                      <control-flag>required</control-flag>
                      <options>
                           <option>
                               <name>addRoles</name>
                               <value>true</value>
                           </option>
                      </options>
                 </login-module>
             </login-modules>
        </application>
    </jazn-loginconfig>

</jazn-data>
```

Supplemental Code Samples

The following code samples are intended as supplemental information. This section presents the following:

- Supplementary Code Sample: Creating an Application Realm
- Supplementary Code Sample: Modifying User Permissions

See Also:

- "Realm Creation" on page 6-26 for further information on creating realms
- "Creating an External Realm" on page 6-26 for further information on creating application realms

Supplementary Code Sample: Creating an Application Realm

The following code sample creates an Application Realm with the objects shown in Table B–1. The objects to be modified are presented in bold.

Table B–1 Objects In Sample Application Realm Creation Code

Objects	Names
sample organization	`dev.com`
`adminUser` (optional)	`John.Singh`
`adminRole`	`administrator`
sample realm name	`devRealm`

Example B–2 Application Realm Creation Code

```
import oracle.security.jazn.spi.ldap.*;
import oracle.security.jazn.*;
import oracle.security.jazn.realm.*;

import java.util.*;

/**
 * Creates an application realm.
 */

public class CreateRealm extends Object
{
    public CreateRealm() {};

    public static void main (String[] args) {
      CreateRealm test = new CreateRealm();
      test.createAppRealm();
    }

    void createAppRealm() {
    Realm realm=null;


 try {
     Hashtable prop = new Hashtable();
     prop.put(Realm.LDAPProperty.USERS_SEARCHBASE,"cn=users,o=dev.com");
```

```
     // specifying the following LDAP directory object class
       // is optional.  When specified, it will
     // be used as a filter to search for users
     prop.put(Realm.LDAPProperty.USERS_OBJ_CLASS,"orclUser");

     // adminUser is optional
    String adminUser = "John.Singh";

     String adminRole = "administrator";

     RealmManager realmMgr = JAZNContext.getRealmManager();

     InitRealmInfo realmInfo = new
          InitRealmInfo(InitRealmInfo.RealmType.APPLICATION_REALM, adminUser,
          adminRole, prop);
     realm = realmMgr.createRealm("devRealm", realmInfo);
     }

catch (Exception e) {
     e.printStackTrace();
    }
  }
}
```

Supplementary Code Sample: Modifying User Permissions

Example B–3 demonstrates granting `java.io.FilePermission` to a user named `Jane.Smith`. The objects to be modified are presented in bold.

Table B–2 lists the objects in Example B–3.

Table B–2 Objects In Sample Modifying User Permissions Code

Objects	Names	Comments...
`RealmUser user`	`Jane.Smith`	
`codesource cs`	`file:/home/task.jar`	
`File path`	`report.data`	Path is the pathname of the file.
sample organization	`abc.com`	`abc.com` does not appear in this code directly, but was acted upon in the creation of this sample External Realm in Example 6–1 on page 6-27.
sample External Realm	`abcRealm`	`abcRealm` appears in this code and in the creation of this sample External Realm in External Realm Creation Code on page 6-27.

Example B–3 Modifying User Permissions Code

Code Sample

```
import oracle.security.jazn.*;
import oracle.security.jazn.policy.*;
import oracle.security.jazn.realm.*;
import java.lang.*;
import java.security.*;
import java.util.*;
import java.net.*;
import java.io.*;

public class Init {

    public static void main(String[] args) {

        try {
            RealmManager realmMgr = JAZNContext.getRealmManager();
            Realm realm = realmMgr.getRealm("abcRealm");
            UserManager userMgr = realm.getUserManager();
            RoleManager roleMgr = realm.getRoleManager();
            final JAZNPolicy policy = JAZNContext.getPolicy();
```

```
                final RealmUser user = userMgr.getUser("Jane.Smith");

                AccessController.doPrivileged (new PrivilegedAction() {
                        public Object run() {

                    try {

                      CodeSource cs = new CodeSource(new URL("
                                file:/home/task.jar"), null);
                        HashSet prop = new HashSet();
                        prop.add((Principal) user);

                        // assign permission to principals
                        policy.grant(new Grantee(prop, cs), new
                                 FilePermission("report.data", "read"));

                        return null;
                            } catch (JAZNException e1) {
                                e1.printStackTrace();
                            } catch (java.net.MalformedURLException e2) {
                                e2.printStackTrace();
                            }
                        return null;
                        }
                    }
                 );

            } catch (JAZNException e) {
                e.printStackTrace();
            }
        }
    }
```

Discussion Of Sample Code

The sample code shown in Example B–3 is preparation for using the sample application, `AccessTest1`, discussed in "Sample J2SE Application" on page 7-5. This sample code grants a user, `Jane.Smith`, permission to use `AccessTest1` as follows:

The name `cs` is assigned to the `file:/home/task.jar`, which includes the sample application `AccessTest1`:

```
CodeSource cs = new CodeSource(new URL("
                            file:/home/task.jar"), null);
```

`Jane.Smith` is the user added to the hashset prop:

```
HashSet prop = new HashSet();
                    prop.add((Principal) user);
```

`Jane.Smith` is granted permission, on the `Codesource cs`, to read the file `report.data`.

```
policy.grant(new Grantee(prop, cs), new
                             FilePermission("report.data", "read"));
```

Index

Symbols

A

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X