Oracle® Containers for J2EE

Enterprise JavaBeans Developer's Guide

10*g* (10.1.3.5.0)

E13981-01

July 2009

ORACLE®

Oracle Containers for J2EE Enterprise JavaBeans Developer's Guide, 10*g* (10.1.3.5.0)

E13981-01

Copyright © 2002, 2009, Oracle and/or its affiliates. All rights reserved.

Primary Author: Joseph Ruzzi

Contributing Author: Peter Purich, Debu Panda, Raghu Kodali

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this software or related documentation is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of this software. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software in dangerous applications.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

This software and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

2 Understanding EJB Application Development

3 Understanding EJB Support in OC4J

Part II EJB 3.0 Session Beans

4 Implementing an EJB 3.0 Session Bean

5 Using an EJB 3.0 Session Bean

Part III JPA Entities

6 Implementing a JPA Entity

7 Using Java Persistence API

8 Implementing JPA Queries

Part IV EJB 3.0 Message-Driven Beans

9 Implementing an EJB 3.0 Message-Driven Bean

10 Using an EJB 3.0 Message-Driven Bean

Part V EJB 2.1 Session Beans

11 Implementing an EJB 2.1 Session Bean

12 Using an EJB 2.1 Session Bean

Part VI EJB 2.1 Entity Beans

13 Implementing an EJB 2.1 Entity Bean

14 Using an EJB 2.1 Entity Bean With Container-Managed Persistence

15 Using an EJB 2.1 Entity Bean With Bean-Managed Persistence

Part VIII Configuring OC4J EJB Services

19 Configuring JNDI Services

20 Configuring Data Sources

21 Configuring Transaction Services

22 Configuring Security Services

23 Configuring Message Services

24 Configuring OC4J EJB Application Clustering Services

25 Configuring Timer Services

Part IX Packaging and Deploying an EJB Application

26 Configuring Deployment Descriptor Files

27 Packaging an EJB Application

28 Deploying an EJB Application to OC4J

Part X Using an EJB in Your Application

29 Accessing an Enterprise Bean From a Client

30 Using EJB and Web Services

31 Administrating an EJB Application

32 Optimizing EJB Performance

A XML Reference for orion-ejb-jar.xml Elements

Preface

This guide gets you started building enterprise Java beans for Oracle Containers for J2EE (OC4J) using:

- Java Enterprise Edition (EE) 5 Enterprise JavaBeans (EJB) 3.0 and the TopLink Java Persistence API (JPA) persistence provider.
- J2EE 1.4 EJB 2.1 and the TopLink EJB 2.1 persistence manager.

It includes code examples to help you develop your application.

The Orion persistence manager is deprecated. Oracle recommends that you use OC4J and the TopLink JPA persistence provider for new development. Using the migration tool (see "Migrating to the TopLink EJB 2.1 Persistence Manager" on page 3-13), you can easily migrate an existing OC4J application that uses EJB 2.0 entity beans with the Orion persistence manager to use EJB 2.0 entity beans with the TopLink persistence manager.

If you have questions about OC4J, see the OC4J user's forum at `http://forums.oracle.com/forums/category.jspa?categoryID=13`.

If you have questions or feedback about this documentation, see the documentation feedback forum at `http://forums.oracle.com/forums/forum.jspa?forumID=165`.

Audience

Anyone developing Enterprise JavaBeans for OC4J will benefit from reading this guide. Written especially for programmers, it will also be of value to architects, systems analysts, project managers, and others interested in EJB applications deployed to OC4J.

This guide assumes that you already have a working knowledge of Java EE and the EJB 3.0 and EJB 2.1 specifications.

Documentation Accessibility

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

accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at http://www.oracle.com/accessibility/.

Accessibility of Code Examples in Documentation

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

Accessibility of Links to External Web Sites in Documentation

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

Deaf/Hard of Hearing Access to Oracle Support Services

To reach Oracle Support Services, use a telecommunications relay service (TRS) to call Oracle Support at 1.800.223.1711. An Oracle Support Services engineer will handle technical issues and provide customer support according to the Oracle service request process. Information about TRS is available at http://www.fcc.gov/cgb/consumerfacts/trs.html, and a list of phone numbers is available at http://www.fcc.gov/cgb/dro/trsphonebk.html.

Related Documents

For more information, see the following documents in the OC4J documentation set:

- *Oracle Application Server Release Notes for Microsoft Windows*
- *Oracle Containers for J2EE Configuration and Administration Guide*
- *Oracle Containers for J2EE Resource Adapter Administrator's Guide*
- *Oracle Containers for J2EE Developer's Guide*
- *Oracle Containers for J2EE Services Guide*
- *Oracle Containers for J2EE Security Guide*
- *Oracle Containers for J2EE Deployment Guide*
- *Oracle Containers for J2EE Job Scheduler Developer's Guide*
- *Oracle Containers for J2EE Servlet Developer's Guide*
- *Oracle Application Server Annotations API Reference*
- *Oracle TopLink Developer's Guide*
- *Oracle TopLink API Reference*
- EJB specifications: http://java.sun.com/products/ejb/docs.html.
- EJB API documentation: http://www.javasoft.com.
- EJB tutorials: http://java.sun.com/developer/onlineTraining/.
- EJB design patterns: http://java.sun.com/blueprints/patterns/.

Conventions

The following text conventions are used in this document:

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

Part I

EJB Overview

This part provides conceptual information to help you understand EJB architecture, EJB application development, and OC4J EJB support.

This part contains the following chapters:

1

Understanding Enterprise JavaBeans

Java Enterprise Edition (Java EE) Enterprise JavaBeans (EJB) are a component architecture that you use to develop and deploy object-oriented, distributed, enterprise-scale applications. An application written according to the EJB architecture is scalable, transactional, and secure. The component types that you can create are commonly referred to as Enterprise JavaBeans.

This chapter describes the following:

What are Enterprise JavaBeans?

The EJB architecture is flexible enough to implement the objects that Table 1–1 lists.

Table 1–1 EJB Types

Type	Description	See ...
Session	An EJB 3.0 or EJB 2.1 component created by a client for the duration of a single client/server session used to perform operations for the client.	"What is a Session Bean?" on page 1-27
Stateless	A session bean that does not maintain conversational state. Used for reusable business services that are not connected to any specific client.	"What is a Stateless Session Bean?" on page 1-28
Stateful	A session bean that does maintain conversational state. Used for conversational sessions with a single client (for the duration of its lifetime) that maintain state, such as instance variable values or transactional state.	"What is a Stateful Session Bean?" on page 1-30
Entity	An EJB 3.0-compliant light-weight entity object that represents persistent data stored in a relational database using the Java Persistence API (JPA) persistence provider specified in its persistence unit (see "What is the persistence.xml File?" on page 2-8).	"What is a JPA Entity?" on page 1-34
Entity Bean	An EJB 2.1 enterprise bean component that represents persistent data stored in a relational database.	"What is an EJB 2.1 Entity Bean?" on page 1-41

Table 1–1 (Cont.) EJB Types

Type	Description	See ...
CMP	An entity bean with container-managed persistence (CMP) is an entity bean that delegates persistence management to the persistence manager used by the container that hosts it.	"What is an EJB 2.1 Entity Bean With Container-Managed Persistence?" on page 1-42
BMP	An entity bean with bean-managed persistence (BMP) is an entity bean that manages its own persistence.	"What is an EJB 2.1 Entity Bean With Bean-Managed Persistence?" on page 1-46
MDB	A message-driven bean (MDB) is an EJB 3.0 or EJB 2.1 component that functions as an asynchronous consumer of Java Message Service (JMS) messages.	"What is a Message-Driven Bean?" on page 1-56

For more information, see: the following

- What is the Anatomy of an EJB 3.0 enterprise bean?
- What is the Anatomy of an EJB 2.1 Enterprise Bean?
- What is the Life Cycle of an Enterprise Bean?
- What is EJB Context?
- How do Annotations and Resource Injection Work?
- Which Type of Enterprise Bean Should You Use?

What is the Anatomy of an EJB 3.0 enterprise bean?

Using EJB 3.0, the interfaces for your EJB implementation are not restricted by EJB type. For example, in your JPA entity implementation you may implement an EJB using a plain old Java object (POJO) and any plain old Java interfaces (POJI): you do not need to implement interfaces such as `javax.ejb.EntityBean` and you do not need to provide separate interfaces that extend `EJBHome`, `EJBLocalHome`, `EJBObject`, or `EJBLocalObject`. A client may instantiate an EJB 3.0 POJO entity instance with `new` (or the `EntityManager`: see "How do you Query for a JPA Entity?" on page 1-39). A client may instantiate an EJB 3.0 session bean using dependency injection or JNDI lookup. For more information, see "EJB 3.0 Support" on page 3-1.

Table 1–2 lists the parts you create when developing an EJB 3.0 enterprise bean.

Table 1–2 Parts of an EJB 3.0 EJB

Part	Type	Description
Home interface	POJI	An optional POJI annotated with `@Home` that specifies an object that the container itself implements: the *home object*. The `@Home` is only provided to help EJB 3.0 beans interoperate with EJB 2.1 clients, if necessary. Most EJB 3.0 bean instances will not need to provide a home interface.

Table 1–2 (Cont.) Parts of an EJB 3.0 EJB

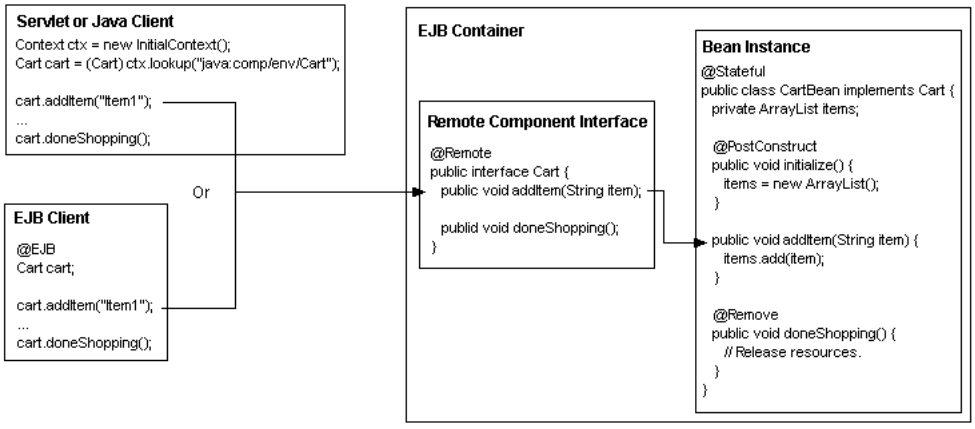
Part	Type	Description
Component interface	POJI	A mandatory POJI annotated with `@Remote` or `@Local` (default) that specifies the business methods that you implement in the bean and that a client can invoke. No other container service methods need be implemented, unless you need to override default container behavior. The bean class does not need to implement this interface.
Bean implementation	POJO	A mandatory POJO that may optionally implement a component interface and contains the Java code that implements the methods defined in the optional home interface and component interface (business methods). If necessary, you can optionally annotate any method to serve as a container life cycle callback function.

Deployment descriptor	`ejb-jar.xml`
	`orion-ejb-jar.xml`
	`toplink-ejb-jar.xml`
	`ejb3-toplink-sessions.xml`
	`persistence.xml`
	`orm.xml`
		Optional means of specifying attributes of the bean for deployment. These designate configuration specifics, such as environment, interface names, transactional support, type of EJB, and persistence information. Because this metadata can be expressed entirely through annotations (or defaults), deployment descriptor XML files are less important in EJB 3.0. Configuration in a deployment descriptor XML file overrides the corresponding annotation configuration, if present. For more information, see "Understanding EJB Deployment Descriptor Files" on page 2-4.

As Figure 1–1 illustrates, to acquire an EJB 3.0 EJB instance, a Web client (such as a servlet) or Java client uses JNDI, while an EJB client may use either JNDI or resource injection. For more information about EJB clients, see "What Type of Client do you Have?" on page 29-1.

For entity beans, EJB 3.0 provides an `EntityManager` that you use to create, find, merge, and persist a JPA entity (see "How do you Query for a JPA Entity?" on page 1-39).

Figure 1–1 A Client Using an EJB 3.0 Stateful Session Bean by Component Interface



The client in Figure 1–1 accesses the EJB as follows:

1. The client retrieves the component interface of the bean.

   The servlet or Java client uses JNDI to look up an instance of `Cart`.

   The EJB client uses resource injection by annotating a `Cart` instance variable with the `@EJB` annotation: at run time, the EJB container will ensure that the variable is initialized accordingly.

   In both cases, the EJB container manages instantiation. A home interface is not necessary.

2. The client invokes a method defined in the component interface (remote or local interface), which delegates the method call to the corresponding method in the bean instance (through a stub).
3. The client can destroy the stateful session bean instance by invoking a method in its component interface that is annotated in the bean instance with `@Remove`.

   Stateless session beans do not require a `remove` method; the container removes the bean if necessary. The container can also remove stateful session beans that exceed their configured timeout or to maintain the maximum configured pool size. Entities do not require a remove method; you use the EJB 3.0 `EntityManager` to create and destroy entities.

What is the Anatomy of an EJB 2.1 Enterprise Bean?

Using EJB 2.1, the interfaces for your EJB implementation are based on EJB type. For example, in your EJB 2.1 entity bean implementation, you must implement the `javax.ejb.EntityBean` interface and you must provide separate interfaces that extend `EJBHome` or `EJBLocalHome` and `EJBObject` or `EJBLocalObject`. A client may instantiate an EJB 2.1 enterprise bean instance only with a `create` method that your EJB home interface provides. For more information, see "EJB 2.1 Support" on page 3-11.

Table 1–3 lists the parts you create when developing an EJB 2.1 enterprise bean.

Table 1–3 Parts of an EJB 2.1 EJB

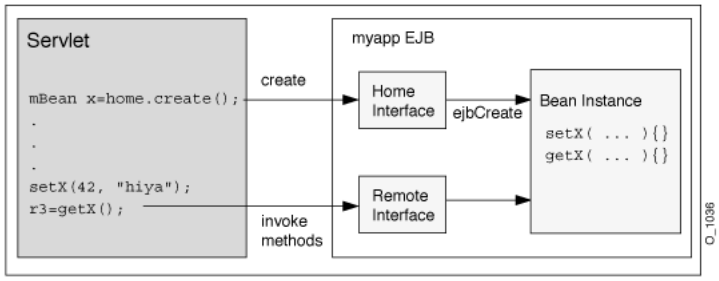
Part	Type	Description
Home interface	`javax.ejb.EJBHome` (remote) `javax.ejb.EJBLocalHome`	Specifies the interface to an object that the container itself implements: the *home object*. The home interface contains the life cycle methods, such as the `create` methods that specify how a bean is created.
Component interface	`javax.ejb.EJBObject` (remote) `javax.ejb.EJBLocalObject`	Specifies the business methods that you implement in the bean. The bean must also implement additional container service methods. The EJB container invokes these methods at different times in the life cycle of a bean.

Bean implementation	`javax.ejb.SessionBean`
	`javax.ejb.EntityBean`
	`javax.ejb.MessageDrivenBean`
		Contains the Java code that implements the methods defined in the home interface (life cycle methods), component interface (business methods), and the required container methods (container callback functions).

Deployment descriptor	`ejb-jar.xml`
	`toplink-ejb-jar.xml`
	`orion-ejb-jar.xml`
		Specifies attributes of the bean for deployment. These designate configuration specifics, such as environment, interface names, transactional support, type of EJB, and persistence information.

A client uses the home interface to acquire an EJB 2.1 enterprise bean instance and uses the component interface to invoke its business methods, as Figure 1–2 illustrates. For more information about EJB clients, see "What Type of Client do you Have?" on page 29-1.

Figure 1–2 A Client Using an EJB 2.1 Stateless Session Bean by Home and Component Interface



The client in Figure 1–2 accesses the EJB as follows:

1. The client retrieves the home interface of the bean–typically, through JNDI.
2. The client invokes the `create` method on the home interface reference (home object). This creates the bean instance and returns a reference to the component interface (remote or local interface) of the bean.
3. The client invokes a method defined in the component interface (remote or local interface), which delegates the method call to the corresponding method in the bean instance (through a stub).
4. The client can destroy the bean instance by invoking the `remove` method that is defined in the component interface (remote or local interface).

   For some beans, such as stateless session beans, calling the `remove` method does nothing: in this case, the container is responsible for removing the bean instance.

What is the Life Cycle of an Enterprise Bean?

The life cycle of an enterprise bean involves important events such as creation, passivation, activation, and removal.

Each such event is associated with a callback method. You can define life cycle callback methods on the following:

- the enterprise bean class itself for any bean type (see "Life Cycle Callback Methods on a Bean Class" on page 1-6)
- an interceptor class of the enterprise bean for EJB 3.0 session and message-driven beans (see "Life Cycle Callback Interceptor Methods on an EJB 3.0 Interceptor Class" on page 1-6)
- an entity listener class of a JPA entity (see "Life Cycle Callback Listener Methods on a JPA Entity Listener Class" on page 1-6)

You can combine these options: for example, you can define some life cycle callbacks as methods of a session bean class, and some in an interceptor class that you associate with the session bean.

The container invokes the callback prior to, or immediately after the life cycle event (depending on the event type).

The life cycle events associated with an enterprise bean and whether or not the container or the bean provider is responsible for implementing callbacks is determined by the type of enterprise beans you are developing (as specified in the appropriate EJB interface).

For an EJB 3.0 enterprise bean, when the container is responsible for the life cycle callback, you do not need to provide an implementation in your bean, unless you want to perform some additional logic.

For an EJB 2.1 enterprise bean, even when the container is responsible for the life cycle callback, and even if you do not want to perform additional logic, you must at least provide an empty implementation of the life cycle methods to satisfy the requirements of the applicable EJB interface.

For more information, see the following:

- "What is the Stateless Session Bean Life Cycle?" on page 1-28
- "What is the Life Cycle of a Stateful Session Bean?" on page 1-30
- "What is the JPA Entity Life Cycle?" on page 1-37
- "What is the Life Cycle of an EJB 2.1 Entity Bean With Container-Managed Persistence?" on page 1-43
- "What is the Life Cycle of an EJB 2.1 Entity Bean With Bean-Managed Persistence?" on page 1-46
- "What is the Life Cycle of a Message-Driven Bean?" on page 1-57

Life Cycle Callback Methods on a Bean Class

For any EJB 3.0 enterprise bean type, you can optionally annotate any EJB class method as a life cycle method.

For an EJB 2.1 enterprise bean, you must at least provide an empty implementation of the life cycle methods to satisfy the requirements of the applicable EJB interface.

Life Cycle Callback Interceptor Methods on an EJB 3.0 Interceptor Class

For an EJB 3.0 session bean or message-driven bean, you can optionally associate the bean class with an interceptor class and annotate any interceptor class method as a life cycle method.

For more information, see the following:

- "Understanding EJB 3.0 Interceptors" on page 2-10
- "Configuring a Life Cycle Callback Interceptor Method on an Interceptor Class of an EJB 3.0 Session Bean" on page 5-5
- "Configuring a Life Cycle Callback Interceptor Method on an Interceptor Class of an EJB 3.0 MDB" on page 10-11

Life Cycle Callback Listener Methods on a JPA Entity Listener Class

For a JPA entity, you can associate the bean class with an entity listener class and annotate any entity listener class method as a life cycle method.

For more information, see "Configuring a Life Cycle Callback Listener Method on an Entity Listener Class of a JPA Entity" on page 7-17.

What is EJB Context?

The `EJBContext` interface provides an instance with access to the container-provided run-time context of an EJB 2.1 enterprise bean instance. This interface is extended by the `SessionContext`, `EntityContext`, and `MessageDrivenContext` interfaces to provide additional methods specific to the enterprise interface Bean type.

The `javax.ejb.EJBContext` interface has the following definition:

```
public interface EJBContext {
    public EJBHome         getEJBHome();
    public Properties      getEnvironment();
    public Principal       getCallerPrincipal();
    public boolean         isCallerInRole(String roleName);
    public UserTransaction getUserTransaction();
    public boolean         getRollbackOnly();
    public void            setRollbackOnly();
}
```

A bean needs the EJB context when it wants to perform the operations listed in Table 1–4.

Table 1–4 EJB 2.1 EJBContext Operations

Method	Description
`getEnvironment`	Get the values of properties for the bean.
`getUserTransaction`	Get a transaction context, which enables programmatic transaction demarcation when using bean-managed transactions (BMT). This is valid only for beans that have been designated transactional.
`setRollbackOnly`	Set the current transaction so that it cannot be committed. Applicable only to container-managed transactions.
`getRollbackOnly`	Check whether the current transaction is marked for rollback only. Applicable only to container-managed transactions.
`getEJBHome`	Retrieve the object reference to the corresponding `EJBHome` (home interface) of the bean.
`lookup`	Use JNDI to retrieve the bean by environment reference name. When using this method, you do *not* prefix the bean reference with "`java:comp/env`".

Do not confuse `EJBContext` with `IntialContext` (see "Configuring the Initial Context Factory" on page 19-19).

For more information, see the following:

- "What is Session Context?" on page 1-34
- "What is Entity Context?" on page 1-48
- "What is Message Driven Context?" on page 1-58
- "Accessing an EJB 2.1 EJBContext" on page 29-27

How do Annotations and Resource Injection Work?

Annotations allow you to control the behavior and deployment of your application. You can use metadata annotations to specify expected requirements on container behavior, to request the injection of services and resources, and to specify object-relational mappings.

Using annotations, an EJB 3.0 enterprise bean may use dependency injection mechanisms to acquire references to resources or other objects in its environment. For example, you can use the following:

- `@Resource`: to inject non-EJB resources such as a database connection.
- `@EJB`: to inject an enterprise bean such as a session bean.
- `@PersistenceContext`: to inject an `EntityManager` instance to create, read, update, and delete EJB 3.0 entities.

If an EJB 3.0 enterprise bean makes use of dependency injection, OC4J injects these references after the bean instance is created, and before any business methods are invoked.

If a dependency on the EJB context is declared, the EJB context is also injected (see "What is EJB Context?" on page 1-6).

If dependency injection fails, OC4J discards the bean instance.

OC4J supports annotation inheritance (see "Annotations and Inheritance" on page 1-9).

In this release, you can use annotations and resource injection in the Web tier (see "Annotations in the Web Tier" on page 1-9).

Annotations are another way of specifying an environment reference without having to use XML. When you annotate a field or property, the container injects the value into the bean on your behalf by looking it up from JNDI. When a reference is specified using annotations, you can still look it up using JNDI. Example 1–1 shows how annotations relate to JNDI. The annotations in this example correspond to the `ejb-jar.xml` file equivalent in Example 1–2. Your code would have the exact same behavior if this XML and JNDI was used instead.

You can override annotation configuration using deployment XML (see "Overriding Annotations With Deployment Descriptor Entries" on page 1-20).

Example 1–1 Using Annotations and Resource Injection

```
@Stateless
@EJB(name="bean1", businessInterface=Bean1.class)
public class MyBean {
    @EJB Bean2 bean2;

    public void doSomething() {
        // Bean2 is already injected and available
        bean2.foo();
        // or it can be looked up from JNDI
        ((Bean2)(new InitialContext().lookup("java:comp/env/bean2"))).foo();
        // Bean1 has not been injected and is only available through JNDI
        ((Bean1)(new InitialContext().lookup("java:comp/env/bean1"))).foo();
    }
}
```

Example 1–2 Equivalent ejb-jar.xml File Configuration

```
<ejb-local-ref>
    <ejb-ref-name>bean1</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local>Bean1.class</local>
</ejb-local-ref>

<ejb-local-ref>
    <ejb-ref-name>bean2</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local>Bean2.class</local>
    <injection-target>
        <injection-target-name>bean2</injection-target-name>
    </injection-target>
</ejb-local-ref>
```

Annotations in the Web Tier

In this release, OC4J supports annotations and resource injection in the Web tier. To use annotations and resource injection in the Web tier, your client must use Java SE 1.5 and Servlet 2.5 or later.

You can use the following annotations in the Web tier:

- `@EJB`
- `@Resource` and `@Resources`
- `@PersistenceUnit` and `@PersistenceUnits`
- `@PersistenceContext` and `@PersistenceContexts`
- `@WebServiceRef`
- `@PostConstruct`
- `@PreDestroy`
- `@DeclaresRoles`
- `@RunAs`

For more information, see the following:

- *Oracle Containers for J2EE Servlet Developer's Guide*
- "Acquiring an Entity Manager in a Web Client" on page 29-10
- "Sending a Message to a JMS Destination Using EJB 3.0" on page 29-17

Annotations and Inheritance

Annotations participate in inheritance. To ensure that annotations are local to their host class, consider the following:

- Class-level annotations only affect the class they annotate and its members (methods and fields). Annotations never affect a member declared by a superclass, even if the member is not hidden or overridden by the subject subclass.
- Explicit member-level annotations have priority over member-level annotations implied by a class-level annotation, except for the cases when the annotation is potentially additive (for example, interceptor annotations): if a member carries a specific member-level annotation, any annotations of the same type implied by a class-level annotation are ignored.
- Interfaces implemented by a class never contribute annotations to the class itself or to any of its members.
- Members inherited from a superclass (the ones that are not hidden or overridden) maintain the annotations they had in the class that declared them, including member-level annotations implied by class-level annotations.
- Member-level annotations on a hidden or overridden member are always ignored.

To find the annotation in-effect for a class member, you need to track down the last nonhidden and nonoverridden declaration of the class member and examine it. If you cannot find the annotation, then you have to examine the enclosing class declaration. If this fails, do not consult any other source files.

Table 1–5 lists annotations and specifies how each of the annotations behave with respect to inheritance in the bean class.

Table 1–5 Annotations and Inheritance

Annotations	Reaction to Inheritance	Comment	OC4J Support

`@Stateless`
`@Stateful`
`@MessageDriven`
	Superclass annotations are ignored.
	Example:
	`@Stateful`
	`class Base {}`
	
	`@Stateless`
	`class A extends Base {}`
	
	`class B extends Base {}`
	
	where:
	- bean `Base` is a stateful session bean;
	- bean `A` is a stateless session bean: `@Stateful` annotation of its parent bean `Base` does not apply (is ignored);
	- bean `B` is a POJO class: `@Stateful` annotation of its parent bean `Base` does not apply (is ignored).
		You must explicitly define a bean class through either a class annotation or deployment descriptor XML file, even if the bean is a subclass of another bean class.	Yes.
			OC4J ignores the superclass-level bean-type annotations.

`@Local`
`@Remote`
`@LocalHome`
`@Home`
	Superclass annotations are ignored.
	You need to define the annotations properly to avoid run-time issues.
	Example:
	`@Local`
	`interface Base {}`
	
	`@Remote`
	`interface A extends Base {}`
	
	`interface B extends Base {}`
	
	where:
	- `Base` is a local business interface;
	- `A` is a remote interface: `@Local` annotation of its parent bean `Base` does not apply (is ignored);
	- `B` is a POJO interface: `@Local` annotation of its parent bean `Base` does not apply (is ignored).
		This also implies the annotation on the bean.
		Example:
		`@Stateful`
		`@Local(I1.class)`
		`class A {}`
		
		`@Stateful`
		`class B extends A {}`
		
		Note: unlike `A`, bean `B` does not have `I1` business interface.
			Yes.
			OC4J ignores the superclass-level bean-type annotations.

`@TransactionManagement(TransactionManagementType.CONTAINER)`
@TransactionManagement(TransactionManagementType.APPLICATION)
	Superclass annotations are ignored.
	Example:
	`@ TransactionManagement (type=TransactionManagementType.CONTAINER)`
	`class Base {}`
	
	`@ TransactionManagement (type=TransactionManagementType.APPLICATION)`
	`class A extends Base {}`
	
	`class B extends Base {}`
	
	where:
	- `A` is a bean that uses bean-managed transactions;
	- `B` is a bean that uses **default** container-managed transactions.
		No class-level transaction management inheritance means that a bean that uses bean-managed transactions and container-managed transactions will be mixed in the application. This might cause run-time issues.
		If not explicitly annotated, a bean by default uses container-managed transactions.
			Yes.
			OC4J ignores the superclass-level bean-type annotation.

Table 1–5 (Cont.) Annotations and Inheritance

Annotations	Reaction to Inheritance	Comment	OC4J Support

`@TransactionAttribute(TransactionAttributeType.REQUIRED) {MANDATORY, REQUIRED, REQUIRES_NEW, SUPPORTS, NOT_SUPPORTED, NEVER}`	Method-level inheritance and "virtual method annotation" inheritance are allowed.
	Example:
	`@Transaction(REQUIRED)`
	`class Base {`
	`  @Transaction(NONE)`
	`  public void foo() {...}`
	`  public void bar() {...}`
	`}`
	
	`class A extends Base {`
	`  public void foo() {...}`
	`}`
	
	`public class B extends Base {`
	`  @Transaction(NEW)`
	`  public void foo() {...}`
	`}`
	
	`@Transaction(NEW)`
	`public class C extends Base {`
	`  public void foo() {...}`
	`  public void bar() {...}`
	`}`
	
	`@Transaction(NEW)`
	`public class D extends Base {`
	`  public void bar() {...}`
	`}`
	
	`@Transaction(NEW)`
	`public class E extends Base {`
	
	where:
	- in bean `A`, the `foo` method is not annotated: bean `A` overrides the `foo` method of its parent bean `Base` without annotating this method. Therefor, the `foo` method in bean `A` does not carry `@Transaction(NONE)` annotation;
	-in bean `B`, the `@Transaction(NEW)` annotation is applicable to the `foo` method: bean `B` overrides the `foo` method of its parent bean `Base` annotating this method with `@Transaction(NEW)`. As a result, `@Transaction(NONE)` annotation from the `foo` method of bean `Base` does not apply to the overridden method in the child bean `B`;
	- in bean `C`, the `@Transaction(NEW)` annotation is applicable to the `foo` method: bean `C` overrides the `foo` method of its parent bean `Base` without annotating this method. Therefor, the `foo` method in bean `C` does not carry `@Transaction(NONE)` annotation. However, bean `C` has a class-level annotation `@Transaction(NEW)`, which is applied to its `foo` method;
	- in bean `D`, the `@Transaction(NEW)` annotation is applicable to the `bar` method: bean `D` overrides the `bar` method of its parent bean `Base` without annotating this method. Therefor, the `bar` method in bean `C` does not carry `@Transaction(NONE)` annotation. However, bean `C` has a class-level annotation `@Transaction(NEW)`, which is applied to its `bar` method;
		Supports the "virtual method annotation", which is annotated at the supercall class level and applied to all methods in the class.
		For more information, see JSR 250 at http://jcp.org/en/jsr/detail?id=250
			Yes.

Table 1–5 (Cont.) Annotations and Inheritance

Annotations	Reaction to Inheritance	Comment	OC4J Support
`@TransactionAttribute(TransactionAttributeType.REQUIRED){ MANDATORY, REQUIRED, REQUIRES_NEW, SUPPORTS, NOT_SUPPORTED, NEVER}` (continues from preceding row)	(continues from preceding row) - in bean `E`, the `@Transaction(REQUIRED)` annotation is applicable to the `bar` method: bean `E` has a class-level annotation `@Transaction(NEW)`, but does not override the `bar` method of its parent bean `Base`. Therefor, the `bar` method in bean `E` carries the class-level `@Transaction(REQUIRED)` annotation from its parent bean `Base`.	(continues from preceding row)	(continues from preceding row)

`@EJB`
`@EJBs`
	All superclasses will be examined to discover all uses of this annotation, including private methods and fields, and private overridden methods (both private in a parent and child).
	Example:
	`@EJB(beanName = "Bean1"…)`
	`public class Base {`
	`  @EJB(beanName =" Bean2"..)`
	`  private Bean2 b2;`
	`  @EJB(beanName =" Bean3"..)`
	`  protected void setB3(Bean3 b3){}`
	`}`
	
	`@EJB(beanName = "Bean4"…)`
	`public class A extends Base {`
	`  @EJB(beanName =" Bean5"..)`
	`  private Bean5 b5;`
	`}`
	
	`public class B extends Base {}`
	
	When parsing bean `A`, all `@EJB` references defined in superclasses, including `Bean1`, `Bean2`, `Bean3`, `Bean4` and Bean5 will be parsed and added. The annotated fields and methods will also be injected.
	When parsing bean `B`, all `@EJB` references defined in superclasses, including `Bean1`, `Bean2`, `Bean3`, `Bean4` will be parsed and added. The annotated fields and methods will also be injected.
			Yes.

`@PersistenceUnit`
`@PersistenceUnits`
`@PersistenceContext`
`@PersistenceContexts`
	Similar to `@EJB` and `@EJBs` (see preceding row).		Yes.

`@Resources` `@Resource`	Similar to `@EJB` and `@EJBs` (see preceding row).		Yes.

Table 1–5 (Cont.) Annotations and Inheritance

Annotations	Reaction to Inheritance	Comment	OC4J Support

`@Interceptors`
`@ExcludeDefaultInterceptor`
`@ExcludeClassInterceptor`
	Inheritance is allowed.
	Method-level business method interceptors are invoked in addition to any default interceptors and interceptors defined for the bean class (and its superclasses).
	Example:
	Default interceptor: `D1.class`
	`@Interceptors({C1.class})`
	`class Base {`
	`  @Interceptors({M1.class})`
	`  public void foo() {...}`
	`  public void bar() {...}`
	`}`
	
	`@Interceptors({C2.class})`
	`class A extends Base {`
	`  public void foo() {...}`
	`}`
	
	`@Interceptors({C2.class})`
	`class B extends Base {`
	`  @Interceptors({M2.class})`
	`  public void foo() {...}`
	`}`
	
	`@Interceptors({C2.class})`
	`class C extends Base {`
	`  public void bar() {...}`
	`}`
	
	`@Interceptors({C3.class, C4.class})`
	`class E extends Base {}`
	
	`@Interceptors({C3.class, C4.class})`
	`class F extends Base {`
	`  @ExcludedDefaultInterceptor`
	`  @ExcludedClassInterceptor`
	`  @Interceptors({M2.class})`
	`  public void bar() {...}`
	`}`
	
	where:
	- interceptors for the `foo` method in bean `Base` are `D1, C1, M1`: `D1` is the default interceptor; `C1` is defined as an interceptor on a bean class level; `M1` defined as an interceptor for `foo` on a method level;
	- interceptors for the `foo` method in bean `A` are `D1, C2`: `D1` is the default interceptor; `C1` is defined as an interceptor on a bean class level. Bean `A` overrides the `foo` method and does not define a method-level interceptor for it;
			Yes

Table 1–5 (Cont.) Annotations and Inheritance

Annotations	Reaction to Inheritance	Comment	OC4J Support

`@Interceptors`
`@ExcludeDefaultInterceptor`
`@ExcludeClassInterceptor`
(continues from preceding row)
	(continues from preceding row)
	- interceptors for the `foo` method in bean `B` are `D1, C2, M2`: `D1` is the default interceptor; `C2` is defined as an interceptor on a bean class level; bean `B` overrides the `foo` method and defines `M2` as an interceptor on a method level;
	- interceptors for the `bar` method in bean `C` are `D1, C2`: `D1` is the default interceptor; `C2` is defined as an interceptor on a bean class level;
	- interceptors for the `bar` method in bean `E` are `D1, C1`: `D1` is the default interceptor; bean `E` has a class-level annotation `@Interceptors({C3.class, C4.class})`, but does not override the `bar` method of its parent bean `Base`. Therefor, the `bar` method in bean `E` carries the class-level `@Interceptors({C1.class})` annotation from its parent bean `Base`.
	- interceptor for the `bar` method in bean `F` is `M2`: the default interceptor `D1` is not applicable to this method, because `bar` is annotated with @ExcludeDefaultInterceptor; interceptors defined at the class level are not applicable, because `bar` is annotated with `@ExcludeClassInterceptor`. Bean `F` overrides the `bar` method and provides it with a `@Interceptors({M2.class})` annotation, and only this annotation applies.
		(continues from preceding row)	(continues from preceding row)

`@AroundInvoke`	If a bean class has superclasses, any methods annotated with `@AroundInvoke` and defined on those superclasses will be invoked, with the most general superclass first.
	Example:
	`class Base { @AroundInvoke public Object foo(InvocationContext cts) {...} }`
	`class A extends Base { @AroundInvoke public Object bar(InvocationContext cts) {...} }`
	`class B extends Base { public Object foo(InvocationContext cts) {...} }`
	where:
	- in bean `Base` an interceptor method is `foo();`
	- in bean `A` there are two interceptor methods–`foo` and `bar`: the `bar` method is defined in bean `A`, and the `foo` method is inherited by bean `A` from its parent bean `Base`. `foo` will be invoked first, and `bar` will be invoked second;
	- there is no interceptor method in bean `B`: bean `B` overrides the `foo` method without annotating it with `@AroundInvoke`, therefor making it a non-interceptor method.
		If an interceptor class has superclasses, the interceptor methods defined by the interceptor class' superclasses will be invoked before the interceptor method defined by the interceptor class, with the most general superclass first.	Yes.

Table 1–5 (Cont.) Annotations and Inheritance

Annotations	Reaction to Inheritance	Comment	OC4J Support

`@PostConstruct`
`@PreDestroy`
`@PostActivate`
`@PrePessivate`
	If a bean class has superclasses, any life cycle callback (interceptor) methods defined on the superclasses will be invoked, with the most general superclass first.
	Note: overridden life cycle methods will not be invoked.
	Example:
	`class Base {`
	`  @PostConstruct`
	`  @PostActivate`
	`  void foo() {...}`
	`}`
	
	`class A extends Base {`
	`  @PostConstruct`
	`  void bar() {...}`
	`  @PostActivate`
	`  void ping() {...}`
	`}`
	
	`class B extends Base {`
	`  @PreDestroy`
	`  void foo() {...}`
	`}`
	
	`class C extends Base {`
	`  ejbCreate() {...}`
	`}`
	
	`class D extends Base {`
	`  @PostConstruct`
	`  ping() {...}`
	`  ejbCreate() {...}`
	`}`
	
	where:
	- in bean `Base`, there are two life cycle methods: one post-construct method `foo`, and one post-activate life cycle method `foo`;
	- in bean `A`, there are two post-construct methods: `foo` and `bar`. The `foo` method will be invoked first, and `bar` will be invoked second. Also, bean `A` has two post-activate life cycle methods: `foo` and `ping`. The `foo` method will be invoked first, and `ping` will be invoked second;
	- in bean `B`, the `foo` method is overridden with `@PreDestroy` annotation. Therefor, the post-construct method is not defined in bean `B`, and the post-activate life cycle method is not defined. Only the pre-destroy life cycle method `foo` is defined in bean `B`;
	- in bean `C`, there are two post-construct methods: `foo`, which is inherited from the parent beans `Base` and which is invoked first, and `ejbCreate`, which is defined by bean `C` and is invoked second;
	- there is an error in bean `D`: the EJB 2.1-style life cycle callback (for example, `ejbCreate()` method) cannot coexist with the corresponding EJB 3.0-style (for example, `@PostConstruct` annotation) callback in one bean class.
		If an interceptor class has superclasses, the life cycle callback interceptor methods defined by the interceptor class' superclasses will be invoked before the life cycle callback interceptor method defined by the interceptor class, with the most general superclass first.	Yes.

Table 1–5 (Cont.) Annotations and Inheritance

Annotations	Reaction to Inheritance	Comment	OC4J Support

`@Timeout`	At most one timeout method is allowed in the inheritance hierarchy.
	Example:
	`class Base {`
	`  @Timeout`
	`  public void foo(Timer) {...}`
	`}`
	
	`class A extends Base {`
	`  @Timeout`
	`  public void bar(Timer) {...}`
	`}`
	
	`class B extends Base {`
	`  public void foo(Timer) {...}`
	`}`
	
	`class C extends Base implements TimedObject`
	`{`
	`  public void ejbTimeout(Timer) {...}`
	`}`
	
	where:
	- `foo` is the timeout method in bean `Base`;
	- there is an error in bean `A`: `bar` is the timeout method defined in bean `A`. Bean `A` also inherits `foo` timeout method from its parent bean `Base`. That makes it two timeout methods in bean `A`, which is not allowed;
	- there is no timeout method in bean `B`: bean `B` overrides the `foo` method without annotating it, thus making a non-timeout method;
	- there is an error in bean `C`: `ejb Timeout` is the timeout method defined in bean `C`. In addition, bean `C` inherits `foo` timeout method from its parent bean `Base`. That makes it two timeout methods in bean `C`, which is not allowed two timeout methods in bean `C`.
		If a method annotated in both the base and the superclass (different method name), the container will throw an exception as the EJB 3.0 specification only allows one timeout method for each bean.	Yes.

Table 1–5 (Cont.) Annotations and Inheritance

Annotations	Reaction to Inheritance	Comment	OC4J Support

`@Remove`	Multiple removes are allowed.
	Example:
	`class Base {`
	`  @Remove`
	`  void foo() {...}`
	`}`
	
	`class A extends Base {`
	`  @Remove`
	`  void bar() {...}`
	`}`
	
	`class B extends Base {`
	`  void foo() {...}`
	`}`
	
	`class C extends Base {`
	`  @Remove`
	`  void foo(int) {...}`
	`}`
	
	were:
	- `foo` is the removal method in bean `Base`;
	- `foo` and `bar` are the removal methods in bean `A`: the bar method is explicitly defined as a removal method in bean `A`; since bean `A` does not override the `foo` method, it inherits `foo` as a removal method from its parent bean `Base`;
	- there is no removal method in bean `B`: bean B overrides the `foo` method and does not supply it with an annotation (the `@Remove` annotation of the `foo` method in bean `Base` would have been inherited, if the method was not overridden);
	- `foo()` and `foo(int)` are the removal methods in bean `C`: the foo(int) method is explicitly defined as a removal method in bean `C`; since bean `C` does not override the `foo()` method, it inherits `foo()` as a removal method from its parent bean `Base`.
		The `@Remove` annotation is additive in nature: more than one removal method is allowed in a bean.	Yes.

Table 1–5 (Cont.) Annotations and Inheritance

Annotations	Reaction to Inheritance	Comment	OC4J Support

`@RolesAllowed`
`@DenyAll`
`@PermitAll`
	Only method-level inheritance is allowed.
	Example:
	`@PermitAll`
	`class Base {`
	`  @DenyAll`
	`  public void foo() {...}`
	`  void bar() {...}`
	`}`
	
	`class A extends Base {`
	`  public void foo() {...}`
	`}`
	
	`public class B extends Base {`
	`  @RolesAllowed({admin})`
	`  public void foo( ) {...}`
	`}`
	
	`@RolesAllowed({guest}, {admin})`
	`public class C extends Base {`
	`  public void foo() {...}`
	`  void bar(){...}`
	`}`
	
	`@DenyAll`
	`public class D extends Base {`
	`  public void bar() {...}`
	`}`
	
	`@RolesAllowed({guest}, {admin})`
	`public class E extends Base {}`
	
	`@RolesAllowed({guest}, {admin})`
	`class F extends Base {`
	`  @RolesAllowed ({admin})`
	`  public void bar() {...}`
	`}`
	
	where:
	- in bean `A`, no security permissions are given (no roles are allowed) in the `foo` method: bean `A` cannot inherit any class-level annotations from its parent class `Base`. Moreover, since bean `A` overrides its parent's `foo` method and does not supply this overridden method with annotations, the `foo` method in bean `A` acts as unannotated method;
	- in bean `B`, only admin role is allowed in the `foo` method: bean `B` does not have its own class-level annotations and cannot inherit any class-level annotations from its parent class `Base`. However, since bean `B` overrides its parent's `foo` method and supplies this overridden method with a `@RolesAllowed({admin})` annotation, the `foo` method in bean `B` sets the admin role;
		This is similar to the transaction attribute scenarios.
		Note: EJB 3.0 specification states that method-level security annotation will override the class-level annotation. But for `ejb-jar.xml` method-permission, it is addictive (or union of both class and method level roles). The example of it is a bean `F` case.
			Yes.

Table 1–5 (Cont.) Annotations and Inheritance

Annotations	Reaction to Inheritance	Comment	OC4J Support

`@RolesAllowed`
`@DenyAll`
`@PermitAll`
(continues from preceding row)
	(continues from preceding row)
	- in bean `C`, guest and admin roles are allowed in the `foo` method: bean `C` has its own class-level annotation of `@RolesAllowed({guest},{admin})`, but cannot inherit any class-level annotations from its parent class `Base`. Since bean `C` overrides its parent's `foo` method and does not supply this overridden method with any annotations, the `foo` method in bean `C` sets the guest and admin roles;
	- in bean `D`, no roles are allowed (all denied) in the `bar` method: bean `D` has its own class-level annotation of `@DenyAll`, and cannot inherit any class-level annotations from its parent class `Base`. Since bean `D` overrides its parent's `bar` method and does not supply this overridden method with any annotations, the `bar` method in bean `D` denies all security permissions;
	- in bean `E`, all the roles are allowed (permit all) in the `bar` method: bean `E` has its own class-level annotation of `@RolesAllowed({guest},{admin})`, but cannot inherit any class-level annotations from its parent class `Base`. However, since bean `E` does not override its parent's `bar` method, bean `E` inherits this methods from bean `Base` along with the `@PermitAll` annotation applied to it;
	- for explanations on bean `F`, see Comments column.
		(continues from preceding row)	(continues from preceding row)

`@RunAs`	Class-level inheritance is not allowed.
	Example:
	`@RunAs ("bob")`
	`class Base {}`
	
	`@RunAs ("joe")`
	`class A extends Base {}`
	
	`class B extends Base {}`
	
	where:
	- run is defined as "joe" for bean `A`: as bean `A` cannot inherit any annotations from its parent class, bean `A` does not inherit run "bob" from its parent class `Base`;
	- run is not defined for bean `B`: as bean `B` cannot inherit any annotations from its parent class, bean `B` does not inherit role "bob" from its parent class `Base`;
			Yes.
			Ignores the superclass-level bean type annotation.

Table 1–5 (Cont.) Annotations and Inheritance

Annotations	Reaction to Inheritance	Comment	OC4J Support

`@DeclareRoles`	Class-level inheritance is not allowed.
	Example:
	`@DeclareRoles ({"bob"})`
	`class Base {}`
	
	`@DeclareRoles ({"joe"})`
	`class A extends Base {}`
	
	`class B extends Base {}`
	
	where:
	- bean `A` declares role "joe": as bean `A` cannot inherit any annotations from its parent class, bean `A` does not inherit role "bob" from its parent class `Base`;
	- bean `B` does not declare any roles: as bean `B` cannot inherit any annotations from its parent class, it does not inherit role "bob" from its parent class `Base`.
			Yes.

`@WebService`	Class-level inheritance is not allowed.
	Example:
	`@WebServices`
	`class Base {}`
	
	`@Stateless`
	`class A extends Base {}`
	
	where:
	- bean `A` is not a Web Service end point, because it does not inherit `@WebService` annotation from its parent class `Base`.
			Yes.
			Ignores the superclass-level bean type annotation.

`@StatefulDeployemt`
`@StatelessDeployemt`
`@MessageDrivenDeployemt`
	Class-level inheritance is not allowed.
	Example:
	`@StatefulDeployment (timeout=60)`
	`class Base {}`
	
	`@StatefulDeployment (timeout=30)`
	`class A extends Base {}`
	
	`class B extends Base {}`
	
	where:
	- bean `A` has a stateful deployment timeout of 30: bean `A` cannot inherit any annotations from its parent class, therefor bean `A` does not inherit a stateful deployment timeout of 60 from its parent class `Base`;
	- bean `B` does not have a timeout: bean `B` cannot inherit any annotations from its parent class, therefor bean `B` does not inherit a stateful deployment timeout of 60 from its parent class `Base`;
		These are OC4J-specific annotations: they are not defined in the EJB 3.0 specification.	Yes.
			Ignores the superclass-level bean type annotation.

Overriding Annotations With Deployment Descriptor Entries

You can combine the use of annotations and deployment descriptors in the design of your application. In this case, a deployment descriptor plays a role of an overriding mechanism for the annotations. For a list of rules that apply when XML descriptor is used to override annotations, see EJB 3.0 specification.

OC4J supports the annotations overriding rules defined in the EJB 3.0 specification. In the current release of OC4J, if a deployment descriptor overriding violates these rules, OC4J logs a warning, ignores the override, and uses the annotation configuration. For example, if you annotate a class with a `@Stateful` annotation, and then in the `ejb-jar.xml` file you override this with an `<entity>` entry, it would be a violation of the overriding rule: you cannot override the bean type. OC4J will behave as follows: it will log a warning, ignore the override, and continue to treat the class as a stateful session bean.

> **Note:** In future releases of OC4J, the warnings will be replaced with exceptions that will fail the deployment.

Table 1–6 lists overriding rules for annotations with XML that are defined in EJB 3.0 specification, as well as the behavior of OC4J (10.1.3.1 release, EJB layer) with regards to these rules.

Table 1–6 Overriding Annotations With XML

Scope	Annotations	XML	EJB 3.0 Specification Overriding Rules	OC4J-specific Behavior (10.1.3.1 Release)
Session bean type	`@Stateless` `@Stateful`	`<session-type>`	Section 19.2 of the EJB 3.0 specification states that if the bean's type has been specified by means of the `@Stateless`, `@Stateful`, or `@MessageDriven` annotation, its type cannot be overridden by means of the deployment descriptor. The bean's type (and its session type), if specified, must be the same as that specified in annotations.	If this rule is broken, OC4J logs a warning. Note: in 11g release of OC4J, the container will throw a validation exception.
Transaction type: BMT or CMT	`@TransactionManagement(TransactionManagementType.CONTAINER, TransactionManagementType.APPLICATION)`	`<transaction-type>`	Section 13.3.6 of the EJB 3.0 specification states that transaction type override is not allowed.	If this rule is broken, OC4J logs a warning. Note: in 11g release of OC4J, the container will throw a validation exception.
Transaction attribute	`@TransactionAttribute(TransactionAttributeType.REQUIRED){MANDATORY, REQUIRED, REQUIRES_NEW, SUPPORTS, NOT_SUPPORTED, NEVER}`	`<container-transaction>` `<trans-attribute>`	Section 13.3.7 of the EJB 3.0 specification states that XML is served as an alternative to metadata annotations to specify the transaction attributes (or as a means to supplement or override metadata annotations for transaction attributes). Transaction attributes specified in the deployment descriptor are assumed to override or supplement transaction attributes specified in annotations.	OC4J complies with the overriding rule.

Table 1–6 (Cont.) Overriding Annotations With XML

Scope	Annotations	XML	EJB 3.0 Specification Overriding Rules	OC4J-specific Behavior (10.1.3.1 Release)

Interceptor	`@Interceptors`
	`@ExcludeDefaultInterceptor`
	`@ExcludeClassInterceptor`
		`<interceptor-binding>`
		`<exclude-default-interceptors>`
		`<exclude-class-interceptors>`
		`<interceptor-order>`
			Section 12.8.2 of the EJB 3.0 specification states that the binding of interceptors to classes is additive. XML is used to augment the interceptors and interceptor methods defined by means of annotations. The specification also states that XML may be used as an alternative to specify the invocation order of interceptors or to override the order specified in metadata annotations.	OC4J does not allow multiple `interceptor-order` definitions. It cannot turn off `exclude-class-interceptors` and `exclude-default-interceptors` flags if `interceptor-order` is used. It cannot define `interceptor-class` outside of `interceptor-order`. Those are not defined in the EJB 3.0 specification.

Interceptor callback	`@PostConstruct`
	`@PreDestroy`
	`@PostActivate`
	`@PrePessivate`
	`@AroundInvoke`
		`<post-construct-method>`
		`<pre-destroy-method>`
		`<post-activate-method>`
		`<pre-passivate-method>`
		`<around-invoke-method>`
			Section 12.8.1 of the EJB 3.0 specification states that at most one method of a given interceptor class can be designated as an around-invoke method, post-construct method, pre-destroy method, pre-passivate method, or post-activate method, regardless of whether the deployment descriptor is used to define interceptors or some combination of annotations and deployment descriptor elements is used.	OC4J adds the life cycle callback method to the descriptor list without validating the singleton restraint.

Security identity	`@DeclareRoles` `@RunAs`	`<security-role>` `<role-name>`	Section 17.3.4 of the EJB 3.0 specification states that XML `<security-identity>` element can be used to override a security identity specified in metadata. The value of the `<security-identity>` element is either `use-caller-identity` or `run-as`.	OC4J complies with the overriding rule.

Table 1–6 (Cont.) Overriding Annotations With XML

Scope	Annotations	XML	EJB 3.0 Specification Overriding Rules	OC4J-specific Behavior (10.1.3.1 Release)

Method permission	`@RolesAllowed`
	`@DenyAll`
	`@PermitAll`
		`<method-permission>`	Section 17.3.2.2 of the EJB 3.0 specification states that the specification of the `<method-permission>` element in XML is served as an alternative to metadata annotations to specify the method permissions (or as a means to supplement or override metadata annotations for method permission values). Any values explicitly specified in the deployment descriptor override any values specified in annotations. The granularity of overriding is at the method level. The method permissions relation is defined as the union of all the method permissions defined in the individual `<method-permission>` elements.	OC4J complies with the overriding rule.

EJB reference	`@EJB` `@EJBs`	`<ejb-ref>` `<ejb-local-ref>`	Section 16.5.2.1 of the EJB 3.0 specification states that the following rules apply to how an XML entry may override an `@EJB` / `@EJBs` annotation: ■ The relevant deployment descriptor entry is located based on the JNDI name used with the annotation (either defaulted or provided explicitly). ■ The type specified in the deployment descriptor using the `<remote>`, `<local>`, `<remote-home>`, or `<local-home>` element and any bean referenced by the `<ejb-link>` element must be assignable to the type of the field or property, or the type specified by the `beanInterface` element of the `@EJB` annotation. ■ The description, if specified, overrides the description element of the annotation. ■ The injection target, if specified, must name exactly the annotated field or property method.	OC4J complies with the overriding rule.

Table 1–6 (Cont.) Overriding Annotations With XML

Scope	Annotations	XML	EJB 3.0 Specification Overriding Rules	OC4J-specific Behavior (10.1.3.1 Release)

Resource reference	`@Resource`
	`@Resources`
		`<env-entry>`
		`<resource-ref>`
		`<resource-env-ref>`
			Section 16.2.3 of the EJB 3.0 specification states that the following rules apply to how a XML entry may override a `@Resource` / `@Resources` annotations:
			■ The relevant deployment descriptor entry is located based on the JNDI name used with the annotation (either defaulted or provided explicitly).
			■ The type specified in the deployment descriptor must be assignable to the type of the field or property, or the type specified in the `@Resource` annotation.
			■ The description, if specified, overrides the description element of the annotation.
			■ The injection target, if specified, must name exactly the annotated field or property method.
			■ The `<res-sharing-scope>` element, if specified, overrides the shareable element of the annotation.
			■ The `<res-auth>` element, if specified, overrides the `authenticationType` element of the annotation.
				OC4J complies with the overriding rule.

Table 1–6 (Cont.) Overriding Annotations With XML

Scope	Annotations	XML	EJB 3.0 Specification Overriding Rules	OC4J-specific Behavior (10.1.3.1 Release)
Persistence unit	`@PersistenceUnits` `@PersistenceUnit`	`<persistence-units>` `<persistence-unit>`	Section 16.10.2.1 of the EJB 3.0 specification states that the following rules apply to how a XML entry may override a `@PersistenceUnit` / `@PersistenceUnits` annotation: ▪ The relevant deployment descriptor entry is located based on the JNDI name used with the annotation (either defaulted or provided explicitly). ▪ The `<persistence-unit-name>` element of the deployment descriptor overrides the `unitName` element of the annotation. ▪ The injection target, if specified, must name exactly the annotated field or property method.	OC4J complies with the overriding rule.

Table 1–6 (Cont.) Overriding Annotations With XML

Scope	Annotations	XML	EJB 3.0 Specification Overriding Rules	OC4J-specific Behavior (10.1.3.1 Release)
Persistence context	`@PersistenceContext` `@PersistenceContexts`	`<persistence-context>` `<persistence-contexts>`	Section 16.11.2 of the EJB 3.0 specification states that the following rules apply to how a XML entry may override a `@PersistenceContext` / `@PersistenceContexts` annotation: ■ The relevant deployment descriptor entry is located based on the JNDI name used with the annotation (either defaulted or provided explicitly). ■ The `<persistence-context-type>` element of the deployment descriptor overrides the `type` element of the annotation. ■ Any `<persistence-property>` elements are added to those specified by the `@PersistenceContext` / `@PersistenceContexts` annotation. If the name of a specified property is the same as one specified by the `@PersistenceContext` annotation, the value specified in the annotation is overridden. ■ The injection target, if specified, must name exactly the annotated field or property method.	OC4J complies with the overriding rule.
Timeout	`@Timeout`	<timeout-method>	Section 18.2.2.of the EJB 3.0 specification states that if the `@Timeout` annotation is used, or the bean implements the `TimedObject` interface, the `<timeout-method>` XML, if specified, can only be used to refer to the same method.	OC4J complies with the overriding rule.

Table 1–6 (Cont.) Overriding Annotations With XML

Scope	Annotations	XML	EJB 3.0 Specification Overriding Rules	OC4J-specific Behavior (10.1.3.1 Release)
Remove	`@Remove (retainIfException=true|false)`	`<remove-method>` `<retain-if-exception>`	Section 4.3.11 of the EJB 3.0 specification states that the XML `<retain-if-exception>` sub-element of the `<remove-method>` element may be explicitly specified to override the `retainIfException` value specified or defaulted by the `@Remove` annotation.	OC4J handles the remove methods properly on stateful session beans.
Activation configuration	`@MessageDriven activationConfig`	`<activation-config>`	Section 5.4.13 of the EJB 3.0 specification states that the activation configuration properties specified in the deployment descriptor are added to those specified by means of the `@MessageDriven` annotation. If a property of the same name is specified in both, the deployment descriptor value overrides the value specified in the annotation.	OC4J complies with the overriding rule. Note: there is a bug in the current release of OC4J: the container does not perform the override at all times. Instead, it creates a new activation configuration object to the list.
Deployment	`@StatefulDeployemt` `@StatelessDeployemt` `@MessageDrivenDeployemt`	`<session-deployment>` `<message-driven-deployment>`	These annotations are not defined in the EJB 3.0 specification: they are OC4J-specific.	OC4J handles these annotations in the following way: if applied, the deployment settings in XML will override the annotation.

OC4J Support for Annotation Attribute mappedName

OC4J supports `@EJB` and `@Resource` attribute `mappedName`. For these annotations, `mappedName` is the equivalent of the `location` attribute of `orion-ejb-jar.xml` in elements `session-deployment`, `entity-deployment`, and `message-driven-deployment`.

OC4J does not support the `mappedName` attribute in `@Stateless`, `@Stateful`, or `@MessageDriven` annotations.

What is a Session Bean?

A session bean is an EJB 3.0 or EJB 2.1 enterprise bean component created by a client for the duration of a single client/server session. A session bean performs operations for the client. Although a session bean can be transactional, it is not recoverable should a system failure occur. Session bean objects are either stateless (see "What is a Stateless Session Bean?" on page 1-28) or stateful: maintaining conversational state across method calls and transactions (see "What is a Stateful Session Bean?" on page 1-30). If a session bean maintains state, then OC4J manages this state if the object must be removed from memory ("When Does Stateful Session Bean Passivation Occur?" on page 1-32). However, the session bean object itself must manage its own persistent data.

From a client's perspective, a session bean is a nonpersistent object that implements some business logic running on the application server. For example, in an on-line store

application, you can use a session bean to implement a `ShoppingCartBean` that provides a `Cart` interface that the client uses to invoke such methods as `purchaseItem` and `checkout`.

Each client is allocated its own session object. A client does not directly access instances of the session bean's class: a client accesses a session object through the session bean's home ("Implementing the Home Interfaces" on page 11-6) and component ("Implementing the Component Interfaces" on page 11-8) interfaces. The client of a session bean may be a local client, a remote client, or a Web service client (stateless session bean only), depending on the interface provided by the bean and used by the client.

OC4J maintains a session context for each session bean instance (see "What is Session Context?" on page 1-34) that you use to make callback requests to the container.

This section describes the following:

- What is a Stateless Session Bean?
- What is a Stateful Session Bean?
- What is Session Context?

For more information, see the following:

- "Implementing an EJB 3.0 Session Bean" on page 4-1
- "Implementing an EJB 2.1 Session Bean" on page 11-1

What is a Stateless Session Bean?

A stateless session bean is a session bean with no conversational state. All instances of a particular stateless session bean class are identical.

A stateless session bean and its client do not share state or identity between method invocations. A stateless session bean is strictly a single invocation bean. It is employed for reusable business services that are not connected to any specific client, such as generic currency calculations, mortgage rate calculations, and so on. Stateless session beans may contain client-independent, read-only state across a call. Subsequent calls are handled by other stateless session beans in the pool. The information is used only for the single invocation.

OC4J maintains a pool of these stateless beans to service multiple clients. An instance is taken out of the pool when a client sends a request. There is no need to initialize the bean with any information.

The client of a stateless session bean may be a Web service client. Only a stateless session bean may provide a Web service client view.

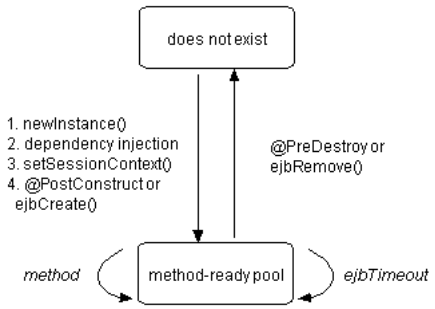
For more information, see the following:

- "Implementing an EJB 3.0 Stateless Session Bean" on page 4-1
- "Implementing an EJB 2.1 Stateless Session Bean" on page 11-1
- "Exposing a Stateless Session Bean as a Web Service" on page 30-1

What is the Stateless Session Bean Life Cycle?

Figure 1–3 shows the life cycle of a stateless session bean. Annotations (such as `@PostConstruct`) are applicable to EJB 3.0 stateless session beans only.

Figure 1–3 Stateless Session Bean Life Cycle



The life cycle for EJB 3.0 and EBJ 2.1 stateless session beans are identical. The difference is in how you register life cycle callback methods (see Table 1–7 and Table 1–8).

Table 1–7 lists the optional EJB 3.0 stateless session bean life cycle callback methods you can define using annotations. For EJB 3.0 stateless session beans, you do not need to implement these methods.

Table 1–7 Life Cycle Methods for an EJB 3.0 Stateless Session Bean

Annotation	Description
`@PostConstruct`	This optional method is invoked for a stateful session bean before the first business method invocation on the bean. This is at a point after which any dependency injection has been performed by the container.
`@PreDestroy`	This optional method is invoked for a stateful session bean when the instance is in the process of being removed by the container. The instance typically releases any resources that it has been holding.

Table 1–8 lists the EJB 2.1 life cycle methods, as specified in the `javax.ejb.SessionBean` interface, that a stateful session bean must implement. For EJB 2.1 stateful session beans, you must at the least provide an empty implementation for all callback methods.

Table 1–8 Life Cycle Methods for an EJB 2.1 Stateless Session Bean

EJB Method	Description
`ejbCreate`	The container invokes this method right before it creates the bean. Use this method to initialize nonclient-specific information such as retrieving a data source.
`ejbActivate`	This method is never called for a stateless session bean. Provide an empty implementation only.
`ejbPassivate`	This method is never called for a stateless session bean. Provide an empty implementation only.
`ejbRemove`	The container invokes this method before it ends the life of the stateless session bean. Use this method to perform any required clean-up (for example, closing external resources such as a data source).
`setSessionContext`	The container invokes this method after it first instantiates the bean. Use this method to obtain a reference to the context of the bean. For more information, see "Implementing the setSessionContext Method" on page 11-9.

For more information, see the following:

- "What is the Life Cycle of an Enterprise Bean?" on page 1-5
- "Configuring a Life Cycle Callback Interceptor Method on an EJB 3.0 Session Bean" on page 5-4
- "Configuring a Life Cycle Callback Interceptor Method on an Interceptor Class of an EJB 3.0 Session Bean" on page 5-5
- "Configuring a Life Cycle Callback Method for an EJB 2.1 Session Bean" on page 12-3

What is a Stateful Session Bean?

A stateful session bean is a session bean that maintains conversational state.

Stateful session beans are useful for conversational sessions, in which it is necessary to maintain state, such as instance variable values or transactional state, between method invocations. These session beans are mapped to a single client for the life of that client.

A stateful session bean maintains its state between method calls. Thus, there is one instance of a stateful session bean created for each client. Each stateful session bean contains an identity and a one-to-one mapping with an individual client.

When the container determines that it must remove a stateful session bean from memory (in order to release resources), the container maintains the bean's state by passivation (serializing the bean to disk). This is why the state that you passivate must be serializable. However, this information does not survive system failures. When the bean instance is requested again by its client, the container activates the previously passivated bean instance.

The type of state that is saved does not include resources. The container invokes the `ejbPassivate` method within the bean to provide the bean with a chance to clean up its resources, such as sockets held, database connections, and hash tables with static information. All these resources can be reallocated and re-created during the `ejbActivate` method.

> **Note:** You can turn off passivation for stateful session beans (see "Configuring Passivation" on page 12-1).

If the bean instance fails, the state can be lost, unless you take action within your bean to continually save state. However, if you must make sure that state is persistently saved in the case of failovers, you may want to use an entity bean for your implementation. Alternatively, you could also use the `SessionSynchronization` interface to persist the state transactionally.

For example, a stateful session bean could implement the server side of a shopping cart on-line application, which would have methods to return a list of objects that are available for purchase, put items in the customer's cart, place an order, change a customer's profile, and so on.

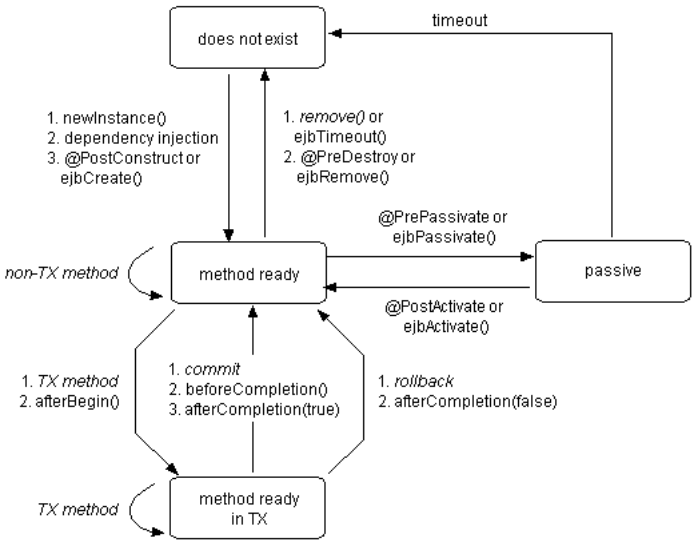
For more information, see the following:

- "Implementing an EJB 3.0 Stateful Session Bean" on page 4-2
- "Implementing an EJB 2.1 Stateful Session Bean" on page 11-3

What is the Life Cycle of a Stateful Session Bean?

Figure 1–4 shows the life cycle of a stateful session bean. Annotations (such as `@PostConstruct`) are applicable to EJB 3.0 stateful session beans only.

Figure 1–4 Stateful Session Bean Life Cycle



The life cycle for EJB 3.0 and EBJ 2.1 stateful session beans are identical. The difference is in how you register life cycle callback methods (see Table 1–9 and Table 1–10).

Table 1–9 lists the optional EJB 3.0 stateful session bean life cycle callback methods you can define using annotations. For EJB 3.0 stateful session beans, you do not need to implement these methods.

Table 1–9 Life Cycle Methods for an EJB 3.0 Stateful Session Bean

Annotation	Description
`@PostConstruct`	This optional method is invoked for a stateful session bean before the first business method invocation on the bean. This is at a point after which any dependency injection has been performed by the container.
`@PreDestroy`	This optional method is invoked for a stateful session bean when the instance is in the process of being removed by the container. The instance typically releases any resources that it has been holding.
`@PrePassivate`	The container invokes this method right before it passivates a stateful session bean. For more information, see the following: ▪ "When Does Stateful Session Bean Passivation Occur?" on page 1-32 ▪ "What Object Types can be Passivated?" on page 1-33 ▪ "Where is a Passivated Stateful Session Bean Stored?" on page 1-34
`@PostActivate`	The container invokes this method right after it reactivates a formerly passivated stateful session bean.

Table 1–10 lists the EJB 2.1 life cycle methods, as specified in the `javax.ejb.SessionBean` interface, that a stateful session bean must implement. For EJB 2.1 stateful session beans, you must at the least provide an empty implementation for all callback methods.

Table 1–10 Life Cycle Methods for an EJB 2.1 Stateful Session Bean

EJB Method	Description
`ejbCreate`	The container invokes this method right before it creates the bean. Stateless session beans must do nothing in this method. Stateful session beans can initiate state in this method.
`ejbActivate`	The container invokes this method right after it reactivates the bean.
`ejbPassivate`	The container invokes this method right before it passivates the bean. For more information, see the following: ■ "When Does Stateful Session Bean Passivation Occur?" on page 1-32 ■ "What Object Types can be Passivated?" on page 1-33 ■ "Where is a Passivated Stateful Session Bean Stored?" on page 1-34
`ejbRemove`	A container invokes this method before it ends the life of the session object. This method performs any required clean-up (for example, closing external resources such as file handles).
`setSessionContext`	The container invokes this method after it first instantiates the bean. Use this method to obtain a reference to the context of the bean. For more information, see "Implementing the setSessionContext Method" on page 11-9.

For more information, see the following:

- "What is the Life Cycle of an Enterprise Bean?" on page 1-5
- "Configuring a Life Cycle Callback Interceptor Method on an EJB 3.0 Session Bean" on page 5-4
- "Configuring a Life Cycle Callback Interceptor Method on an Interceptor Class of an EJB 3.0 Session Bean" on page 5-5
- "Configuring a Life Cycle Callback Method for an EJB 2.1 Session Bean" on page 12-3

When Does Stateful Session Bean Passivation Occur? Passivation enables the container to preserve the conversational state of an inactive idle bean instance by serializing the bean and its state into a secondary storage and removing it from memory. Before passivation, the container invokes the `PrePassivate` or `ejbPassivate` method enabling the bean developer to clean up held resources, such as database connections, TCP/IP sockets, or any resources that cannot be transparently passivated using object serialization. Only certain object types can be serialized and passivated (see "What Object Types can be Passivated?" on page 1-33).

Passivation is enabled by default. For more information on enabling and disabling passivation, see "Configuring Passivation" on page 12-1.

OC4J will passivate stateful session beans when any combination of the following criteria is met:

- exceed idle timeout;
- exceed threshold for maximum number of instances or exceed absolute maximum number of instances;
- exceed threshold for maximum JVM memory consumption;
- shutdown OC4J instance.

Passivation of beans is performed using the least recently used algorithm: of the beans eligible for passivation, OC4J passivates the least used first.

In addition, you can specify how frequently OC4J checks this criterion and the number of instances to passivate when the criterion is met.

For information on configuring this criterion, see "Configuring Passivation Criteria" on page 12-2.

If the passivation serialization fails, then the container attempts to recover the bean back to memory as if nothing happened. No future passivation attempts will occur for any beans that fail passivation. Also, if activation fails, the bean and its references are completely removed from the container.

When a client invokes one of the methods of the passivated bean instance, the preserved conversational state data is activated by deserializing the bean from secondary storage, and bringing back into memory. Before activation, the container invokes the `ejbActivate` method so that you can restore the resources released during `ejbPassivate`. For more information on passivation, see the EJB specification.

A stateful session bean can passivate only certain object types, as designated in "What Object Types can be Passivated?" on page 1-33. If you do not prepare your stateful session beans for passivation by releasing all resources and only letting state to exist within the allowed object types, then passivation will always fail.

If new bean data is propagated to a passivated bean in a cluster, then the bean instance data is overwritten by the propagated data.

What Object Types can be Passivated? When a stateful session bean is passivated, it is serialized to secondary storage. To be successful, the conversational state of a bean must consist of only primitive values and the following data types:

- serializable object (you do not need to declare the field type as serializable as long as the field is initialized with a subclass of the field type that is serializable);
- null;
- reference to an EJB business interface;
- reference to an EJB remote interface, even if the stub class is not serializable;
- reference to an EJB remote home interface, even if the stub class is not serializable;
- reference to an EJB local interface, even if it is not serializable;
- reference to an EJB local home interface, even if it is not serializable;
- reference to the `SessionContext` object, even if it is not serializable;
- reference to the environment naming context (that is, the `java:comp/env` JNDI context) or any of its subcontexts;
- reference to the `UserTransaction` interface;
- reference to resource manager connection factory;
- reference to an `EntityManager` object, even if it is not serializable;
- reference to an `EntityManagerFactory` object, even if it is not serializable;
- reference to `javax.ejb.Timer` object;
- An object that is not directly serializable, but becomes serializable by replacing a reference to an EJB business interface, EJB home and component interfaces, the reference to the `SessionContext` object, the reference to the `java:comp/env` JNDI context and its subcontexts, the reference to the `UserTransaction` interface, and the reference to the `EntityManager`, `EntityManagerFactory`, or both by serializable objects during the object's serialization.

You are responsible for ensuring that all nontransient fields are of these types after the `PrePassivate` method (see "Configuring a Life Cycle Callback Interceptor Method on an EJB 3.0 Session Bean" on page 5-4) or `ejbPassivate` method (see "Configuring a Life Cycle Callback Method for an EJB 2.1 Session Bean" on page 12-3) completes. Within this method, you must set all transient or nonserializable fields to null.

Where is a Passivated Stateful Session Bean Stored? By default, when OC4J passivates a stateful session bean, it writes the serialized instance to `<OC4J_HOME>\j2ee\home\persistence`.

Passivation uses space within this directory to store the passivated beans. If passivation allocates large amounts of disk space, you may need to change the directory to a place on your system where you have the space available (see "Configuring Passivation Location" on page 12-3).

What is Session Context?

OC4J maintains a `javax.ejb.SessionContext` for each session bean instance and makes this session context available to the beans. The bean may use the methods in the session context to make callback requests to the container. In addition, you can use the methods inherited from `EJBContext` (see "What is EJB Context?" on page 1-6).

For more information, see the following:

- "Accessing an EJB 3.0 EJBContext" on page 29-20
- "Accessing an EJB 2.1 EJBContext" on page 29-27

OC4J initializes the session context after it first instantiates the bean. It is the bean provider's responsibility to enable the bean to retrieve the session context. The container will never call this method from within a transaction context. If the bean does not save the session context at this point, the bean will never gain access to the session context.

When the container calls this method, it passes the reference of the `SessionContext` object to the bean. The bean can then store the reference for later use.

If the session bean instance stores in its conversational state an object reference to the `SessionContext` (either with a `setSessionContext` method or using resource injection), OC4J can save and restore the reference across the instance's passivation. OC4J can replace the original `SessionContext` object with a different and functionally equivalent `SessionContext` object during activation.

Note: OC4J does not support `SessionContext` method `getInvokedBusinessInterface`. If you call this method, OC4J throws an `UnsupportedOperationException`.

What is a JPA Entity?

The Java Persistence API (JPA), part of the Java Enterprise Edition 5 (Java EE 5) EJB 3.0 specification, greatly simplifies Java persistence and provides an object-relational mapping approach that lets you declaratively define how to map Java objects to relational database tables in a standard, portable way that works both inside a Java EE 5 application server and outside an EJB container in a Java Standard Edition 5 (Java SE 5) application.

Using JPA, you can designate any POJO class as a JPA entity–a Java object whose nontransient fields should be persisted to a relational database using the services of an

entity manager obtained from a JPA persistence provider (either within a Java EE EJB container or outside of an EJB container in a Java SE application).

An entity has the following characteristics:

- it is EJB 3.0-compliant;
- it is light-weight;
- it manages persistent data in concert with a JPA entity manager;
- it performs complex business logic;
- it potentially uses several dependent Java objects;
- it can be uniquely identified by a primary key.

Entities represent persistent data stored in a relational database automatically using container-managed persistence.They are persistent because their data is stored persistently in some form of data storage system, such as a database: they do survive a server failure, failover, or a network failure. When an entity is reinstantiated, the state of the previous instance is automatically restored.

An entity models a business entity or multiple actions within a single business process. Entities are often used to facilitate business services that involve data and computations on that data. For example, you might implement an entity to retrieve and perform computation on items within a purchase order. Your entity can manage multiple, dependent, persistent objects in performing its tasks.

Entities can represent fine-grained persistent objects, because they are not remotely accessible components.

An entity can aggregate objects together and effectively persist data and related objects using the transactional, security, and concurrency services of a JPA persistence provider.

This section describes the following:

- What are JPA Entity Container-Managed Persistent Fields?
- What are JPA Entity Container-Managed Relationship Fields?
- How do you Avoid Database Resource Contention?
- What is the JPA Entity Life Cycle?
- What is a JPA Entity Primary Key?
- How do you Query for a JPA Entity?

For more information, see "Implementing a JPA Entity" on page 6-1.

What are JPA Entity Container-Managed Persistent Fields?

A container-managed persistent field is a state-field that represents data that must be persisted to a database.

All the data members of a JPA entity are considered persistent fields unless annotated with `@Transient`.

The JPA persistence provider that you specify in the entity's persistence unit (see "What is the persistence.xml File?" on page 2-8) is responsible for ensuring that persistent fields are persisted to the database.

By default, a JPA persistence provider automatically configures a basic mapping for most Java primitive types, wrappers of the primitive types, and enumerations. You can

customize this mapping using the `@Basic`, `@Enumerated`, `@Temporal`, and `@Lob` annotations.

What are JPA Entity Container-Managed Relationship Fields?

A container-managed relationship (CMR) field is an association-field that represents a persistent relationship to one or more other EJB 3.0 entities or EJB 2.1 container-managed entity beans. For example, in an order management application, the `OrderEJB` might be related to a collection of `LineItemEJB` beans and to a single `CustomerEJB` bean.

All the data members of a JPA entity are considered persistent fields unless annotated with `@Transient`.

The JPA persistence provider you specify in the entity's persistence unit (see "What is the persistence.xml File?" on page 2-8) is responsible for ensuring that persistent fields are persisted to the database.

You must configure your entity (using annotations or `persistence.xml`) to specify mappings to other entities. This configuration specifies how the entities relate to one another and how a JPA persistence provider should map the reference to a relational database.

For example, you can configure a relationship mapping for any persistent relationship using relationship mapping annotations `@OneToOne`, `@ManyToOne`, `@OneToMany`, and `@ManyToMany`.

An entity relationship has the following characteristics:

- Multiplicity–there are four types of multiplicities all of which are supported by Oracle Application Server: one-to-one, many-to-one, one-to-many, and many-to-many.
- Directionality–the direction of a relationship may be either bi-directional or unidirectional. In a bi-directional relationship, each entity bean has a relationship field that refers to the other bean. Through the relationship field, an entity bean's code can access its related object. If an entity bean has a relative field, then it "knows" about its related object. For example, if an `ProjectEJB` bean knows what `TaskEJB` beans it has, and if each TaskEJB bean knows to which `ProjectEJB` bean it belongs, then they have a bi-directional relationship. In a unidirectional relationship, only one entity bean has a relationship field that refers to the other. Oracle Application Server supports both unidirectional and bi-directional relationships between enterprise beans.
- Java Persistence query language support–JP QL is an extension of the Enterprise JavaBeans query language (EJB QL) that adds bulk update and delete, `JOIN`, `GROUP BY`, `HAVING`, projection, subqueries, and named parameters. It supports both static and dynamic queries.

  JP QL queries often navigate across relationships. The direction of a relationship determines whether a query can navigate from one bean to another.

For more information, see the following:

- "Configuring a Container-Managed Relationship Field for a JPA Entity" on page 7-9
- "Implementing JPA Queries" on page 8-1

What is the JPA Entity Life Cycle?

Figure 1–5 shows the life cycle of a JPA entity.

Figure 1–5 JPA Entity Life Cycle

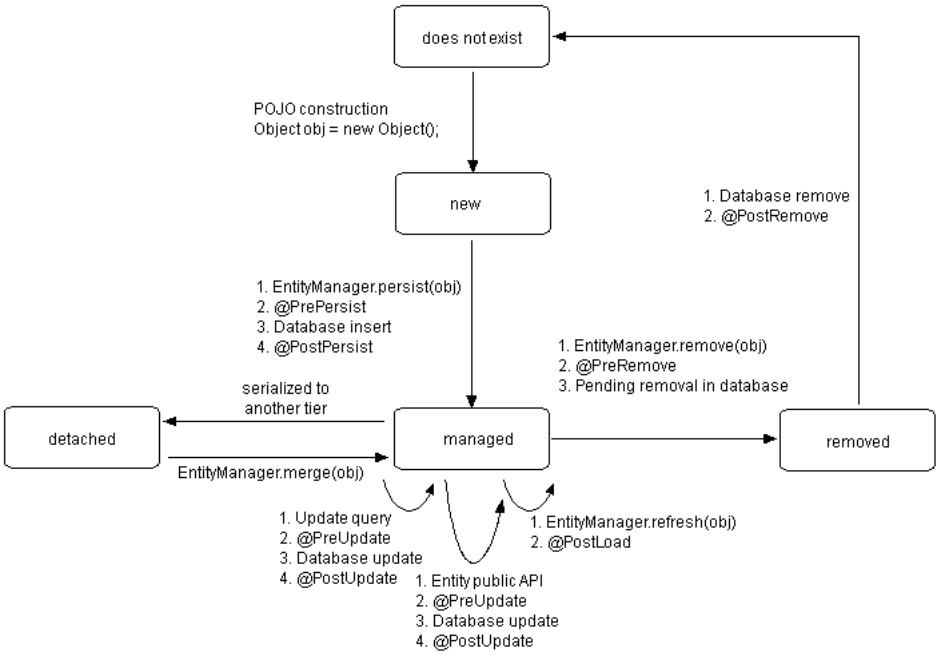


Table 1–11 lists the optional JPA entity life cycle callback methods you can define using annotations. For EJB 3.0 entities, you do not need to implement these methods.

Table 1–11 Life Cycle Methods for a JPA Entity

Annotation	Description
`@PrePersist`	This optional method is invoked for an entity before the corresponding `EntityManager` persist operation is executed. This callback will be invoked on all entities to which these operations are cascaded. If this callback throws an `Exception`, it will cause the current transaction to be rolled back.
`@PostPersist`	This optional method is invoked for an entity after the corresponding `EntityManager` persist operation is executed. This callback will be invoked on all entities to which these operations are cascaded. This method will be invoked after the database insert operation. This may be directly after the persist operation, a flush operation, or at the end of a transaction. If this callback throws an `Exception`, it will cause the current transaction to be rolled back.
`@PreRemove`	This optional method is invoked for an entity before the corresponding `EntityManager` remove operation is executed. This callback will be invoked on all entities to which these operations are cascaded. If this callback throws an `Exception`, it will cause the current transaction to be rolled back.
`@PostRemove`	This optional method is invoked for an entity after the corresponding `EntityManager` remove operation is executed. This callback will be invoked on all entities to which these operations are cascaded. This method will be invoked after the database delete operation. This may be directly after the remove operation, a flush operation, or at the end of a transaction. If this callback throws an `Exception`, it will cause the current transaction to be rolled back.

Table 1–11 (Cont.) Life Cycle Methods for a JPA Entity

Annotation	Description
`@PreUpdate`	This optional method is invoked before the database update operation on entity data. This may be at the time of the entity state update, a flush operation, or at the end of a transaction. OC4J calls this method only if it determines that an actual update is required (only if it is prepared to send SQL to the database). Contrast this with a post-update callback which is called regardless of whether or not an actual change was required.
`@PostUpdate`	This optional method is invoked after the database update operation on entity data. This may be at the time of the entity state update, a flush operation, or at the end of a transaction. OC4J calls this method even if it determines that no actual update is required (even if it determines that no SQL needs to be sent to the database). Use the pre-update callback if you want to be notified only when the object has actually been changed.
`@PostLoad`	This optional method is invoked after the entity has been loaded into the current persistence context from the database or after the refresh operation has been applied to it and before a query result is returned or accessed or an association is traversed.

For more information, see the following:

- "What is the Life Cycle of an Enterprise Bean?" on page 1-5
- "Configuring a Life Cycle Callback Method on a JPA Entity" on page 7-16
- "Configuring a Life Cycle Callback Listener Method on an Entity Listener Class of a JPA Entity" on page 7-17

What is a JPA Entity Primary Key?

Each JPA entity must have a primary key that uniquely identifies it from other instances. The primary key (or the fields contained within a complex primary key) must be persistent fields.

All fields within the primary key are restricted to the following:

- primitive object types;
- serializable types;
- types that can be mapped to SQL types.

In this release, you can define a primary key made up of a single, well-known serializable Java primitive or object type. The primary key variable that is declared within the bean class must be declared as `public` (see "Configuring a JPA Entity Simple Primary Key Field" on page 7-2).

You can assign primary key values yourself, or more typically, you can create an auto-generated primary key (see "Configuring JPA Entity Automatic Primary Key Generation").

Note: Once the primary key for an entity bean has been set, the EJB 3.0 specification forbids you from attempting to change it. Therefore, do not expose the primary key set methods in an entity component interface.

For more information, see "Configuring a JPA Entity Primary Key" on page 7-1

How do you Query for a JPA Entity?

In EJB 3.0, you use a `javax.persistence.EntityManager` to create, find, merge, and persist your EJB 3.0 entities. To find entities, you use the `EntityManager` query API (see "Understanding the JPA EntityManager Query API" on page 1-39).

You can express your selection criteria using an appropriate query syntax (see "Understanding JPA Entity Query Syntax" on page 1-39).

Using query hints, you can use EJB 3.0 JPA persistence provider vendor extensions to this API (see "Configuring TopLink Query Hints in a JPA Query" on page 8-3).

Understanding the JPA EntityManager Query API

In EJB 3.0, you can use the `javax.persistence.EntityManager` and `javax.persistence.Query` API to create and execute named queries or dynamic queries.

Using `Query` API, you can bind parameters, configure hints, and control the number of results returned.

For more information, see the following:

- "What is a JPA Dynamic (Ad-Hoc) Query?" on page 1-39
- "What is a JPA Named (Predefined) Query?" on page 1-39
- "Querying for a JPA Entity Using the EntityManager" on page 29-13

What is a JPA Named (Predefined) Query? A named query is the EJB 3.0 improvement of the EJB 2.1 finder method. In EJB 3.0, you can implement a named query using metadata (see "Implementing a JPA Named Query" on page 8-1), and then create and execute the query by name at run time (see "Creating a Named Query With the EntityManager" on page 29-13).

OC4J supports both Java persistence query language and native SQL named queries.

What is a JPA Dynamic (Ad-Hoc) Query? A dynamic query is a query that you can compose, configure, and execute at run time. You can use dynamic queries in addition to named queries.

OC4J supports both Java persistence query language and native SQL named queries.

You can also create a dynamic query using the TopLink query and expression framework (see "Creating a Dynamic TopLink Expression Query With the EntityManager" on page 29-14).

Understanding JPA Entity Query Syntax

Table 1–16 summarizes the types of query syntax you can use to define queries for EJB 3.0 entities.

Table 1–12 OC4J JPA Entity Query Syntax Support

Query Syntax	See Also
Java Persistence Query Language	"Understanding Java Persistence Query Language Query Syntax" on page 1-40
Native SQL	"Understanding Native SQL Query Syntax in EJB 2.1" on page 1-52

Oracle recommends the use of Java persistence query language, because it is both portable and optimizable.

Understanding Java Persistence Query Language Query Syntax Java persistence query language is a specification language used to define query semantics in a portable and optimizable format.

Although similar to SQL, Java persistence query language offers significant advantages over native SQL. While SQL applies queries against tables using column names, Java persistence query language applies queries against EJB 3.0 entities using the abstract schema name and the fields of the bean within the query. The Java persistence query language statement retains the object terminology. The JPA persistence provider translates the Java persistence query language statement to the appropriate database SQL statement when the application is deployed. Thus, the JPA persistence provider is responsible for converting the entity name and the names of its persistent fields to the appropriate database table and column names. Java persistence query language is portable to all databases supported by OC4J.

In EJB 3.0, Java persistence query language syntax includes everything that is in EJB 2.1 EJB QL (see "Understanding EJB 2.1 Query Syntax" on page 1-50), plus additional features such as bulk update and delete, `JOIN` operations, `GROUP BY`, `HAVING`, projection, subqueries, and the use of Java persistence query language in dynamic queries using the EJB 3.0 `EntityManager` API (see "What is a JPA Dynamic (Ad-Hoc) Query?" on page 1-39).

For more information, see the JSR-220 Enterprise JavaBeans v.3.0 Java Persistence API specification, Chapter 4.

OC4J provides complete support for Java persistence query language with the following important features:

- Automatic Code Generation: Java persistence query language queries are defined in the deployment descriptor of the entity bean. When enterprise beans are deployed to Oracle Application Server, the container automatically translates the queries into the SQL dialect of the target data store. Due to this translation, entity beans with container-managed persistence are portable: their code is not tied to a specific type of a data store.
- Optimized SQL Code Generation: Further, in generating the SQL code, Oracle Application Server makes several optimizations such as the use of bulk SQL, batched statement dispatch, and so on to make database access efficient.
- Support for Oracle and Non-Oracle Databases: Further, Oracle Application Server provides the ability to execute Java persistence query language against any database such as Oracle, MS SQL-Server, IBM DB/2, Informix, and Sybase.
- Relationships: Oracle Application Server supports Java persistence query language for both single entity beans and also with entity beans that have relationships, with support for any type of multiplicity and directionality.

Using EJB 3.0, OC4J supports all of the enhanced Java persistence query language features defined in the EJB 3.0 persistence specification, including SQRT and date, time, and timestamp options.

Understanding Native SQL Query Syntax in EJB 3.0 In this release, the TopLink JPA persistence provider takes the query syntax you specify (see "Understanding JPA Entity Query Syntax" on page 1-39) and generates Sequential Query Language (SQL) native to your underlying relational database.

Java persistence query language is the preferred syntax, because it is portable and optimizable.

Native SQL is appropriate for taking advantage of advanced query features of your underlying relational database that Java persistence query language does not support.

OC4J supports native SQL in both named and dynamic queries.

What is an EJB 2.1 Entity Bean?

An entity bean is an EJB 2.1 enterprise bean component that manages persistent data, performs complex business logic, potentially uses several dependent Java objects, and can be uniquely identified by a primary key.

Entity beans persist business data using one of the two following methods:

- Automatically by the container using an entity bean with container-managed persistence (see "What is an EJB 2.1 Entity Bean With Container-Managed Persistence?" on page 1-42)
- Programmatically through methods implemented in an entity bean with bean-managed persistence (see "What is an EJB 2.1 Entity Bean With Bean-Managed Persistence?" on page 1-46). These methods use JDBC, SQLJ, or a persistence framework (such as TopLink) to manage persistence.

For information on choosing between container-managed persistence and container-managed persistence architectures, see "When do you use Bean-Managed Versus Container-Managed Persistence?" on page 1-59.

Entity beans are persistent because their data is stored persistently in some form of data storage, such as a database: entity beans survive a server failure, failover, or a network failure. When an entity bean is reinstantiated, the state of the previous instance is automatically restored. OC4J manages this state if the entity bean must be removed from memory (see "When Does Entity Bean Passivation Occur?" on page 1-48).

An entity bean models a business entity or multiple actions within a single business process. Entity beans are often used to facilitate business services that involve data and computations on that data. For example, you might implement an entity bean to retrieve and perform computation on items within a purchase order. Your entity bean can manage multiple, dependent, persistent objects in performing its tasks.

A common design pattern pairs entity beans with a session bean that acts as the client interface. The entity bean functions as a coarse-grained object that encapsulates functionality and represents persistent data and relationships to dependent (typically, find-grained) objects. Thus, you decouple the client from the data so that if the data changes, the client is not affected. For efficiency, the session bean can be collocated with entity beans and can coordinate between multiple entity beans through their local interfaces. This is known as a session facade design. See the `http://java.sun.com` Web site for more information on session facade design.

An entity bean can aggregate objects together and effectively persist data and related objects using container transactional, security, and concurrency services.

This section describes the following:

For more information, see "Implementing an EJB 2.1 Entity Bean" on page 13-1.

What is an EJB 2.1 Entity Bean With Container-Managed Persistence?

When you choose to have the container manage your persistent data for an entity bean, you define an entity bean with container-managed persistence. A class of an entity bean with container-managed persistence is an abstract class (the container provides the implementation class that is used at run time), whose persistent data is specified as container-managed persistent fields (see "What are Container-Managed Persistent Fields?" on page 1-42) for simple data, or as container-managed relationship fields (see "What are Container-Managed Relationship Fields?" on page 1-42) for relationships with other entity beans with container-managed persistence. In this case, you do not have to implement some of the callback methods to manage persistence for your bean's data (see "What is the Life Cycle of an EJB 2.1 Entity Bean With Container-Managed Persistence?" on page 1-43), because the container stores and reloads your persistent data to and from the database. When you use container-managed persistence, the container invokes a persistence manager class that provides the persistence management business logic. OC4J uses the TopLink persistence manager by default. In addition, you do not have to provide management for the primary key (see "What is a Primary Key of an Entity Bean With Container-Managed Persistence?" on page 1-45): the container provides this key for the bean.

For more information, see the following:

- "Implementing an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 13-1
- "What is Entity Context?" on page 1-48
- "How do you Avoid Database Resource Contention?" on page 1-59
- "How do you Query for an EJB 2.1 Entity Bean?" on page 1-50
- "When Does Entity Bean Passivation Occur?" on page 1-48
- "What are Entity Bean Commit Options?" on page 1-48

What are Container-Managed Persistent Fields?

A container-managed persistent field is a state-field that represents data that must be persisted to a database.

By specifying a container-managed persistent field, you are instructing OC4J to take responsibility for ensuring that the field's value is persisted to the database. All other fields in the entity bean with container-managed persistence are considered nonpersistent (transient).

Using EJB 2.1, you must explicitly specify container-managed persistent fields (see "Configuring a Container-Managed Persistent Field for an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 14-7).

What are Container-Managed Relationship Fields?

A container-managed relationship field is an association-field that represents a persistent relationship to one or more other entity beans with container-managed persistence. For example, in an order management application the `OrderEJB` might be related to a collection of `LineItemEJB` beans and to a single `CustomerEJB` bean.

By specifying a container-managed relationship field, you are instructing OC4J to take responsiblity for ensuring that a reference to one or more related entity beans with

container-managed persistence is persisted to the database. For this reason, a relationship between entity beans with container-managed persistence is often referred to as container-managed relationship or a mapping from one entity bean with container-managed persistence to another.

A container-managed relationship has the following characteristics:

- Multiplicity–there are four types of multiplicities all of which are supported by Oracle Application Server:
- Directionality–the direction of a relationship may be either bi-directional or unidirectional. In a bi-directional relationship, each entity bean has a relationship field that refers to the other bean. Through the relationship field, an entity bean's code can access its related object. If an entity bean has a relative field, then it "knows" about its related object. For example, if an `ProjectEJB` bean knows what `TaskEJB` beans it has and if each TaskEJB bean knows to which `ProjectEJB` bean it belongs, then they have a bi-directional relationship. In a unidirectional relationship, only one entity bean has a relationship field that refers to the other. Oracle Application Server supports both unidirectional and bi-directional relationships between enterprise beans.
- EJB QL query support–EJB QL queries often navigate across relationships. The direction of a relationship determines whether a query can navigate from one bean to another. With OC4J, EJB QL queries can traverse container-managed relationships with any type of multiplicity and with both unidirectional and bi-directional relationships.

For more information, see the following:

- "Configuring a Container-Managed Relationship Field for an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 14-9
- "Implementing EJB 2.1 Queries" on page 16-1

What is the Life Cycle of an EJB 2.1 Entity Bean With Container-Managed Persistence?

Figure 1–6 shows the life cycle of an EJB 2.1 entity bean with container-managed persistence.

Figure 1–6 Life Cycle of EJB 2.1 Entity Bean With Container-Managed Persistence

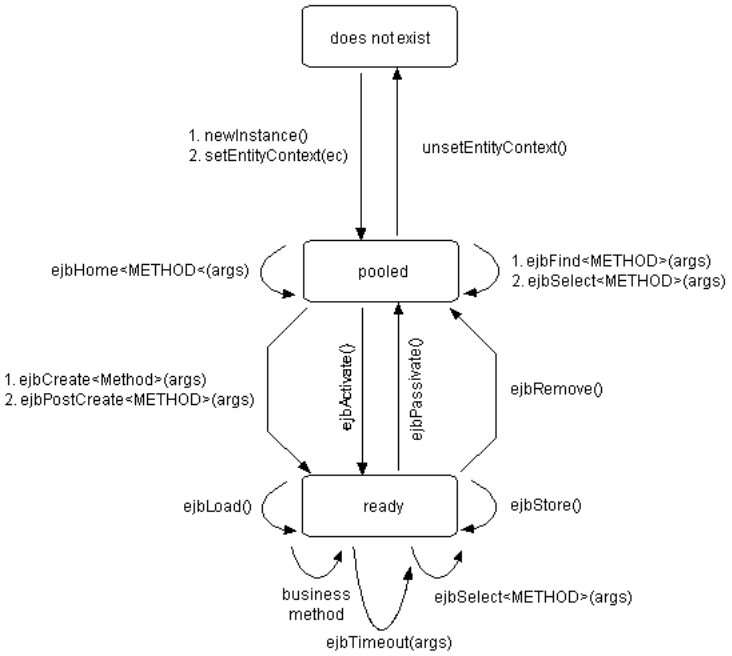


Table 1–13 lists the EJB 2.1 enterprise bean's life cycle methods, as specified in the `javax.ejb.EntityBean` interface, that an entity bean with container-managed persistence must implement. For EJB 2.1 entity beans with container-managed persistence, you must at the least provide an empty implementation for all callback methods.

Table 1–13 Life Cycle Methods for an EJB 2.1 Entity Bean With Container-Managed Persistence

EJB Method	Description
`ejbCreate`	You must implement an `ejbCreate` method corresponding to each `create` method declared in the home interface. When the client invokes the `create` method, the container first invokes the constructor to instantiate the object, then it invokes the corresponding `ejbCreate` method. For an entity bean with container-managed persistence, use this method to initialize container-managed persistent fields. The return type of all ejbCreate methods is the type of the bean's primary key. Optionally, you can initialize the bean with a unique primary key and return it. If you rely on the container to create and initialize the primary key, return `null`.
`ejbPostCreate`	The container invokes this method after the environment is set. For each `ejbCreate` method, an `ejbPostCreate` method must exist with the same arguments. For an entity bean with container-managed persistence, you can leave this implementation empty, or use your implementation to initialize parameters within or from the entity context.
`ejbRemove`	The container invokes this method before it ends the life of the entity bean. For an entity bean with container-managed persistence, you can leave this implementation empty, or use your implementation to perform any required clean-up (for example, closing external resources such as file handles).
`ejbStore`	The container invokes this method right before a transaction commits. It saves the persistent data to an outside resource, such as a database. For an entity bean with container-managed persistence, you can leave this implementation empty.

Table 1–13 (Cont.) Life Cycle Methods for an EJB 2.1 Entity Bean With Container-Managed Persistence

EJB Method	Description
`ejbLoad`	The container invokes this method when the data should be reinitialized from the database. This normally occurs after activation of an entity bean. For an entity bean with container-managed persistence, you can leave this implementation empty.
`ejbActivate`	The container calls this method directly before it activates an object that was previously passivated. Perform any necessary reaquisition of resources in this method.
`ejbPassivate`	The container calls this method before it passivates the object. Release any resources that can be easily re-created in `ejbActivate`, and save storage space. Typically, you want to release resources that cannot be passivated, such as sockets or database connections. Retrieve these resources in the `ejbActivate` method.

For more information, see the following:

- "What is the Life Cycle of an Enterprise Bean?" on page 1-5
- "Configuring a Life Cycle Callback Method for an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 14-15

What is a Primary Key of an Entity Bean With Container-Managed Persistence?

Each entity bean instance has a primary key that uniquely identifies it from other instances. You must declare the primary key (or the fields contained within a complex primary key) as a container-managed persistent field in the deployment descriptor.

All fields within the primary key are restricted to the following:

- primitive object types;
- serializable types;
- types that can be mapped to SQL types.

You can define a primary key in one of the following ways:

- Define a simple primary key made up of a single, well-known serializable Java primitive or object type. The primary key variable that is declared within the bean class must be declared as `public` (see "Configuring a Primary Key Field for an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 14-2).
- Define a composite primary key class made up of one or more well-known serializable Java primitive and object types within a `<`*`name`*`>PK` class that is serializable (see "Configuring a Composite Primary Key Class for an EJB 2.1 Entity Bean With Container-Managed Persistence").

Typically, you rely on OC4J to assign primary key values automatically. To configure how OC4J assigns primary key values, you use TopLink persistence API. For more information, see the following:

- "Customizing the TopLink EJB 2.1 Persistence Manager" on page 3-13
- "Understanding Sequencing in Relational Projects" in the *Oracle TopLink Developer's Guide*

> **Note:** Once the primary key for an entity bean has been set, the EJB 2.1 specification forbids you from attempting to change it. Therefore, do not expose the primary key set methods in an entity bean component interface.

For more information, see "Configuring a Primary Key for an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 14-2.

What is an EJB 2.1 Entity Bean With Bean-Managed Persistence?

When you choose to manage your persistent data for an entity bean yourself, you define an entity bean with bean-managed persistence. A class of an entity bean with bean-managed persistence is a concrete class (you provide the implementation that is used at run time), whose persistent data is specified as bean-managed persistent fields (see "What are Bean-Managed Persistent Fields?" on page 1-46) for simple data, or as bean-managed relationship fields (see "What are Bean-Managed Relationship Fields?" on page 1-46) for relationships with other entity beans with bean-managed persistence. In this case, you must implement all of the callback methods to manage persistence for your bean's data, including storing and reloading your persistent data to and from the database (see "What is the Life Cycle of an EJB 2.1 Entity Bean With Bean-Managed Persistence?" on page 1-46). When you use bean-managed persistence, you must supply the code that provides the persistence management business logic. In addition, you must provide management for the primary key (see "What is a Primary Key of an Entity Bean With Bean-Managed Persistence?" on page 1-47).

You can specify an entity bean with bean-managed persistence as read-only (see "Configuring a Read-Only Entity Bean With Bean-Managed Persistence" on page 15-4) and take advantage of the optimizations with which OC4J provides read-only entity beans with bean-managed persistence depending on the commit option you choose (see "What are Entity Bean Commit Options?" on page 1-48)

For more information, see the following:

- "Implementing an EJB 2.1 Entity Bean With Bean-Managed Persistence" on page 13-6
- "What is Entity Context?" on page 1-48
- "How do you Avoid Database Resource Contention?" on page 1-59
- "How do you Query for an EJB 2.1 Entity Bean?" on page 1-50
- "When Does Entity Bean Passivation Occur?" on page 1-48
- "What are Entity Bean Commit Options?" on page 1-48

What are Bean-Managed Persistent Fields?

With bean-managed persistence, the code that you write determines which fields of an entity bean with bean-managed persistence are persistent.

What are Bean-Managed Relationship Fields?

With bean-managed persistence, the code that you write implements the relationships between entity beans with bean-managed persistence.

What is the Life Cycle of an EJB 2.1 Entity Bean With Bean-Managed Persistence?

Table 1–14 lists the life cycle methods, as specified in the `javax.ejb.EntityBean` interface, that an entity bean with bean-managed persistence must implement.

For an entity bean with bean-managed persistence, you must provide a complete implementation of all life cycle methods.

Table 1–14 EJB Life Cycle Methods for an Entity Bean With Bean-Managed Persistence

EJB Method	Description
`ejbCreate`	You must implement an `ejbCreate` method corresponding to each `create` method declared in the home interface. When the client invokes the `create` method, the container first invokes the constructor to instantiate the object, then it invokes the corresponding `ejbCreate` method. The `ejbCreate` method performs the following: ■ creates any persistent storage for its data, such as database rows; ■ initializes a unique primary key and returns it.
`ejbPostCreate`	The container invokes this method after the environment is set. For each `ejbCreate` method, an `ejbPostCreate` method must exist with the same arguments. This method can be used to initialize parameters within or from the entity context.
`ejbRemove`	The container invokes this method before it ends the life of the session object. This method can perform any required clean-up (for example, closing external resources such as file handles).
`ejbStore`	The container invokes this method right before a transaction commits. It saves the persistent data to an outside resource, such as a database.
`ejbLoad`	The container invokes this method when the data should be reinitialized from the database. This usually occurs after activation of an entity bean.
`ejbActivate`	The container calls this method directly before it activates an object that was previously passivated. Perform any necessary reaquisition of resources in this method.
`ejbPassivate`	The container calls this method before it passivates the object. Release any resources that can be easily re-created in `ejbActivate`, and save storage space. Typically, you want to release resources that cannot be passivated, such as sockets or database connections. Retrieve these resources in the `ejbActivate` method.

For more information, see the following:

- "What is the Life Cycle of an Enterprise Bean?" on page 1-5
- "Configuring a Life Cycle Callback Method for an EJB 2.1 Entity Bean With Bean-Managed Persistence" on page 15-7

What is a Primary Key of an Entity Bean With Bean-Managed Persistence?

An entity bean primary key is a uniquely identifiable value that distinguishes one instance of a particular type of entity bean class from another. Each entity bean has a persistent identity associated with it. That is, the entity bean contains a unique identity that can be retrieved if you have the primary key: given the primary key, a client can retrieve the entity bean. If the bean is not available, the container instantiates the bean and repopulates the persistent data for you.

The type for the unique key is defined by the bean provider.

All fields within the primary key are restricted to the following:

- primitive object types;
- serializable types;
- types that can be mapped to SQL types;
- types that are a legal Value Type in RMI-IIOP;
- types that provide a suitable implementation of the `hashCode()` and `equals(Object)` methods.

You can define a primary key in one of the following ways (in either case, for an entity bean with bean-managed persistence, you create the primary key in the `ejbCreate` method):

- Define the type of the primary key to be a well-known Java type. The primary key variable that is declared within the bean class must be declared as `public` (see

"Configuring a Primary Key Field for an EJB 2.1 Entity Bean With Bean-Managed Persistence" on page 15-2).

- Define the type of the primary key as a serializable object within a `<name>PK` class that is serializable (see "Configuring a Primary Key Class for an EJB 2.1 Entity Bean With Bean-Managed Persistence" on page 15-2).

What is Entity Context?

OC4J maintains a `javax.ejb.EntityContext` for each EJB 2.1 entity bean with container-managed persistence or entity bean with bean-managed persistence instance and makes this entity context available to the beans. The bean may use the methods in the entity context to make callback requests to the container. In addition, you can use the methods inherited from `EJBContext` (see "What is EJB Context?" on page 1-6).

For more information, see the following:

- "Implementing the setEntityContext and unsetEntityContext Methods" on page 13-20
- "Accessing an EJB 2.1 EJBContext" on page 29-27.

When Does Entity Bean Passivation Occur?

Entity bean passivation applies only to EJB 2.1 entity beans with container-managed persistence.

OC4J passivates an instance when the container decides to disassociate the instance from an entity object identity, and to put the instance back into the pool of available instances. OC4J calls the instance's `ejbPassivate` method to give the instance the chance to release any resources (typically, allocated in the `ejbActivate` method) that should not be held while the instance is in the pool. This method executes with an unspecified transaction context. The entity bean must not attempt to access its persistent state or relationships using the accessor methods during this method.

What are Entity Bean Commit Options?

Commit options determine entity bean instance state at transaction commit time and offer the flexibility to allow OC4J to optimize certain application conditions.

Table 1–15 lists the commit options as defined by the EJB 2.1 specification and indicates which are supported by OC4J.

Table 1–15 OC4J Support for Entity Bean Commit Options

Commit Option	OC4J Support	Description	Instance state written to database?	Instance stays ready	Instance state remains valid	Advantages	Disadvantages
A	✔[1]	Cached bean: At the end of the transaction, the instance stays in the ready state (cached) and the instance state is valid (`ejbLoad` called once on activation).	✔	✔	✔	Least database access.	Exclusive access required. Multiple threads share same bean instance (poor performance).
B		Stale bean: At the end of the transaction, the instance stays in the ready state (cached) but the instance state is not valid: `ejbLoad` and `ejbStore` called for each transaction.	✔	✔		Moderate database access. Allows concurrent requests.	Overhead of multiple bean instances representing the same data. Each transaction calls `ejbLoad`
C	✔[2]	Pooled bean: At the end of the transaction, neither the instance nor its state is valid (instance will be passivated and returned to the pool). Every client call causes an `ejbActivate`, `ejbLoad`, then the business method, then `ejbStore`, and `ejbPassivate`.	✔			Best scalability. Allows concurrent requests. Need not hold on to connections.	Most database access (every business method call). No caching.

[1] Entity beans with bean-managed persistence only (see "Commit Options and BMP Applications" on page 1-50).

[2] Entity beans with container-managed persistence only (see "Commit Options and CMP Applications" on page 1-49).

Commit Options and CMP Applications

For an EJB 2.1 CMP application deployed to OC4J using the TopLink persistence manager, by default, OC4J uses TopLink configuration to approximate commit option C. This option provides the best performance and scalability over the widest range of applications.

OC4J EJB 2.1 CMP conforms to option C in terms of life cycle method calls. However, the TopLink persistence manager introduces the following innovations:

- It provides caching using the TopLink cache.
- It does not synchronize the instance with the data source at the beginning of every transaction if the instance is already in the TopLink cache.

You can use locking or synchronization with a TopLink pessimistic or optimistic locking policy to handle concurrent services to the same bean. This provides the best performance for concurrent access of the same instance while guaranteeing an instance is not updated with stale data.

For more information on making fine-grained TopLink configuration changes, see the following:

- "Customizing the TopLink EJB 2.1 Persistence Manager" on page 3-13
- "Configuring Locking Policy" in the *Oracle TopLink Developer's Guide*

Commit Options and BMP Applications

For an EJB 2.1 BMP application deployed to OC4J, you can configure commit option A or C (see "Configuring Commit Options for an Entity Bean With Bean-Managed Persistence" on page 15-5).

When you configure an entity bean with bean-managed persistence as read-only, OC4J uses a special case of commit option A to improve performance. In this case, OC4J caches the instance and does not update the instance or call `ejbStore` when the transaction commits. For more information, see "Configuring a Read-Only Entity Bean With Bean-Managed Persistence" on page 15-4.

You can use BMP commit option A and read-only entity beans with bean-managed persistence independently (that is, you can configure an entity bean with bean-managed persistence with commit option A without using read-only and you can use read-only without configuring an entity bean with bean-managed persistence with commit option A).

How do you Query for an EJB 2.1 Entity Bean?

To query for an EJB 2.1 entity bean instance, you use a finder or select method (see "Understanding Finder Methods" on page 1-53 and "Understanding Select Methods" on page 1-55).

In either case, you express your selection criteria using an appropriate query syntax (see "Understanding EJB 2.1 Query Syntax" on page 1-50).

For more information, see "Implementing EJB 2.1 Queries" on page 16-1.

Understanding EJB 2.1 Query Syntax

Table 1–16 summarizes the types of query syntax you can use to define EJB queries.

Table 1–16 OC4J EJB 2.1 Query Syntax Support

Query Syntax	See Also
EJB QL	"Understanding EJB 2.1 Query Syntax" on page 1-50
TopLink	"Understanding TopLink Query Syntax" on page 1-51
Predefined Finder	"Predefined TopLink Finders" on page 1-53
Default Finder	"Default TopLink Finders" on page 1-54
Custom Finder	"Custom TopLink Finders" on page 1-55
Custom Select	"Custom TopLink Select Methods" on page 1-56
Native SQL	"Understanding Native SQL Query Syntax in EJB 2.1" on page 1-52

Oracle recommends the use of EJB QL because it is both portable and optimizable.

Understanding EJB QL Query Syntax EJB QL is a specification language used to define semantics of finder and select methods (see "Understanding Finder Methods" on page 1-53 and "Understanding Select Methods" on page 1-55) in a portable and optimizable format. You ensure that an EJB QL statement is associated with each finder and select method.

Although similar to SQL, EJB QL offers significant advantages over native SQL. While SQL applies queries against tables, using column names, EJB QL applies queries against entity beans with container-managed persistence, using the abstract schema name and the container-managed persistent and relationship fields of the bean within the query. The EJB QL statement retains the object terminology. The container

translates the EJB QL statement to the appropriate database SQL statement when the application is deployed. Thus, the container is responsible for converting the entity bean name, container-managed president field names, and container-managed relationship field names to the appropriate database tables and column names. EJB QL is portable to all databases supported by OC4J.

In EJB 2.1, EJB QL is a subset of SQL92, that includes extensions that allow navigation over the relationships defined in an entity bean's abstract schema. The abstract schema is part of an entity bean's deployment descriptor and defines the bean's persistent fields and relationships. The term "abstract" distinguishes this schema from the physical schema of the underlying data store. The abstract schema name is referenced by EJB QL queries since the scope of an EJB QL query spans the abstract schemas of related entity beans that are packaged in the same EJB JAR file.

For an entity bean with container-managed persistence, an EJB QL query must be defined for every finder method (except `findByPrimaryKey`). Using OC4J with the TopLink persistence manager, you can take advantage of predefined and default finder and select methods (see "TopLink Finders" on page 1-53 and "Custom TopLink Select Methods" on page 1-56). The EJB QL query determines the query that is executed by the EJB container when the finder or select method is invoked.

Oracle Application Server provides complete support for EJB QL with the following important features:

- Automatic Code Generation: EJB QL queries are defined in the deployment descriptor of the entity bean. When the enterprise beans are deployed to Oracle Application Server, the container automatically translates the queries into the SQL dialect of the target data store. Because of this translation, entity beans with container-managed persistence are portable: their code is not tied to a specific type of data store.
- Optimized SQL Code Generation: Further, in generating the SQL code, Oracle Application Server makes several optimizations such as the use of bulk SQL, batched statement dispatch, and so on to make database access efficient.
- Support for Oracle and Non-Oracle Databases: Further, Oracle Application Server provides the ability to execute EJB QL against any database such as Oracle, MS SQL-Server, IBM DB/2, Informix, and Sybase.
- CMP with Relationships: Oracle Application Server supports EJB QL for both single entity beans and also with entity beans that have relationships, with support for any type of multiplicity and directionality.

Using EJB 2.1, OC4J provides proprietary EJB QL extensions to support SQRT and date, time, and timestamp options not available in EJB 2.1 (see "OC4J EJB 2.1 EJB QL Extensions" on page 16-8).

Understanding TopLink Query Syntax In this release, because TopLink is the default persistence manager (see "TopLink EJB 2.1 Persistence Manager" on page 3-12), you can express selection criteria for an EJB 2.1 finder or select method using the TopLink query and expressions framework. This EJB QL alternative offers numerous advantages (see "Advantages of TopLink Queries and Expressions" on page 1-52).

You can use the TopLink Workbench to customize your `ejb-jar.xml` file to create advanced finder and select methods using the TopLink query and expression framework.

You also can take advantage of the predefined and default finders and select methods that the TopLink persistence manager provides (see "TopLink Finders" on page 1-53 and "Custom TopLink Select Methods" on page 1-56).

For more information, see the following:

- "Understanding TopLink Queries" in the *Oracle TopLink Developer's Guide*
- "Understanding TopLink Expressions" in the *Oracle TopLink Developer's Guide*.
- "Configuring Named Queries at the Descriptor Level" in the *Oracle TopLink Developer's Guide*
- "Using EJB Finders" in the *Oracle TopLink Developer's Guide*
- "Working with the ejb-jar.xml File" in the *Oracle TopLink Developer's Guide*

Advantages of TopLink Queries and Expressions

Using the TopLink expressions framework, you can specify query search criteria based on your domain object model.

Expressions offer the following advantages over SQL when you access a database:

- Expressions are easier to maintain, because, like EJB QL, the database is abstracted.
- Changes to descriptors or database tables do not affect the querying structures in the application.
- Expressions enhance readability by standardizing the `Query` interface so that it looks similar to traditional Java calling conventions. For example, the Java code required to get the street name from the `Address` object of the `Employee` class looks as follows:

  ```
  emp.getAddress().getStreet().equals("Meadowlands");
  ```

  The expression to get the same information is similar:

  ```
  emp.get("address").get("street").equal("Meadowlands");
  ```

- Expressions allow read queries to transparently query between two classes that share a relationship. If these classes are stored in multiple tables in the database, TopLink automatically generates the appropriate join statements to return information from both tables.
- Expressions simplify complex operations. For example, the following Java code retrieves all employees that live on "Meadowlands" whose salary is greater than 10,000:

  ```
  ExpressionBuilder emp = new ExpressionBuilder();
  Expression exp = emp.get("address").get("street").equal("Meadowlands");
  Vector employees = session.readAllObjects(Employee.class,
    exp.and(emp.get("salary").greaterThan(10000)));
  ```

  TopLink automatically generates the appropriate SQL from that code:

  ```
  SELECT t0.VERSION, t0.ADDR_ID, t0.F_NAME, t0.EMP_ID, t0.L_NAME, t0.MANAGER_ID,
  t0.END_DATE, t0.START_DATE, t0.GENDER, t0.START_TIME, t0.END_TIME,t0.SALARY
  FROM EMPLOYEE t0, ADDRESS t1 WHERE (((t1.STREET = 'Meadowlands')AND (t0.SALARY
  > 10000)) AND (t1.ADDRESS_ID = t0.ADDR_ID))
  ```

Understanding Native SQL Query Syntax in EJB 2.1 In this release, the TopLink persistence manager takes the query syntax you specify ("Understanding EJB QL Query Syntax" on page 1-50 or "Understanding TopLink Query Syntax" on page 1-51) and generates Sequential Query Language (SQL) native to your underlying relational database.

EJB QL is the preferred syntax because it is portable and optimizable.

Native SQL is appropriate for taking advantage of advanced query features of your underlying relational database that EJB QL does not support.

Using EJB 2.1 and the TopLink query syntax, you can use the following:

- default finders that take a native SQL string (see "Default TopLink Finders" on page 1-54);
- custom finder or select methods that use native SQL calls (see "TopLink Finders" on page 1-53 and "Custom TopLink Select Methods" on page 1-56).

To use native SQL otherwise, you must use straight JDBC calls.

Understanding Finder Methods

A finder method is an EJB method the name of which begins with `find` that you define in the `Home` interface of an EJB (see or "Implementing the EJB 2.1 Home Interfaces" on page 13-18) and associate with a query to return one or more instances of that EJB type. At deployment time, OC4J provides an implementation of this method that executes the associated query.

Finder methods are the means by which clients retrieve EJB 2.1 entity beans with container-managed persistence. Using EJB 2.1, you can do the following:

- Expose any of the predefined and default finders that OC4J and the TopLink persistence manager provide to all entity beans with container-managed persistence (see "Predefined TopLink Finders" on page 1-53 and "Default TopLink Finders" on page 1-54).
- Define custom EJB QL finders (see "Implementing an EJB 2.1 EJB QL Finder Method" on page 16-1) and custom TopLink finders (see "Custom TopLink Finders" on page 1-55).

A finder that returns a single EJB instance has a return type of that EJB instance.

A finder that returns more than one EJB instance has a return type of `Collection`. If no matches are found, an empty `Collection` is returned. To ensure that no duplicates are returned, specify the `DISTINCT` keyword in the associated EJB query.

All finders throw `FinderException`.

At the very least, you must expose the `findByPrimaryKey` finder method to retrieve a reference for each entity bean using its primary key.

TopLink Finders The TopLink persistence manager provides OC4J entity beans with a variety of predefined (see "Predefined TopLink Finders" on page 1-53) and default (see "Default TopLink Finders" on page 1-54) finders. You can expose these finders to your clients as you would for any other finder. You do not need to specify a corresponding query. You can also create custom TopLink finders (see "Custom TopLink Finders" on page 1-55).

Predefined TopLink Finders

Table 1–17 lists the predefined finders you can expose for EJB 2.1 entity beans with container-managed persistence. The TopLink persistence manager reserves the method names listed in Table 1–17.

Table 1–17 Predefined TopLink CMP Finders

Method	Arguments	Return
`findAll`	`()`	`Collection`
`findManyByEJBQL`	`(String ejbql)` `(String ejbql, Vector args)`	`Collection`
`findManyByQuery`	`(DatabaseQuery query)` `(DatabaseQuery query, Vector args)`	`Collection`
`findManyBySQL`	`(String sql)` `(String sql, Vector args)`	`Collection`
`findByPrimaryKey`	`(Object primaryKeyObject)`	`EJBObject` or `EJBLocalObject`[1]
`findOneByEJBQL`	`(String ejbql)`	Component interface
`findOneByEJBQL`	`(String ejbql, Vector args)`	`EJBObject` or `EJBLocalObject`[1]
`findOneByQuery`	`(DatabaseQuery query)`	Component interface
`findOneByQuery`	`(DatabaseQuery query, Vector args)`	`EJBObject` or `EJBLocalObject`[1]
`findOneBySQL`	`(String sql)`	Component interface
`findOneBySQL`	`(String sql, Vector args)`	`EJBObject` or `EJBLocalObject`[1]

[1] Depending on whether or not the finder is defined in the home or component interface.

Example 1–4 shows an `EJBHome` that defines two predefined finders (`findByPrimaryKey` and `findManyBySQL`). TopLink will provide the query implementation for these finders.

Example 1–3 Specifying Predefined TopLink Finders

```
public interface EmpBeanHome extends EJBHome {
    public EmpBean create(Integer empNo, String empName) throws CreateException;

    /**
    * Finder methods. These are implemented by the container. You can
    * customize the functionality of these methods in the deployment
    * descriptor through EJB-QL.
    **/

    // Predefined Finders: <query> element in ejb-jar.xml not required

    public Topic findByPrimaryKey(Integer key) throws FinderException;
    public Collection findManyBySQL(String sql, Vector args) throws FinderException

}
```

Default TopLink Finders

For each finder method defined in the home interface of an entity bean, whose name matches `findBy<`*`CMP-FIELD-NAME`*`>` where <*`CMP-FIELD-NAME`*> is the name of a persistent field on the bean, TopLink generates a finder implementation including a TopLink query that uses the TopLink expressions framework. If the return type is a single bean type, TopLink creates a `oracle.toplink.queryframework.ReadObjectQuery`; if the return type is `Collection`, TopLink creates a `oracle.toplink.queryframework.ReadAllQuery`. You can expose these finders to your clients as you would for any other finder. You do not need to specify a corresponding query.

Example 1–4 shows an `EJBHome` that defines a default finder (`findByEmpNo`). TopLink will provide the query implementation for this finder.

Example 1–4 Specifying Default TopLink Finders

```
public interface EmpBeanHome extends EJBHome {
    public EmpBean create(Integer empNo, String empName) throws CreateException;

    /**
    * Finder methods. These are implemented by the container. You can
    * customize the functionality of these methods in the deployment
    * descriptor through EJB-QL.
    **/

    // Default Finder: <query> element in ejb-jar.xml not required

    public Topic findByEmpNo(Integer empNo);
}
```

Custom TopLink Finders

You can take advantage of the TopLink query and expression framework to define advanced finders, including `Call`, `DatabaseQuery`, primary key, `Expression`, EJB QL, native SQL, and redirect finders (that delegate execution to the implementation that you define as a static method on an arbitrary helper class).

Using EJB 2.1, to create custom TopLink finders, use your existing `toplink-ejb-jar.xml` file with the TopLink Workbench (see "Using TopLink Workbench" on page 16-4).

Understanding Select Methods

An entity bean select method is a query method intended for internal use within an EJB 2.1 entity bean with container-managed persistence instance. You define a select method as an abstract method of the abstract entity bean class itself and associate an EJB QL query with it. You do not expose the select method to the client in the home or component interface. You may define zero or more select methods. The container is responsible for providing the implementation of the select method based on the EJB QL query you associate with it.

You typically call a select method within a business method to retrieve the value of a container-managed persistent field or entity bean references of container-managed relationship fields. A select method executes in the transaction context determined by the transaction attribute of the invoking business method.

A select method has the following signature:

```
public abstract <ReturnType> ejbSelect<METHOD>(...) throws FinderException
```

- It must be declared as `public` and `abstract`.
- The return type must conform to the select method return type rules (see "What Type Can Your Select Method Return?" on page 1-56).
- The method name must start with `ejbSelect`.
- The method must throw `javax.ejb.FinderException` and may also throw other application-specific exceptions as well.

Although the select method is not based on the identity of the entity bean instance on which it is invoked, it can use the primary key of an entity bean as an argument. This creates a query that is logically scoped to a particular entity bean instance.

Using EJB 2.1, you can define custom EJB QL select methods (see "Implementing an EJB 2.1 EJB QL Select Method" on page 16-5) and you can define custom TopLink select methods (see "Custom TopLink Select Methods" on page 1-56).

What Type Can Your Select Method Return? The select method return type is not restricted to the entity bean type on which the select is invoked. Instead, it can return any type corresponding to a container-managed persistent or container-managed relationship field.

Your select method must conform to the following return type rules:

- All values must be returned as `Object`; any primitive types are wrapped in their corresponding `Object` types (for example, a primitive `int` is wrapped in an `Integer` object).
- Single object: If your select method returns only a single item, the container returns the same type as specified in your select method signature.

  If multiple objects are returned, a `FinderException` is raised.

  If no objects are found, a `FinderException` is raised
- Multiple objects: If your select method returns multiple items, you must define the return type as a `Collection`.

  Choose the Collection type to suit your needs. For example, a `Collection` may include duplicates, a `Set` eliminates duplicates, and a `SortedSet` will return an ordered `Collection`.

  If no objects are found, an empty `Collection` is returned.

  – Container-managed persistent values: If you return multiple container-managed persistent values, the container returns a `Collection` of objects whose type it determines from the EJB QL select statement.

  – Container-managed relationship values: If you return multiple container-managed relationship values, then, by default, the container returns a `Collection` of objects whose type is the local bean interface type.

    You can change this to the remote bean interface with annotations or deployment XML configuration. For more information, see "Implementing an EJB 2.1 EJB QL Select Method" on page 16-5.

Custom TopLink Select Methods Using EJB 2.1, you can create custom TopLink select methods.

Using EJB 2.1, you can utilize the TopLink query and expression framework to define advanced select methods that can use any of the TopLink query and expression framework features, including `Call`, `DatabaseQuery`, `Expression`, EJB QL, and native SQL. For more information, see "Using TopLink Workbench" on page 16-7.

What is a Message-Driven Bean?

A message-driven bean (MDB) is an EJB 3.0 or EJB 2.1 enterprise bean component that functions as an asynchronous message consumer. An MDB has no client-specific state but may contain message-handling state such as an open database connection or object references to another EJB. A client uses an MDB to send messages to the destination for which the bean is a message listener.

Using OC4J, you can use an MDB with a variety of message providers (see "What Message Service Providers Can you use With Your MDB?" on page 2-21). You associate the MDB with an existing message provider and the container handles much of the setup required, as follows:

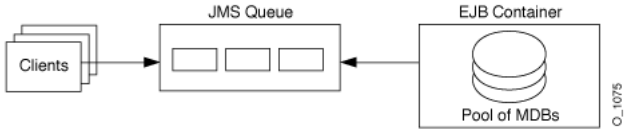
- The EJB container creates a consumer of type `QueueReceiver` or `TopicSubscriber` for the listener.

- At deployment time, the EJB container registers the MDB with the consumer, which is either a `QueueReceiver` or `TopicSubscriber`, and its factory.
- The EJB container specifies the message acknowledgment mode.
- The EJB container dequeues messages and passes them to the MDB using its message listener method.
- The EJB container sends an acknowledgment (if configured to do so).

The purpose of an MDB is to exist within a pool and to receive and process incoming messages from a message provider. The container invokes a bean from the queue to handle each incoming message from the queue. No object invokes an MDB directly: all invocation for an MDB comes from the container. After the container invokes the MDB, it can invoke other enterprise beans or Java objects to continue the request.

A MDB is similar to a stateless session bean, because it does not save conversational state and is used for handling multiple incoming requests. Instead of handling direct requests from a client, MDBs handle requests placed on a queue. Figure 1–7 demonstrates this by showing how clients place requests on a queue. The container takes the requests off of the queue and gives the request to an MDB in its pool.

Figure 1–7 Message Driven Beans



This section describes the following:

- What is the Life Cycle of a Message-Driven Bean?
- What is Message Driven Context?

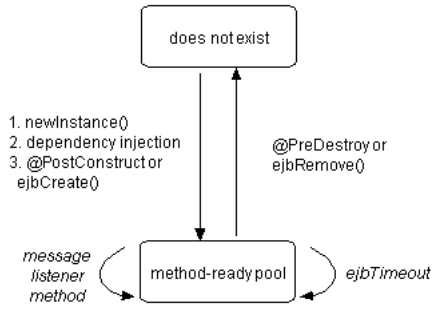
For more information, see the following:

- "Implementing an EJB 3.0 Message-Driven Bean" on page 9-1
- "Implementing an EJB 2.1 Message-Driven Bean" on page 17-1

What is the Life Cycle of a Message-Driven Bean?

Figure 1–8 shows the life cycle of a message-driven bean. Annotations (such as `@PostConstruct`) are applicable to EJB 3.0 message-driven beans only.

Figure 1–8 Life Cycle of an EJB 2.1 MDB



The life cycle for EJB 3.0 (see Table 1–18) and EBJ 2.1 (see Table 1–19) message-driven beans are identical. The difference is in how you register life cycle callback methods.

Table 1–18 lists the optional EJB 3.0 message-driven bean life cycle callback methods you can define using annotations. For EJB 3.0 message-driven beans, you do not need to implement these methods.

Table 1–18 Life Cycle Methods for an EJB 3.0 Message-Driven Bean

Annotation	Description
`@PostConstruct`	This optional method is invoked for a message-driven bean before the first business method invocation on the bean. This is at a point after which any dependency injection has been performed by the container.
`@PreDestroy`	This optional method is invoked for a message-driven bean when the instance is in the process of being removed by the container. The instance typically releases any resources that it has been holding.

Table 1–19 lists the EJB 2.1 life cycle methods, as specified in the `javax.ejb.MessageDrivenBean` interface, that a message-driven bean must implement. For EJB 2.1 message-driven beans, you must at the least provide an empty implementation for all callback methods.

Table 1–19 Life Cycle Methods for an EJB 2.1 Message-Driven Bean

EJB Method	Description
`ejbCreate`	The container invokes this method right before it creates the bean. A message-driven bean must do nothing in this method.
`ejbRemove`	A container invokes this method before it ends the life of a MDB. Use this method to perform any required clean-up (for example, closing external resources such as file handles).

For more information, see the following:

- "What is the Life Cycle of an Enterprise Bean?" on page 1-5
- "Configuring a Life Cycle Callback Interceptor Method on an EJB 3.0 MDB" on page 10-11
- "Configuring a Life Cycle Callback Interceptor Method on an Interceptor Class of an EJB 3.0 MDB" on page 10-11
- "Configuring a Life Cycle Callback Method for an EJB 2.1 MDB" on page 18-10

What is Message Driven Context?

OC4J maintains a `javax.ejb.MessageDrivenContext` for each message-driven bean instance and makes this message-driven context available to the beans. The bean may use the methods in the message-driven context to make callback requests to the container.

In addition, you can use the methods inherited from `EJBContext` (see "What is EJB Context?" on page 1-6).

For more information, see the following:

- "Accessing an EJB 3.0 EJBContext" on page 29-20
- "Accessing an EJB 2.1 EJBContext" on page 29-27

Which Type of Enterprise Bean Should You Use?

This section describes the following:

- Which Type of Session Bean Should You Use?
- When do you use Bean-Managed Versus Container-Managed Persistence?

Which Type of Session Bean Should You Use?

Stateless session beans are useful mainly in middle-tier application servers that provide a pool of beans to process frequent and brief requests.

When do you use Bean-Managed Versus Container-Managed Persistence?

In practical terms, Table 1–20 provides a definition for both BMP and CMP, and a summary of the programmatic and declarative differences between them.

Table 1–20 Comparison of Bean-Managed and Container-Managed Persistence

Management Issues	Bean-Managed Persistence	Container-Managed Persistence
Persistence management	You are required to implement the persistence management within the `ejbStore`, `ejbLoad`, `ejbCreate`, and `ejbRemove EntityBean` methods. These methods must contain logic for saving and restoring the persistent data. For example, the `ejbStore` method must have logic in it to store the entity bean's data to the appropriate database. If it does not, the data can be lost.	The management of the persistent data is done for you. That is, the container invokes a persistence manager on behalf of your bean. You use `ejbStore` and `ejbLoad` for preparing the data before the commit or for manipulating the data after it is refreshed from the database. The container always invokes the `ejbStore` method right before the commit. In addition, it always invokes the `ejbLoad` method right after reinstating CMP data from the database.
Finder methods allowed	The `findByPrimaryKey` method and other finder methods are allowed.	The `findByPrimaryKey` method and other finder methods clause are allowed.
Defining container-managed persistent fields	N/A	Required within the EJB deployment descriptor. The primary key must also be declared as a container-managed persistent field.
Mapping container-managed persistent fields to resource destination	N/A	Required. Dependent on persistence manager.
Definition of persistence manager	N/A	Required within the Oracle-specific deployment descriptor. By default,OC4J uses the TopLink persistence manager.

With CMP, you can build components to the EJB 2.0 specification that can save the state of your EJB to any Java EE supporting application server and database without having to create your own low-level JDBC-based persistence system.

With BMP, you can tailor the persistence layer of your application at the expense of additional coding and support effort.

For more information, see the following:

- "What is an EJB 2.1 Entity Bean With Container-Managed Persistence?" on page 1-42
- "What is an EJB 2.1 Entity Bean With Bean-Managed Persistence?" on page 1-46

How do you Avoid Database Resource Contention?

OC4J and the TopLink EJB 3.0 JPA persistence provider and EJB 2.1 persistence manager use a combination of transaction isolation (see "Transaction Isolation" on page 1-60) and concurrency mode (see "Concurrency (Locking) Mode" on page 1-60) to avoid database resource contention and to permit concurrent access to database tables.

Transaction Isolation

The degree to which concurrent (parallel) transactions on the same data are allowed to interact is determined by the level of *transaction isolation* configured. ANSI/SQL defines four levels of database transaction isolation as shown in Table 1–21. Each offers a trade-off between performance and resistance from the following unwanted actions:

- Dirty read: a transaction reads uncommitted data written by a concurrent transaction.
- Nonrepeatable read: a transaction rereads data and finds it has been modified by some other transaction that was committed after the initial read operation.
- Phantom read: a transaction re executes a query and the returned data has changed due to some other transaction that was committed after the initial read operation.

Table 1–21 Transaction Isolation Levels

Transaction Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read Uncommitted	Yes	Yes	Yes
Read Committed	No	Yes	Yes
Repeatable Read	No	No	Yes
Serializable	No	No	No

By default, OC4J and the TopLink EJB 3.0 JPA persistence provider and EJB 2.1 persistence manager provide read-committed transaction isolation.

To configure the transaction isolation mode, you must customize the TopLink EJB 3.0 JPA persistence provider or EJB 2.1 persistence manager.

For more information, see the following:

- "Customizing the JPA Persistence Provider" on page 3-3
- "Customizing the TopLink EJB 2.1 Persistence Manager" on page 3-13
- "Unit of Work Transaction Isolation" in the *Oracle TopLink Developer's Guide*
- "Database Transaction Isolation Levels" in the *Oracle TopLink Developer's Guide*

Concurrency (Locking) Mode

OC4J also provides concurrency modes for handling resource contention and parallel execution within EJB 3.0 entities and EJB 2.1 entity beans with container-managed persistence.

Entity beans with bean-managed persistence manage the resource locking within the bean implementation themselves.

Concurrency modes include the following:

- Optimistic Locking: Multiple users have read access to the data. When a user attempts to make a change, the application checks a version field (also known as a write-lock field) to ensure the data has not changed since the user read the data.

  When optimistic locking is enabled, TopLink caches the value of this version field as it reads an object from the data source. When the client attempts to write the object, TopLink compares the cached version value with the current version value in the data source in the following way:

    - If the values are the same, TopLink updates the version field in the object and commits the changes to the data source.
    - If the values are different, the write operation is disallowed because another client must have updated the object since this client initially read it.
- Pessimistic Locking: The first user, who accesses the data with the purpose of updating it, locks the data until completing the update. This manages resource contention and does not allow parallel execution. Only one user at a time is allowed to execute the entity bean at a single time.
- Read-only: Multiple users can execute the entity bean in parallel. The container does not allow any updates to the bean's state.

These concurrency modes are defined for each bean and apply on the transaction boundaries.

By default, in EJB 3.0, the JPA persistence manager assumes that the application is responsible for data consistency. Oracle recommends that you use the `@Version` annotation to specify a version field and enable JPA-managed optimistic locking.

By default, in EJB 2.1, the TopLink persistence manager enforces optimistic locking by using a code-generated numeric version field that TopLink updates each time an object change is committed.

To configure the concurrency mode otherwise, you must customize the TopLink EJB 3.0 JPA persistence provider or EJB 2.1 persistence manager.

For more information, see the following:

- "Customizing the JPA Persistence Provider" on page 3-3
- "Customizing the TopLink EJB 2.1 Persistence Manager" on page 3-13
- "Locking" in the *Oracle TopLink Developer's Guide*
- "Configuring Locking Policy" in the *Oracle TopLink Developer's Guide*
- "Configuring Read-Only Descriptors" in the *Oracle TopLink Developer's Guide*

2

Understanding EJB Application Development

This chapter describes the following:

- Using EJB Development Tools
- What OC4J Services Can You Use With an EJB?
- How do you Package and Deploy an EJB Application?
- How do you use an Enterprise Bean in Your Application?

Using EJB Development Tools

This section describes developing EJB applications using the following:

- Using JDeveloper
- Using Eclipse
- Using TopLink Workbench

Using JDeveloper

Oracle JDeveloper greatly simplifies Java EE application development, packaging, and deployment by providing extensive automation, a built-in OC4J for rapid deployment and testing, and many other productivity enhancements. For example:

- Developing session beans:
  `http://www.oracle.com/technology/products/jdev/101/viewlets/101/ejb30sessionbeanviewlet_viewlet_swf.htm`
- Developing entity beans:
  `http://www.oracle.com/technology/products/jdev/101/viewlets/101/ejb30entitybeanviewlet_viewlet_swf.htm`

For more information on JDeveloper, see
`http://www.oracle.com/technology/products/jdev/index.html`.

Using Eclipse

Eclipse is a widely adopted integrated development environment that simplifies Java EE application development, packaging, and deployment.

Oracle is developing extensible frameworks and exemplary tools on the Eclipse platform for the definition and editing of Object-Relational (O/R) mappings for EJB 3.0 entities. EJB 3.0 O/R mapping support will focus on minimizing the complexity of

mapping by providing creation and automated initial mapping wizards, and programming assistance such as dynamic problem identification

For more information on EJB 3.0 support in Eclipse, see `http://www.eclipse.org/dali/`.

Using TopLink Workbench

You can use the TopLink Workbench to create and configure the following:

- EJB 3.0 `toplink-ejb-jar.xml` and `ejb3-toplink-sessions.xml` files
- EJB 2.1 `toplink-ejb-jar.xml` file
- `ejb-jar.xml` file

For more information, see the following:

- "Understanding the TopLink Workbench" in the *Oracle TopLink Developer's Guide*
- "Understanding EJB Deployment Descriptor Files" on page 2-4

What OC4J Services Can You Use With an EJB?

Table 2–1 lists some of the important services that OC4J provides and shows the EJB types with which you can use them.

Table 2–1 OC4J Services and EJB Support

OC4J Service	Stateful Session Bean	Stateless Session Bean	EJB 3.0 Entities	CMP Entity Bean	BMP Entity Bean	Message-Driven Bean
"Understanding EJB Persistence Services" on page 2-12			✔	✔	✔	
"Understanding EJB JNDI Services" on page 2-14	✔	✔	✔	✔	✔	✔
"Understanding EJB Data Source Services" on page 2-14	✔	✔	✔	✔	✔	✔
"Understanding EJB Transaction Services" on page 2-17	✔	✔	✔	✔	✔	✔
"Understanding EJB Security Services" on page 2-20	✔	✔	✔	✔	✔	✔
"Understanding Message Services" on page 2-20						✔
"Understanding OC4J EJB Application Clustering Services" on page 2-29	✔					
"Understanding EJB Timer Services" on page 2-31		✔		✔	✔	✔

For more information on OC4J services, see the appropriate OC4J guide as shown in Table 2–2:

Table 2–2 Location of Information for Java EE Subjects

Java EE Subject	The Subject is Documented in this OC4J Book
Optimization	*Oracle Application Server Performance Guide*
Web Services	*Oracle Application Server Web Services Developer's Guide*
Security	*Oracle Containers for J2EE Security Guide*
JNDI	*Oracle Containers for J2EE Services Guide*
Data Source	*Oracle Containers for J2EE Services Guide*
RMI and RMI/IIOP	*Oracle Containers for J2EE Services Guide*

Table 2–2 (Cont.) Location of Information for Java EE Subjects

Java EE Subject	The Subject is Documented in this OC4J Book
CSiV2	*Oracle Containers for J2EE Services Guide*
JMS	*Oracle Containers for J2EE Services Guide*
Clustering	*Oracle Containers for J2EE Services Guide*
Timers	*Oracle Containers for J2EE Services Guide*
J2EE Connector Architecture (J2CA)	*Oracle Containers for J2EE Services Guide*
Java Object Cache	*Oracle Containers for J2EE Services Guide*
HTTPS	*Oracle Containers for J2EE Services Guide*
Transactions (JTA)	*Oracle Containers for J2EE Services Guide*
Default Persistence	*Oracle TopLink Developer's Guide*

How do you Package and Deploy an EJB Application?

This section describes the following:

- Understanding Packaging
- Understanding Deployment
- Understanding EJB Deployment Descriptor Files

Understanding Packaging

The Java EE architecture provides a variety of ways to package (or assemble) your application and its various Java EE components.

The most efficient way to package a Java EE application is to use a Java EE tool such as JDeveloper or Eclipse.

For more information, see the following:

- "Using EJB Development Tools" on page 2-1
- "Packaging an EJB Application" on page 27-1
- *Oracle Application Server Enterprise Deployment Guide*

Understanding Deployment

After you package your Java EE application, to execute the application and make it available to end users, you deploy it to OC4J.

The most efficient way to deploy a Java EE application to OC4J is to use Oracle Enterprise Manager 10*g* Application Server Control.

For more information, see the following:

- "In What Order Does OC4J Deploy EJB Modules?" on page 2-4
- "Understanding EJB Deployment Descriptor Files" on page 2-4
- "Using Oracle Enterprise Manager 10g Application Server Control" on page 31-1
- "Deploying an EJB Application to OC4J" on page 28-1
- *Oracle Application Server Enterprise Deployment Guide*

In What Order Does OC4J Deploy EJB Modules?

OC4J deploys EJB modules in the order in which they appear in the `application.xml` deployment descriptor. In general, loading order is component-specific and based on natural ordering for each component type.

For example, consider the `application.xml` file shown in Example 2–1.

Example 2–1 application.xml

```
<application>
  <display-name>master-application</display-name>
  <module>
    <ejb>ejb1.jar</ejb>
  </module>
  <module>
    <ejb>ejb2.jar</ejb>
  </module>
  <module>
    <java>appclient.jar</java>
  </module>
  <module>
    <web>
        <web-uri>clientweb.war</web-uri>
        <context-root>webapp</context-root>
    </web>
  </module>
  <module>
    <ejb>ejb3.jar</ejb>
  </module>
```

Based on this `application.xml` file, OC4J will load components in the following order:

1. `ejb1`
2. `ejb2`
3. `ejb3`
4. `clientweb.war`
5. `appclient.jar`

Understanding EJB Deployment Descriptor Files

This section describes the various EJB deployment descriptor files that you use in EJB applications deployed to OC4J.

Table 2–3 lists the various EJB deployment descriptor files that you use in EJB applications deployed to OC4J. For each deployment descriptor file, it indicates the EJB types to which the deployment descriptor applies and whether or not the deployment descriptor is optional, required, or not applicable to the EJB specification you are using.

Table 2–3 OC4J EJB Deployment Descriptor Files

Deployment Descriptor File	Session Bean	JPA Entity	EJB 2.1 Entity Bean	Message-Driven Bean	EJB 3.0	EJB 2.1
"What is the ejb-jar.xml File?" on page 2-5	✔		✔	✔	Optional	Required
"What is the orion-ejb-jar.xml File?" on page 2-6	✔	✔[1]	✔	✔	Optional	Optional
"What is the toplink-ejb-jar.xml File?" on page 2-6		✔	✔		Optional	Required

Table 2–3 (Cont.) OC4J EJB Deployment Descriptor Files

Deployment Descriptor File	Session Bean	JPA Entity	EJB 2.1 Entity Bean	Message-Driven Bean	EJB 3.0	EJB 2.1
"What is the ejb3-toplink-sessions.xml File?" on page 2-7		✔			Optional	Not Applicable
"What is the persistence.xml File?" on page 2-8		✔			Optional	Not Applicable
"What is the orm.xml File?" on page 2-9		✔			Optional	Not Applicable

[1] `<entity-deployment>` element `disable-default-persistent-unit` attribute only.

What is the ejb-jar.xml File?

The `ejb-jar.xml` file is an EJB deployment descriptor file, and, when used, it describes the following:

- mandatory structural information about all included enterprise beans;
- a descriptor for container managed relationships, if any;
- an optional name of an ejb-client-jar file for the ejb-jar;
- an optional application-assembly descriptor.

When it is required, the `ejb-jar.xml` file describes EJB information applicable to any Java EE application server. This information may be augmented by application server-specific EJB deployment descriptor files (see "What is the orion-ejb-jar.xml File?" on page 2-6 and "What is the toplink-ejb-jar.xml File?" on page 2-6).

For more information, see "Configuring the ejb-jar.xml File" on page 26-1.

EJB 3.0

If you are using EJB 3.0, this deployment descriptor file is optional: you can use annotations instead. In this release, OC4J supports the use of both EJB 3.0 annotations and `ejb-jar.xml` for all options of session and message-driven beans. The `ejb-jar.xml` file is not used for EJB 3.0 entities. Configuration in the `ejb-jar.xml` file overrides annotations (see "Overriding Annotations With Deployment Descriptor Entries" on page 1-20).

For EJB 30. entities, you must either use annotations or TopLink JPA persistence provider deployment XML files (`toplink-ejb-jar.xml` and `ejb3-toplink-sessions.xml`).

For more information, see:

- "What is the toplink-ejb-jar.xml File?"
- "What is the ejb3-toplink-sessions.xml File?"

EJB 2.1

If you are using EJB 2.1, this deployment descriptor file is required.

XML Reference

The XML reference for this deployment descriptor file depends on the EJB version you are using.

For EJB 3.0, this deployment descriptor file conforms to the XML schema document located at `http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd`.

For EJB 2.1, this deployment descriptor file conforms to the XML schema document located at `http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd`.

What is the orion-ejb-jar.xml File?

The `orion-ejb-jar.xml` file is an EJB deployment descriptor file that contains all OC4J-proprietary options. This file extends the configuration that you specify in the `ejb-jar.xml` file (see "What is the ejb-jar.xml File?" on page 2-5).

For more information, see the following:

- "Configuring the orion-ejb-jar.xml File" on page 26-3
- "XML Reference for orion-ejb-jar.xml Elements" on page A-1

EJB 3.0

If you are using EJB 3.0, this file is optional. You can deploy without an `orion-ejb-jar.xml` file and set OC4J-proprietary options using OC4J-proprietary annotations (such as `@StatelessDeployment`, `@StatefulDeployment`, and `@MessageDrivenDeployment`) or Application Server Control. Vendor extensions set in the `orion-ejb-jar.xml` file override extensions set using OC4J-proprietary annotations. Configuration in the `orion-ejb-jar.xml` file overrides annotations (see "Overriding Annotations With Deployment Descriptor Entries" on page 1-20).

For more information, see the following:

- "Configuring OC4J-Proprietary Deployment Options on an EJB 3.0 Session Bean" on page 5-10
- "Configuring OC4J-Proprietary Deployment Options on an EJB 3.0 MDB" on page 10-17

EJB 2.1

If you are using EJB 2.1, `orion-ejb-jar.xml` file is mandatory for all OC4J-proprietary options.

For more information, see "Customizing the TopLink EJB 2.1 Persistence Manager" on page 3-13.

XML Reference

This deployment descriptor file conforms to the XML schema document at `http://www.oracle.com/technology/oracleas/schema/index.html`.

What is the toplink-ejb-jar.xml File?

The `toplink-ejb-jar.xml` file (also known as the TopLink `project.xml` file) is a TopLink JPA preview persistence configuration descriptor file, and, when used, it describes TopLink project-level options (see "Configuring a Relational Project" in the *Oracle TopLink Developer's Guide*) such as TopLink descriptors and mappings.

Note: By default, OC4J uses the TopLink Essentials JPA persistence provider. In this case, you can configure TopLink descriptor-level options (including mappings) using TopLink JPA extensions ("Accessing TopLink API at Run Time With TopLink Essentials JPA Persistence" on page 3-4).

For more information, see "Configuring the toplink-ejb-jar.xml File" on page 26-2.

EJB 3.0

If you are using EJB 3.0 with the default TopLink Essentials JPA persistence provider, this file is not used.

If you are using EJB 3.0, the `toplink-ejb-jar.xml` file is only used to customize TopLink JPA preview persistence provider configuration (see "Customizing the JPA Persistence Provider" on page 3-3). If you use this file to customize the TopLink persistence provider, you must also use an `ejb3-toplink-sessions.xml` file (see "What is the ejb3-toplink-sessions.xml File?" on page 2-7).

EJB 2.1

If you are using EJB 2.1, the `toplink-ejb-jar.xml` file is optional. If you omit this file from your application, you can configure OC4J to automatically construct it for you (see "Configuring Default Relationship Generation" on page 14-6). Alternatively, you can use this file to configure TopLink persistence options yourself (see "Customizing the TopLink EJB 2.1 Persistence Manager" on page 3-13).

XML Reference

The `toplink-ejb-jar.xml` file conforms to the XML schema documents located at `<OC4J_HOME>\toplink\config\xsds`. Oracle does not recommend manual configuration of this file. To create and configure this file, use the TopLink Workbench (see "Understanding the TopLink Workbench" in the *Oracle TopLink Developer's Guide*).

What is the ejb3-toplink-sessions.xml File?

The `ejb3-toplink-sessions.xml` file is a TopLink JPA preview persistence configuration descriptor file, and, when used with the TopLink JPA preview persistence provider, it describes TopLink session-level options (see "Configuring Server Sessions" in the *Oracle TopLink Developer's Guide*) such as data sources, login information, caching options, and logging. It is equivalent to the `sessions.xml` file that TopLink users are familiar with.

> **Note:** By default, OC4J uses the TopLink Essentials JPA persistence provider. In this case, you can configure TopLink session-level options using TopLink JPA extensions ("Accessing TopLink API at Run Time With TopLink Essentials JPA Persistence" on page 3-4).

This file provides a reference to the primary project (see "What is the toplink-ejb-jar.xml File?" on page 2-6), if used.

For more information, see "Configuring the ejb3-toplink-sessions.xml File" on page 26-3.

EJB 3.0

If you are using EJB 3.0 with the default TopLink Essentials JPA persistence provider, this file is not used.

If you are using EJB 3.0, the `ejb3-toplink-sessions.xml` file is only used to customize TopLink JPA preview persistence provider configuration (see "Customizing the JPA Persistence Provider" on page 3-3). If you use this file to customize the

TopLink JPA preview persistence provider, you may also use a `toplink-ejb-jar.xml` file (see "What is the toplink-ejb-jar.xml File?" on page 2-6).

EJB 2.1

If you are using EJB 2.1, the `ejb3-toplink-sessions.xml` file is not used.

XML Reference

The `ejb3-toplink-sessions.xml` file conforms to the XML schema documents located at `<OC4J_HOME>\toplink\config\xsds`. Oracle does not recommend manual configuration of this file. To create and configure this file, use the TopLink Workbench (see "Understanding the TopLink Workbench" in the *Oracle TopLink Developer's Guide*).

What is the persistence.xml File?

The `persistence.xml` file is a persistence descriptor file that you use to define one or more persistence units in an EJB 3.0 application that uses entities.

In this release, you can define persistence.xml in an EJB JAR, WAR, or EAR.

A persistence unit defines an entity manager's configuration. You specify a persistence unit by name when you acquire an entity manager (see "Acquiring an EntityManager" on page 29-8). Alternatively, you can take advantage of the OC4J default persistence unit (see "Understanding OC4J Persistence Unit Defaults" on page 2-8).

A persistence unit is a logical grouping of the following:

- Entity manager: including, entity manager provider, the entity managers obtained from it, and entity manager configuration.
- Data source (see "Specifying a Data Source in a Persistence Unit" on page 26-5).
- Vendor extensions (see "Configuring Vendor Extensions in a Persistence Unit" on page 26-5).
- Persistent managed classes: the classes you intend to manage using an entity manager, namely, entity classes, embeddable classes, and mapped superclasses (see "What Persistent Managed Classes Does This Persistence Unit Include?" on page 26-4).

  All persistent managed classes in a given persistence unit must be collocated in their mapping to a single database.
- Mapping metadata: the information that describes how to map persistent managed classes to database tables. You can specify mapping metadata using annotations on persistent managed classes and `orm.xml` files (see "What is the orm.xml File?" on page 2-9).

For more information, see the following:

- "Configuring the persistence.xml File" on page 26-3
- "Packaging a JPA Entity Application" on page 27-1
- EJB 3.0 specification

Understanding OC4J Persistence Unit Defaults

To simplify persistence unit configuration, you can use the following OC4J features:

- Smart Defaulting

- Acquiring an Entity Manager by Default Persistence Unit Name

For more information, see the following:

- "Configuring the persistence.xml File for the OC4J Default Persistence Unit" on page 26-5
- "Acquiring an EntityManager" on page 29-8

Smart Defaulting

For EJB modules only, you can rely on OC4J to build a default `persistence.xml` file and configure it with appropriate default values to define a default persistence unit with a default name if:

- You deploy an application without a `persistence.xml` and your application contains at least one class annotated with `@Entity`.
- You deploy an application with an empty `persistence.xml`.

Acquiring an Entity Manager by Default Persistence Unit Name

If your application specifies one and only one persistence unit (either explicitly or by way of smart defaulting), you need not specify the persistence unit name when you acquire an entity manager. In this case, OC4J defaults the persistence unit name.

To disable this feature, set `orion-ejb-jar.xml` file attribute `disable-default-persistent-unit` to `true`.

If you disable this feature, you can still use the OC4J default persistence unit if you specify an empty persistence unit in a `persistence.xml` file, then, when you acquire an entity manager in that persistence unit's scope, you do not need to specify a persistence unit name. In this case, OC4J will use its own default persistence unit and will assume that all JPA entity classes in the persistence unit root belong to that persistence unit. You may specify one and only one such empty persistence unit in your application.

EJB 3.0

If you are using EJB 3.0 entities, the `persistence.xml` file is mandatory (unless you are using the OC4J default persistence unit).

EJB 2.1

If you are using EJB 2.1, the `persistence.xml` file is not used.

XML Reference

For EJB 3.0, this deployment descriptor file conforms to the XML schema document defined in the EJB 3.0 specification at `http://java.sun.com/products/ejb/docs.html`.

What is the orm.xml File?

The `orm.xml` file is the XML deployment descriptor you use to specify object-relational mapping configuration. You can use an `orm.xml` file as an alternative to annotations and to override annotations.

You can specify more than one `orm.xml` file and these files may be present anywhere on the class path.

For more information, see the following:

- "What is the persistence.xml File?" on page 2-8
- "Packaging a JPA Entity Application" on page 27-1

EJB 3.0

If you are using EJB 3.0 entities, the `orm.xml` file is optional.

EJB 2.1

If you are using EJB 2.1, the `orm.xml` file is not used.

XML Reference

For EJB 3.0, this deployment descriptor file conforms to the XML schema document defined in the EJB 3.0 specification at `http://java.sun.com/products/ejb/docs.html`.

How do you use an Enterprise Bean in Your Application?

In general, you use an enterprise bean from a client (see "Understanding Client Access" on page 2-10).

You can also use enterprise beans to implement fine-grained control over method invocation flow (see "Understanding EJB 3.0 Interceptors" on page 2-10).

You can also use enterprise beans with Web services, either as a Web service client or as a Web service endpoint (see "Understanding EJB and Web Services" on page 2-12).

In a deployed EJB application, you can exploit the component nature of a Java EE application to monitor and control EJB performance and resource utilization (see "Understanding EJB Administration" on page 2-12).

Understanding Client Access

In general, you use an enterprise bean from a client (see "What Type of Client do you Have?" on page 29-1) to perform application tasks such as conducting a session, persistence, or message handling. For more information, see "Accessing an Enterprise Bean From a Client" on page 29-1.

Understanding EJB 3.0 Interceptors

An interceptor is a method that you associate with an EJB 3.0 session bean business method or message-driven bean message listener method. When a client invokes such a method, OC4J intercepts the client invocation and invokes your interceptor method before allowing the client invocation to proceed.

You can define an interceptor method and interceptor life cycle callback methods on the bean class or in a separate interceptor class that you associate with the bean.

You can define only one non-life cycle callback interceptor for each bean: this method is known as the `javax.interceptor.AroundInvoke` method. Each time a business method is invoked, OC4J first invokes the `AroundInvoke` method. A life cycle callback interceptor method is invoked only when the corresponding life cycle event occurs.

An interceptor method you define in a separate interceptor class takes an invocation context as argument: using this context, your interceptor method implementation can access details of the original session bean business method or message-driven bean message listener method invocation.

This section describes the following:

- Interceptor Restrictions
- Singleton Interceptors

For more information, see the following:

- "Configuring a Life Cycle Callback Interceptor Method on an EJB 3.0 Session Bean" on page 5-4
- "Configuring a Life Cycle Callback Interceptor Method on an Interceptor Class of an EJB 3.0 Session Bean" on page 5-5
- "Configuring an Around Invoke Interceptor Method on an EJB 3.0 Session Bean" on page 5-6
- "Configuring an Interceptor Class for an EJB 3.0 Session Bean" on page 5-8
- "Configuring a Life Cycle Callback Interceptor Method on an EJB 3.0 MDB" on page 10-11
- "Configuring a Life Cycle Callback Interceptor Method on an Interceptor Class of an EJB 3.0 MDB" on page 10-11
- "Configuring an Around Invoke Interceptor Method on an EJB 3.0 MDB" on page 10-13
- "Configuring an Interceptor Class for an EJB 3.0 MDB" on page 10-15
- EJB 3.0 specification

Interceptor Restrictions

You can use interceptors with session beans (stateless and stateful) and message-driven beans.

OC4J applies an interceptor to all business methods of a bean.

If there are multiple interceptors (one interceptor method and one or more interceptor classes each with one interceptor method of their own), then each time a client invokes a business method, OC4J first invokes the interceptor classes in the order in which they are defined and then the interceptor method, before allowing the client invocation to proceed.

An interceptor method may not be a business method.

An interceptor method must have the following signature:

```
Object <METHOD>(InvocationContext) throws Exception
```

An interceptor method may have public, private, protected, or package level access but must not be declared as `final` or `static`.

Within an interceptor, you can use the `InvocationContext` to access client invocation metadata.

Interceptor method invocations occur within the same transaction and security context as the business method for which they are invoked.

Interceptor methods can mark their transaction for rollback by throwing a run-time exception, or by calling `setRollbackOnly` using its `EJBContext` object as follows:

```
InvocationContext.getEJBContext().setRollbackOnly();
```

Interceptors may cause this rollback before or after they call `InvocationContext.proceed()`.

For more information, see "Using a Rollback Strategy" on page 21-10.

When using container-managed transactions (see "What are Container-Managed Transactions?" on page 2-18), interceptors must not use any resource-manager specific transaction management methods that would interfere with the container's demarcation of transaction boundaries. For example, the interceptor must not use the following methods of the `java.sql.Connection` interface: `commit`, `setAutoCommit`, and `rollback`; or the following methods of the `javax.jms.Session` interface: `commit` and `rollback`. Interceptors must not attempt to obtain or use the `javax.transaction.UserTransaction` interface.

Singleton Interceptors

As specified in the EJB 3.0 specification, by default, OC4J creates bean interceptors: the life cycle of a bean interceptor instance is the same as that of the bean instance with which it is associated. When the bean instance is created, interceptor instances are created for each interceptor class defined for the bean. These interceptor instances are destroyed when the bean instance is removed. This allows you to store state in your interceptors.

If your interceptors are stateless, you can use an OC4J optimization extension to the EJB 3.0 specification that allows you to specify singleton interceptors. When you configure a session bean or message-driven bean to use singleton interceptors and you associate the bean with an interceptor class, OC4J creates a single instance of the interceptor class that all bean instances share. This can reduce memory requirements and life cycle overhead.

For more information, see the following:

- "Specifying Singleton Interceptors in a Session Bean" on page 5-10
- "Specifying Singleton Interceptors in an MDB" on page 10-17

Understanding EJB and Web Services

You can expose a stateless session bean as a Web service endpoint. Any EJB type can be the client of a Web service.

For more information, see "Using EJB and Web Services" on page 30-1.

Understanding EJB Administration

After you deploy your Java EE application, you can use Java EE administration features to monitor and optimize your application at run time.

For more information, see the following:

- "Administrating an EJB Application" on page 31-1
- "Optimizing EJB Performance" on page 32-1

Understanding EJB Persistence Services

OC4J supports the following persistence APIs:

- TopLink EJB 3.0 JPA persistence provider (see "How Does OC4J Manage Persistence in an EJB 3.0 Application?" on page 3-2)
- TopLink EJB 2.1 persistence manager (see "How Does OC4J Manage Persistence in an EJB 2.1 Application?" on page 3-12)

- Orion EJB 2.0 persistence manager (deprecated: see the *Oracle Containers for J2EE Orion CMP Developer's Guide*)

OC4J chooses the type of persistence to use based on the type of object-relational mappings you define and the presence or absence of certain deployment XML files. How OC4J chooses depends on the type of EJB application you are deploying:

- EJB 3.0 Applications
- EJB 2.n Applications

EJB 3.0 Applications

OC4J uses the TopLink EJB 3.0 JPA persistence provider if you deploy EJB 3.0 entities in an `ejb.jar` file without an `ejb-jar.xml` file, or if OC4J detects one or more EJB 3.0 annotations.

For more information, see the following:

- "How do You Define an EJB 3.0 Application?" on page 3-2
- "Customizing the JPA Persistence Provider" on page 3-3

EJB 2.*n* Applications

For EJB 2.1 and EJB 2.0 applications, OC4J uses the algorithm that Table 2–4 summarizes by your action. For example, if you deploy a CMP application without a `toplink-ejb-jar.xml` file, OC4J uses the TopLink persistence manager and creates default TopLink object-relational mappings.

Table 2–4 OC4J EJB 2.n Persistence Manager Selection

Your Action	toplink-ejb-jar.xml	orion-ejb-jar.xml	Persistence Manager	Mapping Type
1. Deploy.	Absent	Optional; if present, contains no mappings and no `persistence-manager` element.	Toplink	Default TopLink
1. Deploy.	Present	Optional; if present, contains no mappings and no `persistence-manager` element.	Toplink	TopLink as defined in `toplink-ejb-jar.xml` (default persistence manager properties)
1. Edit the `orion-ejb-jar.xml` file to set `persistence-manager` element `name` attribute to `toplink`[1]. 2. Edit additional `persistence-manager` subentries[1]. 3. Deploy.	Present	Optional; if present, contains no mappings	Toplink	TopLink as defined in `toplink-ejb-jar.xml` (custom persistence manager properties)
1. Deploy.	Absent	Present and contains Orion mappings; `persistence-manager` element is optional.	Orion	Orion as defined in `orion-ejb-jar.xml`
1. Edit the `orion-ejb-jar.xml` file to set `persistence-manager` element `name` attribute to `orion`[1]. 2. Deploy.	Absent	Optional; if present, contains no mappings	Orion	Default Orion

[1] See "<persistence-manager>" on page A-3.

For more information, see the following:

- "How do you Define an EJB 2.1 Module?" on page 3-11
- "Customizing the TopLink EJB 2.1 Persistence Manager" on page 3-13

Understanding EJB JNDI Services

The Java Naming and Directory Interface (JNDI) provides your Java EE application with a unified interface to multiple naming and directory services. You use JNDI to organize and locate components in a distributed Java EE environment. You can define environment references for Java EE components and associated JNDI properties.

You can use JNDI to look up and retrieve these components using the following:

- JNDI initial context
- EJB context
- EJB 3.0 annotations and resource injection

For more information, see the following:

- "Configuring JNDI Services" on page 19-1
- "How do Annotations and Resource Injection Work?" on page 1-7

Understanding EJB Data Source Services

A data source is a Java object that represents the physical enterprise information system to which OC4J persists entities. Your application uses a data source object to retrieve a connection to the enterprise information system the data source represents.

This section describes the following:

- What Types of Data Source Does OC4J Support?
- How do you Define a Connection URL in OC4J?
- What Transaction Types do Data Sources Support?
- Where do you Configure Data Source Information in OC4J?
- What is a Default Data Source?
- How Does OC4J Handle Multiple Data Sources?

For more information, see the following:

- "Configuring Data Sources" on page 20-1
- "Data Sources" in the *Oracle Containers for J2EE Services Guide*

What Types of Data Source Does OC4J Support?

OC4J supports the following types of data sources:

- Managed Data Source
- Native Data Source

Table 2–5 lists the characteristics of these OC4J data sources.

Table 2–5 OC4J Data Source Type Characteristics

Characteristic	Managed	Native
Uses OC4J connection pool?	Yes	No
Connections can participate in global transactions?	Yes	No
Connections wrapped with an OC4J `Connection` proxy?	Yes	No

Managed Data Source

A managed data source (see Example 2–2) is an OC4J-provided implementation of the `java.sql.DataSource` interface that acts as a wrapper for a JDBC driver or data source. You can associate a managed data source with a separate connection pool. Multiple managed data sources may share the same connection pool.

Example 2–2 Managed Data Source

```
<connection-pool name="ScottConnectionPool">
    <connection-factory
        factory-class="oracle.jdbc.pool.OracleDataSource"
        user="scott"
        password="tiger"
        url="jdbc:oracle:thin:@//localhost:1521/ORCL" >
    </connection-factory>
</connection-pool>

<managed-data-source
    name="OracleManagedDS"
    jndi-name="jdbc/OracleDS"
    connection-pool-name="ScottConnectionPool"
/>
```

For more information, see "Configuring a Data Source for an Oracle Database" on page 20-1.

Native Data Source

A native data source (see Example 2–3) is a JDBC vendor-provided implementation of the `java.sql.DataSource` interface. You use the connection pool provided by the data source instance you choose. Each native data source must use its own connection pool.

Example 2–3 Native Data Source

```
<native-data-source
    name="nativeDataSource"
    jndi-name="jdbc/nativeDS"
    description="Native DataSource"
    data-source-class="com.ddtek.jdbcx.sqlserver.SQLServerDataSource"
    user="frank"
    password="frankpw"
    url="jdbc:datadirect:sqlserver://server_name:1433;User=usr;Password=pwd">
</native-data-source>
```

For more information, see "Configuring a Data Source for a Third-Party Database" on page 20-2.

How do you Define a Connection URL in OC4J?

You specify a connection URL to tell OC4J where to find the underlying physical data source.

When you define a managed data source (see "Managed Data Source" on page 2-15), the connection URL is an attribute of the connection pool you associate with it (see Example 2–2).

When you define a native data source (see "Native Data Source" on page 2-15), the connection URL is an attribute of the native data source (see Example 2–3).

When specifying the connection URL to an Oracle database, you must use a service-based URL: that is, of the form `host:port/SID` (not `host:port:SID`), as Example 2–4 shows.

Example 2–4 OC4J Service-Based Connection URL

```
url="jdbc:oracle:thin:@//localhost:1521/ORCL"
```

When specifying the connection URL to a non-Oracle database, you use a URL appropriate for that system. Example 2–5 shows a typical connection URL for an SQLServer database.

Example 2–5 Non-Oracle Connection URL

```
url="jdbc:datadirect:sqlserver://server_name:1433;User=usr;Password=pwd"
```

What Transaction Types do Data Sources Support?

Managed data sources support both local and global (two-phase commit) transactions. By default, they are configured to support global transactions. For more information, see "Configuring a Data Source for an Oracle Database" on page 20-1).

Native data sources support only local transactions.

Where do you Configure Data Source Information in OC4J?

In OC4J, you configure data source information in a `data-sources.xml` file.

You can include a `data-sources.xml` file in your EAR but OC4J does not support multiple `data-sources.xml` files.

In an EJB 3.0 application, you associate a data source with a persistence unit (see "Specifying a Data Source in a Persistence Unit" on page 26-5).

For more information, see the following:

- "How Does OC4J Handle Multiple Data Sources?" on page 2-17
- "What is a Default Data Source?" on page 2-16

What is a Default Data Source?

To simplify application configuration, you can define default data sources.

How you define a default data source depends on the type of the application from which you want to access the default data source.

For information on how to configure data sources for different types of applications, see the following:

- "Configuring a Default Data Source for an EJB 3.0 Application" on page 20-3
- "Configuring a Default Data Source for an EJB 2.1 Application" on page 20-4

How Does OC4J Handle Multiple Data Sources?

OC4J does not support multiple data sources within different entities in `orion-ejb-jar.xml` file.

If your application is composed of more than one EAR and each EAR contains a `data-sources.xml`, then, when you deploy your application, OC4J will use the last entity bean's `data-source.xml` file for all entity beans.

To accommodate this scenario, specify the data source in `orion-application.xml` file or specify a default data source.

For more information, see the following:

- "In What Order Does OC4J Deploy EJB Modules?" on page 2-4
- "What is a Default Data Source?" on page 2-16

Understanding EJB Transaction Services

You can enable OC4J to manage transactions by using the Java Transaction API (JTA) supported by the Java Transaction Service (JTS). Using annotations or the deployment descriptor, you define the transactional properties of enterprise beans during design or deployment, and then let OC4J take over the responsibility of transaction management.

> **Note:** Only flat transactions are supported; nested transactions are not supported.

This section describes the following:

- Who Manages a Transaction?
- How are Transactions Handled When a Client Invokes a Business Method?
- How do You Participate in a Global or Two-Phase Commit (2PC) Transaction?

For more information, see the following:

- "Configuring Transaction Services" on page 21-1
- "Transaction Best Practices" on page 21-9
- "OC4J Transaction Support" in the *Oracle Containers for J2EE Services Guide*

Who Manages a Transaction?

A transaction can be managed by either the container (see "What are Container-Managed Transactions?" on page 2-18) or the bean ("What are Bean-Managed Transactions?" on page 2-18).

Container-managed transaction management is the default.

When configuring transaction management for your enterprise beans, consider the following restrictions:

- EJB 3.0 entities cannot be configured with a transaction management type. EJB 3.0 entities execute within the transactional context of the caller.
- EJB 2.1 entity beans must always use container-managed transaction demarcation. An EJB 2.1 entity bean must not be designated with bean-managed transaction demarcation.

For all other EJB types, you can choose either container-managed or bean-managed transaction management.

For more information, see the following:

- "Configuring EJB 3.0 Transaction Management" on page 21-1
- "Configuring EJB 2.1 Transaction Management" on page 21-4

What are Container-Managed Transactions?

When you use container-managed transactions (CMT), your EJB delegates to the container the responsibility to ensure that a transaction is started and committed when appropriate.

All session and message-driven beans may use CMT.

EJB 2.1 entity beans must use CMT.

EJB 3.0 entities cannot be configured with a transaction management type. EJB 3.0 entities execute within the transactional context of the caller.

When developing an enterprise bean that uses CMT, consider the following:

- Do not use resource manager-specific transaction management methods such as `java.sqlConnection` methods `commit`, `setAutoCommit`, and `rollback` or `javax.jms.Session` methods `commit` or `rollback`.
- Do not obtain or use the `javax.transaction.UserTransaction` interface.
- A stateful session bean using CMT may implement the `javax.ejb.SessionSynchronization` interface.
- An enterprise bean that uses CMT may use `javax.ejb.EJBContext` methods `setRollbackOnly` and `getRollbackOnly`.

For an EJB that uses CMT, for each business method, you can also specify a transaction attribute that determines how the container manages transactions when a client invokes the method (see "How are Transactions Handled When a Client Invokes a Business Method?" on page 2-19).

What are Bean-Managed Transactions?

When you use bean-managed transactions (BMT), the bean-provider is responsible for ensuring that a transaction is started and committed when appropriate.

Only session and message-driven beans may use BMT.

When developing an EJB that uses BMT, consider the following:

- Use the `javax.transaction.UserTransaction` methods `begin` and `commit` to demarcate transactions.
- A stateful session bean instance may, but is not required to, commit a started transaction before a business method returns.

  If a transaction has not been completed by the end of a business method, the container retains the association between the transaction and the instance across multiple client calls until the instance eventually completes the transaction.
- A stateless session bean instance must commit any transactions that it started before a business method or timeout callback method returns.
- A message-driven bean instance must commit a transaction before a message listener method or timeout callback method returns.

- After starting a transaction, do not use resource-manager specific transaction management methods such as `java.sqlConnection` methods `commit`, `setAutoCommit`, and `rollback` or `javax.jms.Session` methods `commit` or `rollback`.
- A bean that uses BMT must not use `EJBContext` methods `getRollbackOnly` and `setRollbackOnly`. It must use `UserTransaction` method `getStatus` and `rollback` instead.

How are Transactions Handled When a Client Invokes a Business Method?

For an enterprise bean that uses CMT (see "What are Container-Managed Transactions?" on page 2-18), you can specify a transaction attribute that determines how the container must manage transactions when a client invokes a bean method.

You can specify a transaction attribute for each of the following types of bean method:

- a method of a bean's business interface;
- a message listener method of a message-driven bean;
- a timeout callback method;
- a stateless session bean's Web service endpoint method;
- for EJB 2.1 and earlier, a method of a session or entity bean's home or component interface

Table 2–6 shows what transaction (if any) an EJB method invocation uses depending on how its transaction attribute is configured and whether or not a client-controlled transaction exists at the time the method is invoked.

OC4J starts a container-controlled transaction implicitly to satisfy the transaction attribute configuration when a bean method is invoked in the absence of a client-controlled transaction.

Table 2–6 EJB Transaction Support by Transaction Attribute

Transaction Attribute	Client-Controlled Transaction Exists	Client-Controlled Transaction Does Not Exist
Not Supported	Container suspends the client transaction	Use no transaction
Supports	Use client-controlled transaction	Use no transaction
Required[1]	Use client-controlled transaction	Container starts a new transaction
Requires New	Container suspends the client transaction and starts a new transaction	Container starts a new transaction
Mandatory	Use client-controlled transaction	Exception raised
Never	Exception raised	Use no transaction

[1] For message-driven beans using the OEMS JMS (in-memory or file-based) message service provider, Required is supported only if you access this message service provider using the Oracle JMS Connector. For more information, see "Restrictions When Accessing a Message Service Provider Without a J2CA Resource Adapter" on page 2-25.

Oracle recommends that you do not make modifications to entity beans under conditions identified as "Use no transaction". Oracle also recommends that you avoid using the `Supports` transaction attribute because it leads to a non-transactional state whenever the client does not explicitly provide a transaction.

Using TopLink CMP, a transaction must be present in order to modify an EJB 2.X CMP entity bean: If no transaction is present, the TopLink persistence manager returns a read-only copy of the bean.

For more information, see the following:

- "Configuring an EJB 3.0 Transaction Attribute" on page 21-2
- "Configuring an EJB 2.1 Transaction Attribute" on page 21-4
- "How Does OC4J Manage Persistence in an EJB 2.1 Application?" on page 3-12

How do You Participate in a Global or Two-Phase Commit (2PC) Transaction?

If all resources enlisted in a transaction are XA-compliant, then OC4J automatically coordinates a global or two-phase commit transaction.

In this release of OC4J, transaction coordination functionality is now located in OC4J, replacing in-database coordination, which is now deprecated. Also, the middle-tier coordinator is now *heterogeneous*, meaning that it supports all XA-compatible resources, not just those from Oracle.

The middle-tier coordinator supports the following:

- any XA-compliant resource;
- interpositioning and transaction inflow;
- last resource commit optimization;
- recovery logging;

For more information, see the following:

- "Configuring Message Services for Two-Phase Commit (2PC) Transactions" on page 2-29
- "Middle-Tier Two-Phase Commit (2PC) Coordinator" in the *Oracle Containers for J2EE Services Guide*

Understanding EJB Security Services

You can configure your EJB to use the Java EE security services that OC4J provides, including the following:

- Java 2 Security Model;
- Java Authentication and Authorization Service (JAAS);

For more information, see the following:

- "Configuring Security Services" on page 22-1
- "Standard Security Concepts" in the *Oracle Containers for J2EE Security Guide*

Understanding Message Services

A message service provider is responsible for providing a destination to which clients can send messages and from which message-driven beans can receive messages for processing.

OC4J supports a variety of message service providers for both XA-enabled two-phase commit (2PC) and non-XA enabled transactions.

You can access a message service provider directly or by way of a J2EE Connector Architecture (J2CA) resource adapter such as Oracle JMS Connector. For more information about the Oracle JMS Connector, see "Oracle JMS Connector: J2EE Connector Architecture (J2CA)-Based Provider" on page 2-21.

> **Note:** Oracle recommends that you access a message service provider using a J2CA resource adapter such as the Oracle JMS Connector. For more information, see "Restrictions When Accessing a Message Service Provider Without a J2CA Resource Adapter" on page 2-25.

For more information, see the following:

- "What is a Message-Driven Bean?" on page 1-56
- "What Message Service Providers Can you use With Your MDB?" on page 2-21
- "Message Service Configuration Options: Annotations or XML? Attributes or Activation Configuration Properties?" on page 2-26
- "Configuring Message Services for Two-Phase Commit (2PC) Transactions" on page 2-29
- "Configuring Message Services" on page 23-1
- "Configuring an EJB 3.0 MDB to Access a Message Service Provider Using J2CA" on page 10-1
- "Configuring an EJB 3.0 MDB to Access a Message Service Provider Directly" on page 10-3
- "Configuring an EJB 2.1 MDB to Access a Message Service Provider Using J2CA" on page 18-1
- "Configuring an EJB 2.1 MDB to Access a Message Service Provider Directly" on page 18-3

What Message Service Providers Can you use With Your MDB?

Using OC4J, you can use an MDB with any of the following Oracle Enterprise Messaging Service (OEMS) providers:

- Oracle JMS Connector: J2EE Connector Architecture (J2CA)-Based Provider
- OEMS JMS: In-Memory or File-Based Provider
- OEMS JMS Database: Advanced Queueing (AQ)-Based Provider

> **Note:** Oracle recommends that you access a message service provider using a J2CA resource adapter such as the Oracle JMS Connector. For more information, see "Restrictions When Accessing a Message Service Provider Without a J2CA Resource Adapter" on page 2-25.

Oracle JMS Connector: J2EE Connector Architecture (J2CA)-Based Provider

The Oracle JMS Connector is a J2CA 1.5-compliant resource adapter that allows OC4J-managed applications to have a unified mechanism to access any JMS provider that implements JMS 1.1 or 1.02b. Using the Oracle JMS Connector, you can integrate OC4J with various Oracle and non-Oracle JMS providers, as Figure 2–1 shows.

From the perspective of OC4J, J2CA is only used as a means of accessing a message service provider for use with message-driven beans.

Oracle JMS Connector supports both XA factories for two-phase commit (2PC) transactions and non-XA factories for transactions that do not require 2PC.

Figure 2–1 Demonstration of an MDB Interacting with a J2CA JMS Destination

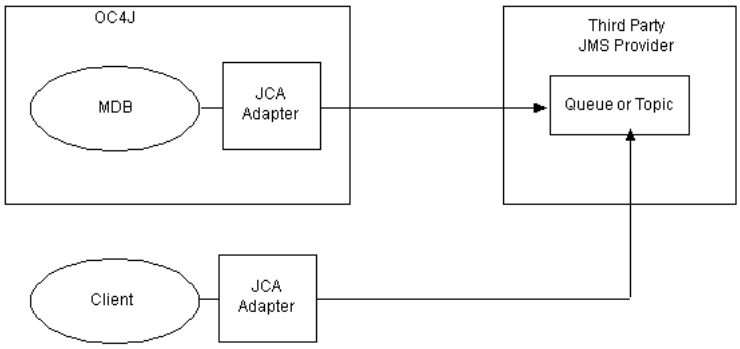


Table 2–7 summarizes the JMS message service providers that the Oracle JMS Connector supports.

Table 2–7 Oracle JMS Connector Support for JMS Message Service Providers

JMS Provider	Version
OEMS JMS: In-Memory or File-Based Provider	all
OEMS JMS Database: Advanced Queueing (AQ)-Based Provider	all
IBM WebSphere MQ-based JMS	Server Version 5.3 and 6.0
TIBCO Enterprise for JMS	3.1.0
SonicMQ	6.0

> **Note:** Oracle recommends that you access a message service provider using a J2CA resource adapter such as the Oracle JMS Connector. For more information, see "Restrictions When Accessing a Message Service Provider Without a J2CA Resource Adapter" on page 2-25.

For more information, see the following:

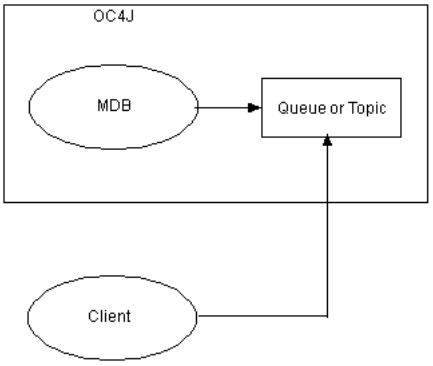
- "Configuring a J2CA Resource Adapter for use With Your Message Service Provider" on page 23-1
- "Configuring OC4J J2CA Resource Adapter Deployment XML Files" on page 23-2
- "How do You Participate in a Global or Two-Phase Commit (2PC) Transaction?" on page 2-20
- "Configuring an EJB 3.0 MDB to Access a Message Service Provider Using J2CA" on page 10-1
- "Configuring an EJB 2.1 MDB to Access a Message Service Provider Using J2CA" on page 18-1
- "Introducing Oracle JMS Support and Generic JMS Resource Adapter" in the *Oracle Containers for J2EE Resource Adapter Administrator's Guide*
- "Overview: Administering Resource Adapters" in the *Oracle Containers for J2EE Resource Adapter Administrator's Guide*

Note: For a code example of configuring a J2CA message service provider resource adapter and MDB application, see http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/how-to-gjra-with-oracleasjms/doc/how-to-gjra-with-oracleasjms.html.

OEMS JMS: In-Memory or File-Based Provider

OEMS JMS is a native Java JMS provider implementation that provides in-memory or file-based persistence and is tightly integrated with OC4J. It is the default JMS provider included with OC4J. Figure 2–2 shows how a client sends an asynchronous request directly to the OEMS JMS queue or topic that is located internally within OC4J. The MDB receives the message directly from OEMS JMS.

Figure 2–2 Demonstration of an MDB Interacting with an OEMS JMS Destination



You can access OEMS JMS directly or by way of the Oracle JMS Connector (see "Oracle JMS Connector: J2EE Connector Architecture (J2CA)-Based Provider" on page 2-21).

Note: Oracle recommends that you access a message service provider using a J2CA resource adapter such as the Oracle JMS Connector. For more information, see "Restrictions When Accessing a Message Service Provider Without a J2CA Resource Adapter" on page 2-25.

OEMS JMS supports both XA factories for two-phase commit (2PC) transactions and non-XA factories for transactions that do not require 2PC. For more information on 2PC support, see "How do You Participate in a Global or Two-Phase Commit (2PC) Transaction?" on page 2-20. For more information on how to configure OEMS JMS to support XA factories, see "Configuring jms.xml" on page 23-4.

For more information, see the following:

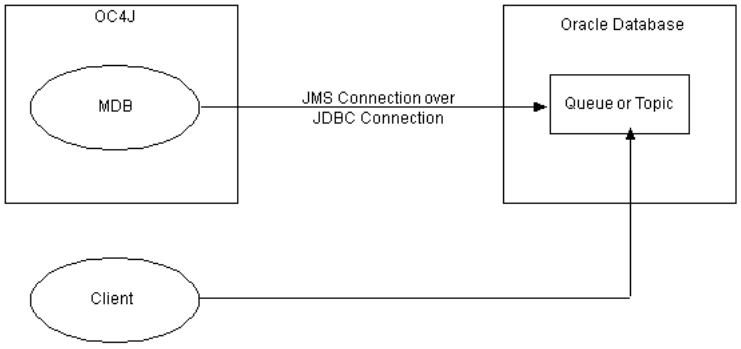
- "Configuring an OEMS JMS Message Service Provider" on page 23-3
- "Configuring an EJB 3.0 MDB to Access a Message Service Provider Using J2CA" on page 10-1
- "Configuring an EJB 3.0 MDB to Access a Message Service Provider Directly" on page 10-3

- "Configuring an EJB 2.1 MDB to Access a Message Service Provider Using J2CA" on page 18-1
- "Configuring an EJB 2.1 MDB to Access a Message Service Provider Directly" on page 18-3
- "Java Message Service (JMS)" in the *Oracle Containers for J2EE Services Guide*.

OEMS JMS Database: Advanced Queueing (AQ)-Based Provider

OEMS JMS Database is the JMS interface to the Oracle Database Streams Advanced Queueing (AQ) feature. Oracle AQ is the Oracle database-integrated message queuing feature built on the Oracle Streams information integration infrastructure that you install and configure within an Oracle database (see Figure 2–3).

Figure 2–3 Demonstration of an MDB Interacting with an OEMS JMS Database Destination



An MDB uses OEMS JMS Database as follows:

1. The MDB opens a JMS connection to the database using a data source with a username and password. The data source represents the Oracle JMS provider and uses a JDBC driver to facilitate the JMS connection.
2. The MDB opens a JMS session over the JMS connection.
3. Any message for the MDB is routed to the `onMessage` method of the MDB.

At any time, the client can send a message to the Oracle JMS topic or queue on which MDBs are listening. The Oracle JMS topic or queue is located in the database.

Before using Oracle JMS, you must create the appropriate queue or table in the database.

> **Note:** MDBs only work with certain versions of the Oracle database. See the certification matrix in the JMS chapter of the Oracle Containers for J2EE Services Guide for more information.

You can access OEMS JMS Database directly or by way of the Oracle JMS Connector (see "Oracle JMS Connector: J2EE Connector Architecture (J2CA)-Based Provider" on page 2-21).

> **Note:** Oracle recommends that you access a message service provider using a J2CA resource adapter such as the Oracle JMS Connector. For more information, see "Restrictions When Accessing a Message Service Provider Without a J2CA Resource Adapter" on page 2-25.

OEMS JMS Database supports both XA factories for two-phase commit (2PC) transactions and non-XA factories for transactions that do not require 2PC. For more information on 2PC support, see "How do You Participate in a Global or Two-Phase Commit (2PC) Transaction?" on page 2-20. For more information on how to configure OEMS JMS to support XA factories, see step 2 in "Installing and Configuring the OEMS JMS Database Provider" on page 23-6.

For more information, see the following:

- "Configuring an OEMS JMS Database Message Service Provider" on page 23-5
- "Configuring an EJB 3.0 MDB to Access a Message Service Provider Using J2CA" on page 10-1
- "Configuring an EJB 3.0 MDB to Access a Message Service Provider Directly" on page 10-3
- "Configuring an EJB 2.1 MDB to Access a Message Service Provider Using J2CA" on page 18-1
- "Configuring an EJB 2.1 MDB to Access a Message Service Provider Directly" on page 18-3
- *Oracle Streams Advanced Queuing User's Guide and Reference*
- "Java Message Service (JMS)" in the *Oracle Containers for J2EE Services Guide*

Restrictions When Accessing a Message Service Provider Without a J2CA Resource Adapter

Oracle recommends that you access a message service provider using a J2CA resource adapter such as the Oracle JMS connector (see "Oracle JMS Connector: J2EE Connector Architecture (J2CA)-Based Provider" on page 2-21).

If you choose not to use a J2CA resource adapter, consider the following restrictions:

- If your MDB uses container-managed transactions and the transaction attribute is set to `Required` or `RequiresNew`, you must use the Oracle JMS Connector.
- To be enlisted in a global, two-phase commit (2PC) transaction, both the JMS producer and consumer must come from a J2CA-based XA connection factory.

  If a JMS producer or consumer is not derived from a J2CA-based XA connection factory, it will not enlist in a global transaction (if present): in this case, JMS will behave as though it is outside the global transaction, even if you create the (non-J2CA) JMS session with a transaction parameter set to `true`, for example:

  ```
  QueueConnection.createQueueSession(true,1)
  ```

  In this case, you must invoke the `Session` methods `commit` or `rollback` independent of any global transaction.

> **Note:** In 10.1.3.1 release, by default, OC4J auto-enlists MDB connections only if the MDB uses J2CA and an XA factory. For more information, see "MDB Auto-Enlisting in Two-Phase Commit (2PC) XA Transactions" on page 2-29.

- If you are in a global transaction and you attempt to produce or consume a JMS message from a producer or consumer that is not from a J2CA-based XA connection factory, you will get a run-time JMS exception.
- Container-Managed Transactions (CMT's) and Bean-Managed Transactions (BMT's) with the OEMS JMS provider (see OEMS JMS: In-Memory or File-Based Provider on page 2-23) are only supported through the use of a J2CA resource adapter. These features will not be supported when using the `oc4j.jms.pseudoTransactionEnlistment` backward compatibility flag (see "MDB Auto-Enlisting in Two-Phase Commit (2PC) XA Transactions" on page 2-29).

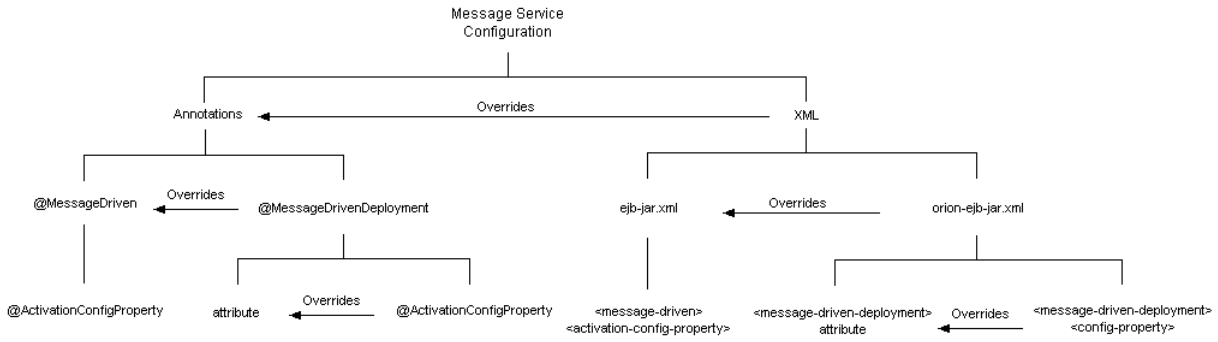
Message Service Configuration Options: Annotations or XML? Attributes or Activation Configuration Properties?

As Figure 2–4 shows, there are two ways to configure message service provider options:

- Message Service Configuration Using Annotations
- Message Service Configuration Using XML

In general, Oracle recommends that you use activation configuration properties instead of attributes, whether you choose annotations or XML.

Figure 2–4 Message Service Configuration Options



Message Service Configuration Using Annotations

Using annotations, you can configure message service options using the `@MessageDrivenDeployment` or `@MessageDriven` annotation: `@MessageDrivenDeployment` configuration overrides that in `@MessageDriven`.

If you use `@MessageDrivenDeployment`, you can configure message service options using nested `@ActivationConfigProperty` annotations or using `@MessageDrivenDeployment` attributes: `@ActivationConfigProperty` configuration overrides `@MessageDrivenDeployment` attributes.

If you use `@MessageDriven`, you can configure message service options using nested `@ActivationConfigProperty` annotations only.

If you configure using `@MessageDrivenDeployment` attributes, your application can only access a message service provider without a J2CA resource adapter. If later you decide to access your message service provider using a J2CA resource adapter, your application will fail to deploy. If you configure using nested `@ActivationConfigProperty` annotations, your application can access a message service provider with or without a J2CA resource adapter. Oracle recommends that if you configure using annotations, you should use the `@ActivationConfigProperty` approach.

Example 2–6 shows both a `@MessageDrivenDeployment` and `@MessageDriven` annotation using `@ActivationConfigProperty` annotations for message service configuration. Note that the `DestinationName` activation configuration property in the `@MessageDrivenDeployment` annotation overrides that in the `@MessageDriven` annotation.

Example 2–6 @MessageDriven and @MessageDrivenDeployment Annotation for a J2CA Message Service Provider

```
import javax.ejb.MessageDriven;
import oracle.j2ee.ejb.MessageDrivenDeployment;
import javax.ejb.ActivationConfigProperty;
import javax.jms.Message;
import javax.jms.MessageListener;

@MessageDriven(
    activationConfig = {
        @ActivationConfigProperty(
            propertyName="DestinationName", propertyValue="OracleASjms/MyQueue"
        )
    }
)

@MessageDrivenDeployment(
    activationConfig = {
        @ActivationConfigProperty(
            propertyName="DestinationName", propertyValue="OracleASjms/DeployedQueue"
        ),
        @ActivationConfigProperty(
            propertyName="ResourceAdapter", propertyValue="OracleASjms"
        )
    }
)

public class JCAQueueMDB implements MessageListener
{
    public void onMessage(Message msg) {
        ...
    }
}
```

Message Service Configuration Using XML

Using XML, you can configure message service options using the `orion-ejb-jar.xml` file `<message-driven-deployment>` element or the `ejb-jar.xml` file `<message-driven>` element: `orion-ejb-jar.xml` configuration overrides that in `ejb-jar.xml`.

If you use the `orion-ejb-jar.xml` file `<message-driven-deployment>` element, you can configure message service options using nested `<config-property>` elements or using `<message-driven-deployment>`

attributes: <config-property> configuration overrides <message-driven-deployment> attributes.

If you use the ejb-jar.xml file <message-driven> element, you can configure message service options using nested <activation-config-property> elements only.

If you configure using orion-ejb-jar.xml file <message-driven-deployment> element attributes, your application can only access a message service provider without a J2CA resource adapter. If later you decide to access your message service provider using a J2CA resource adapter, your application will fail to deploy. If you configure using nested <config-property> elements, your application can access a message service provider with or without a J2CA resource adapter. Oracle recommends that if you configure using XML, you should use the <config-property> approach.

Example 2–8 shows how to use <config-property> elements in the orion-ejb-jar.xml file <message-driven-deployment> element and Example 2–7 shows how to use <activation-config-property> elements in the ejb-jar.xml file <message-driven> element. Note that the DestinationName activation configuration property in the <message-driven-deployment> element overrides that in the <message-driven> element. Also note that in the ejb-jar.xml file <message-driven> element, <activation-config-property> elements are contained in an <activation-config> element.

Example 2–7 ejb-jar.xml <activation-config-property>

```
<message-driven>
    <ejb-name>JCA_QueueMDB</ejb-name>
    <ejb-class>test.JCA_MDB</ejb-class>
    <messaging-type>javax.jms.MessageListener</messaging-type>
    <transaction-type>Container</transaction-type>
    ...
    <activation-config>
        <activation-config-property>
            <activation-config-property-name>
                DestinationName
            </activation-config-property-name>
            <activation-config-property-value>
                OracleASJMSSubcontext
            </activation-config-property-value>
        </activation-config-property>
    </activation-config>
    ...
</message-driven>
```

Example 2–8 orion-ejb-jar.xml <config-property>

```
<message-driven-deployment
    name="JCA_QueueMDB"
    resource-adapter="OracleASjms">
    ...
    <config-property>
        <config-property-name>
            DestinationName
        </config-property-name>
        <config-property-value>
            OracleASJMSRASubcontext
        </config-property-value>
    </config-property>
    ...
</message-driven-deployment>
```

Configuring Message Services for Two-Phase Commit (2PC) Transactions

OC4J supports 2PC transactions with XA-enabled resources (see "How do You Participate in a Global or Two-Phase Commit (2PC) Transaction?" on page 2-20).

> **Note:** In 10.1.3.1 release, by default, OC4J auto-enlists MDB connections only if the MDB uses J2CA and an XA factory. For more information, see "MDB Auto-Enlisting in Two-Phase Commit (2PC) XA Transactions" on page 2-29.

For more information on configuring JMS message service providers to be XA-compliant, see the following:

- Oracle JMS Connector: "Configuring OC4J J2CA Resource Adapter Deployment XML Files" on page 23-2
- OEMS JMS: "Configuring jms.xml" on page 23-4
- OEMS JMS Database: step 2 in "Installing and Configuring the OEMS JMS Database Provider" on page 23-6

MDB Auto-Enlisting in Two-Phase Commit (2PC) XA Transactions

In 10.1.2 and 10.1.3.0 releases of OC4J, both normal non-XA JMS connections as well as XA JMS connections were automatically enlisted into an OC4J global transaction by the native OEMS JMS provider. In 10.1.3.1 release, neither XA nor normal JMS connections are enlisted into an OC4J global transaction. If you use the provided JMS APIs, you should explicitly enlist an XA connection into an OC4J global transaction using the `javax.jms.XA*` implementation of OEMS JMS. You should also explicitly commit or rollback the local transaction of a given JMS session created from a non-XA JMS connection.

For backward-compatibility reasons, it is still possible (but discouraged) to use the auto-enlisting feature in 10.1.3.1 release. Disabled by default, you can enable auto-enlisting by setting global OC4J system property `oc4j.jms.pseudoTransactionEnlistment` to `true`.

A J2CA MDB configured to use an XA connection factory (thereby creating an XA session) will auto-enlist as expected. Sessions created from non-XA factories will not enlist. The `oc4j.jms.pseudoTransactionEnlistment` property is only required to force enlistment of non-XA sessions. This is mostly a concern to legacy applications that do not use J2CA.

For more information, see "Java Message Service (JMS)" in the *Oracle Containers for J2EE Services Guide*.

Understanding OC4J EJB Application Clustering Services

Oracle Application Server provides an extensive suite of high availability and failover options, including clustering–the distribution of application server and end-user application components across multiple hosts configured with the appropriate means of host-to-host communication.

OC4J application clustering is a state management service available to HTTP sessions and stateful session beans. In this context, a cluster is defined as two or more OC4J

server nodes hosting the same set of applications. In this release, configuration has been simplified and made identical for both HTTP sessions and stateful session beans.

Transactions cannot failover. There is no reinstating an interrupted transaction in another bean. Instead, the transaction rolls back and must start over. For more information, see "Understanding EJB Transaction Services" on page 2-17.

The performance for clustering stateful session beans is dependent on the type of replication (see "State Replication" on page 2-30) and load balancing (see "Load Balancing" on page 2-31) options you choose.

You must choose the appropriate balance between replication frequency and robustness: the more frequently you replicate, the smaller the window of opportunity for losing state but the higher the load on the application server and network.

Note: If you have a servlet (or other Web component) that invokes a stateful session bean, you must configure both HTTP session and stateful session bean clustering.

This section describes OC4J application clustering for stateful session beans, including the following:

- State Replication
- Load Balancing

For more information, see the following:

- "Configuring OC4J EJB Application Clustering Services" on page 24-1
- "Clustering Overview" in the *Oracle Containers for J2EE Configuration and Administration Guide*
- "Application Clustering in OC4J" in the *Oracle Containers for J2EE Configuration and Administration Guide*
- "Oracle Application Server Cluster (OC4J) in Active-Active Topologies" in the *Oracle Application Server High Availability Guide*
- "Stateful Session EJB State Replication with Oracle Application Server Cluster (OC4J)" in the *Oracle Application Server High Availability Guide*

State Replication

When you configure a replication policy for a clustered OC4J EJB application, OC4J handles the replication of objects and values contained in stateful session bean instances. Only stateful session beans can be clustered. Because stateless session beans have no state to be replicated, they need not be clustered.

You must configure a replication policy to take advantage of failover: replication of bean state, so that when the original bean terminates unexpectedly, the request can be transparently forwarded to another OC4J process in the cluster.

If you only want to take advantage of load balancing, replication is not required (see "Load Balancing" on page 2-31).

A replication policy determines the state replication trigger–the conditions under which bean state is broadcast to all other OC4J processes in the cluster. For stateful session beans, when replication is triggered, all the attributes of the stateful session bean are replicated (regardless of whether or not they have changed).

Replication can have an impact on application server and network performance. The fewer times the state is sent out, the better your performance. However, there is a trade-off between performance and the confidence that the bean state is replicated to cover for all areas of the bean instance failing.

For more information, see the following:

- "Configuring EJB 3.0 and EJB 2.1 Stateful Session Bean Replication Policy" on page 24-1
- "Understanding OC4J EJB Application Clustering Services" on page 2-29

Load Balancing

Load balancing refers to how incoming client requests are distributed over all the OC4J instances in your cluster. You can choose from among the following load balancing strategies:

- Replication-based: hen you configure a replication policy for a clustered OC4J EJB application (see "State Replication" on page 2-30), OC4J will automatically select an OC4J instance at random from the pool of OC4J instances in the cluster when the first client request is serviced.
- Static retrieval: If you decide not to use EJB replication, but you want to load balance client requests across several statically specified OC4J processes, you can use static retrieval by providing the URLs for all of these processes in the JNDI URL property. For more information, see "Configuring Static Retrieval Load Balancing" on page 24-3.
- DNS: If you decide not to use EJB replication, but you want to load-balance client requests across several DNS-managed OC4J processes, you can use DNS retrieval by configuring your DNS server with a single hostname associated with the desired OC4J host IP addresses and specifying this hostname in the JNDI URL property. For more information, see "Configuring DNS Load Balancing" on page 24-3.

For all load balancing strategies, you can configure how a client's requests are load balanced across the OC4J instances in your cluster (see "Configuring Load Balancing Behavior" on page 24-4).

Understanding EJB Timer Services

You can set up a timer that invokes a timeout callback method at a specified time, after a specified elapsed time, or at specified intervals.

Note: Timers apply to all EJB types except stateful session beans and EJB 3.0 entities.

EJB timers are supported only in an OC4J instance that runs on a single JVM (where numprocs=1 in the `<process-set>` element of the `opmn.xml` configuration file).

For EJB 3.0 applications, using the `@Timeout` annotation, you can annotate any EJB method as the timeout callback method.

For EJB 2.1 applications, your EJB must implement the `TimedObject` interface and provide a timeout callback method named `ejbTimeout`.

Timers are for use in modeling of application-level processes, not for real-time events.

OC4J provides standard Java EE timers as well as a convenient Java EE timer extension that allows configuration similar to the Unix cron utility.

For more information, see:

- "Understanding Java EE Timer Services" on page 2-32
- "Understanding OC4J Cron Timer Services" on page 2-32

Timer and timeout callback methods should be called within a transaction. OC4J supports transaction attribute `REQUIRES_NEW` for timeout callbacks. For more information on transaction attributes, see "How are Transactions Handled When a Client Invokes a Business Method?" on page 2-19).

An enterprise bean accesses EJB timer services by means of dependency injection, through the `EJBContext` interface, or through lookup in the JNDI namespace.

For more information, see "Configuring Timer Services" on page 25-1.

Understanding Java EE Timer Services

The EJB timer service is a container-managed service you use to define callback methods on your EJB that are scheduled for time-based events. The EJB timer service provides a reliable and transactional notification service for timed events. Timer notifications may be scheduled to occur at a specific time, after a specific elapsed duration, or at specific recurring intervals. You can define callback methods on your EJB to receive these time-based events. The Java EE timer service is implemented by OC4J.

For more information, see "Configuring an Enterprise Bean With a Java EE Timer" on page 25-1.

Understanding OC4J Cron Timer Services

In UNIX, you can schedule a cron timer to execute regularly at specified intervals. Oracle has extended OC4J to support cron timers with EJB. You can use cron expressions for scheduling timer events with EJB deployed to OC4J. Using an OC4J cron timer, you can create timers that invoke a timeout callback method or any arbitrary Java class's `main` method.

For more information, see "Configuring an Enterprise Bean With an OC4J Cron Timer" on page 25-3.

3

Understanding EJB Support in OC4J

This chapter describes the following:

- EJB 3.0 Support
- EJB 2.1 Support

For more information, see *Oracle Application Server Release Notes for Microsoft Windows*.

EJB 3.0 Support

In this release, OC4J supports all but a small subset of the functionality specified in the final EJB 3.0 specification (`http://jcp.org/aboutJava/communityprocess/pr/jsr220/index.html`).

You may need to make minor code changes to your EJB 3.0 OC4J application after OC4J is updated to full EJB 3.0 compliance. For more information, see "Migrating a 10.1.3.0 TopLink JPA Preview Application to 10.1.3.1 TopLink Essentials JPA" on page 3-5.

In this release, OC4J supports the use of annotations, standard deployment XML (`ejb-jar.xml` or `orion-ejb-jar.xml`), or both for all EJB 3.0 features except for the object-relational entity mapping types (namely basic, binary large object (LOB), serialized, one-to-one, many-to-one, one-to-many, many-to-many, and aggregate mappings). For these, you must either use annotations or TopLink JPA persistence provider customization.

OC4J supports the proprietary EJB 3.0 annotations that the *Oracle Application Server Annotations API Reference* describes.

For more information, see:

- "Implementing a JPA Entity" on page 6-1
- "Customizing the JPA Persistence Provider" on page 3-3

In this release, OC4J supports resource injection in the Web tier. For more information, see "Annotations in the Web Tier" on page 1-9.

This section describes the following:

- What JDK is Required?
- How do You Define an EJB 3.0 Application?
- How Does OC4J Manage Persistence in an EJB 3.0 Application?

What JDK is Required?

By default, if you are using EJB 3.0, then you must use JDK 1.5. By default, OC4J does not support the use of EJB 3.0 and JDK 1.4.

OC4J supports the use of EJB 3.0 (excluding annotations and interceptors) with JDK 1.4 only with the TopLink JPA preview persistence provider. For more information, see the discussion of system property `default.persistence.provider` in "JPA Persistence JAR Files" on page 3-2.

How do You Define an EJB 3.0 Application?

For entities, OC4J assumes that the application is an EJB 3.0 application, if an EJB JAR is deployed without an `ejb-jar.xml` file. For more information, see "Understanding EJB Persistence Services" on page 2-12

For session beans and message-driven beans, OC4J assumes that the application is an EJB 3.0 application if the `ejb-jar.xml` file `<ejb-jar>` element `version` attribute is set to `3.0`.

How Does OC4J Manage Persistence in an EJB 3.0 Application?

In an EJB 3.0 application, OC4J delegates persistence operations to a JPA persistence provider: in this release, OC4J uses TopLink Essentials, the JPA persistence provider for the EJB 3.0 Reference Implementation (see "TopLink Essentials JPA Persistence Provider" on page 3-2).

TopLink Essentials JPA Persistence Provider

Oracle TopLink is an advanced, object-persistence and object-transformation framework that provides development tools and run-time capabilities that reduce development and maintenance efforts, and increase enterprise application functionality.

In this release, OC4J manages EJB 3.0 entities using TopLink Essentials, the JPA persistence provider for the EJB 3.0 Reference Implementation. For more information, see "What is TopLink?" in the *Oracle TopLink Developer's Guide*.

OC4J provides JAR files for both the classes that the EJB 3.0 persistence specification mandates and the classes that make up the TopLink Essentials JPA persistence provider implementation. For more information about persistence JAR files, see "JPA Persistence JAR Files" on page 3-2.

For EJB 3.0 projects, you configure persistence properties through annotations or `persistence.xml` file. OC4J translates this metadata into TopLink configuration. For more information on customizing the TopLink Essentials EJB 3.0 JPA persistence provider, see "Customizing the JPA Persistence Provider" on page 3-3.

JPA Persistence JAR Files

OC4J uses the JAR files that Table 3–1 lists to provide the TopLink Essentials JPA persistence provider implementation. These JAR files are located in the `<ORACLE_HOME>/toplink/jlib` directory.

The system property `default.persistence.provider` determines which JPA persistence provider implementation OC4J uses. The following are valid values:

- `essentials` (default): OC4J uses the `<ORACLE_HOME>/j2ee/home/lib/persistence.jar` to provide EJB 3.0 JPA classes that the EJB 3.0 persistence specification mandates and `<ORACLE_`

HOME>/toplink/jlib/toplink-essentials.jar and toplink-essentials-agent.jar for the persistence provider implementation, providing full support for the final EJB 3.0 persistence specification.

- toplink: OC4J uses the <ORACLE_HOME>/j2ee/home/lib/preview-persistence.jar to provide EJB 3.0 JPA classes that the public review EJB 3.0 persistence specification mandated and <ORACLE_HOME>/toplink/jlib/toplink.jar as the persistence provider implementation, providing a JPA preview based on a subset of the functionality specified in the EJB 3.0 public review draft. You can use this option to run an application written to the preview API. Oracle does not recommend that you use this option.

Table 3–1 TopLink JAR Files

JAR File	Contents
antlr.jar	This JAR file contains the Antlr (ANother Tool for Language Recognition) tool.
toplink.jar	This JAR file contains all the classes that comprise the TopLink API, including classes with Oracle JDBC dependencies. If you want to use an Oracle JDBC driver version different than the default version installed with OC4J, see "Associating TopLink With an Oracle JDBC Driver" on page 20-4.
toplink-essentials.jar	This JAR file contains the open source JPA edition of TopLink, the JPA persistence provider for the EJB 3.0 Reference Implementation.
toplink-essentials-agent.jar	This JAR file contains the classes that TopLink uses to perform byte-code weaving on JPA entities to automatically enable features such as ValueHolder indirection. You invoke toplink-essentials-agent.jar by adding -javaagent:toplink-essentials-agent.jar to your client JVM command line or by using the toplink.weaving TopLink JPA extension that you can define in a persistence.xml file. This option is not necessary on the server JVM command line. This JAR file provides part of TopLink Essentials, the JPA persistence provider for the EJB 3.0 Reference Implementation. It is used with toplink-essentials.jar. This JAR is optional. It should never be placed on the classpath and should only be used as part of -javaagent.
toplink-oc4j.jar	This JAR contains the classes that TopLink uses to integrate with Oracle Containers for J2EE. This JAR file is only used in OC4J; the container is preconfigured to use toplink-oc4j.jar. In a non-OC4J application, use toplink.jar.

Customizing the JPA Persistence Provider

Typically, you use object-relational annotations (see "Configuring a Container-Managed Relationship Field for a JPA Entity" on page 7-9) to specify how you want OC4J to store a persistent field in the database and rely on the default TopLink EJB 3.0 JPA persistence provider configuration for each such annotation. However, you may wish to override this default behavior to suit your application requirements. As well, although the TopLink EJB 3.0 JPA persistence provider is JPA compliant, it provides additional extensions beyond what is defined in the JPA specification.

You may customize the OC4J JPA persistence provider in any of the following ways:

- Set vendor-specific query hints in a named or dynamic query (see "Configuring TopLink Query Hints in a JPA Query" on page 8-3).
- Set vendor-specific properties in the persistence.xml file using a <properties> element or in the Map of properties passed into the javax.persistence.Persistence method

`createEntityManagerFactory` (see "Configuring Vendor Extensions in a Persistence Unit" on page 26-5).

- If you are using the TopLink Essentials persistence provider (default), you can access TopLink API in a JPA entity application at run time by using TopLink JPA extensions ("Accessing TopLink API at Run Time With TopLink Essentials JPA Persistence" on page 3-4).
- If you are using the TopLink JPA preview persistence provider, you can access the TopLink API in a JPA entity application at run time by creating an `ejb3-toplink-sessions.xml` and `toplink-ejb-jar.xml` file and packaging them in the `META-INF` directory of the EJB-JAR that contains your EJB 3.0 entities (see "Accessing TopLink API at Run Time With TopLink JPA Preview Persistence" on page 3-4).
- Configure the TopLink EJB 3.0 JPA persistence provider to use an Oracle JDBC driver version different than the default version installed with OC4J (see "Associating TopLink With an Oracle JDBC Driver" on page 20-4).

Accessing TopLink API at Run Time With TopLink Essentials JPA Persistence

If you are using the TopLink Essentials JPA persistence provider (default), you access TopLink API in a JPA entity application at run time by using TopLink JPA extensions `toplink.session.customizer` and `toplink.descriptor.customizer.<ENTITY>` (see Table 26–5 on page 26-20).

Accessing TopLink API at Run Time With TopLink JPA Preview Persistence

If you are using the TopLink JPA preview persistence provider, you access the TopLink API in a JPA entity application at run time by creating an `ejb3-toplink-sessions.xml` file (see "What is the ejb3-toplink-sessions.xml File?" on page 2-7) and `toplink-ejb-jar.xml` (see "What is the toplink-ejb-jar.xml File?" on page 2-6) file.

Note: By default, OC4J uses the TopLink Essentials JPA persistence provider. In this case, you access TopLink API in a JPA entity application at run time by using TopLink JPA extensions (see "Accessing TopLink API at Run Time With TopLink Essentials JPA Persistence" on page 3-4).

You package these files in the META-INF directory of the EJB-JAR that contains your EJB 3.0 entities.

- To customize TopLink session-level options, you only need an `ejb3-toplink-sessions.xml` file.
- To customize TopLink persistence-specific options, you need both an `ejb3-toplink-sessions.xml` and `toplink-ejb-jar.xml` file.

You can use the TopLink API to customize persistence by overriding annotations or by replacing annotations altogether. For example, you might use annotations for most of your object-relational mappings and an `ejb3-toplink-sessions.xml` and `toplink-ejb-jar.xml` file to specify object-relational mappings for a subset of complex relationships not suited to annotation.

If the only JDK 1.5 language extension that your entity classes use are annotations, you can use the TopLink Workbench to create and configure these files. Oracle recommends using the TopLink Workbench to create and configure these files.

To customize the TopLink JPA preview persistence provider, do the following:

1. Create a relational TopLink Workbench project (see "Creating a Project" in the *Oracle TopLink Developer's Guide*).
2. Configure the TopLink Workbench project classpath to include your JDK 1.5 compiled entity classes (see "Configuring Project Classpath" in the *Oracle TopLink Developer's Guide*).
3. Configure the project deployment XML file name (as `toplink-ejb-jar.xml`) and save location (see "Configuring Project Deployment XML Options" in the *Oracle TopLink Developer's Guide*).
4. Optionally, configure other TopLink project-level options (see "Configuring a Relational Project" in the *Oracle TopLink Developer's Guide*).
5. Configure TopLink relational descriptors for the entity classes you want to customize (see "Creating a Relational Descriptor" in the *Oracle TopLink Developer's Guide* and "Configuring a Relational Descriptor" in the *Oracle TopLink Developer's Guide*).
6. Configure TopLink relational mappings for the persistent fields you want to customize (see "Creating a Mapping" in the *Oracle TopLink Developer's Guide* and "Configuring a Relational Mapping" in the *Oracle TopLink Developer's Guide*).
7. Export your TopLink Workbench project to the `toplink-ejb-jar.xml` XML file (see "Exporting Deployment XML Information" in the *Oracle TopLink Developer's Guide*).
8. Create a TopLink sessions configuration file named `ejb3-toplink-sessions.xml` (see "Creating a Server Session" in the *Oracle TopLink Developer's Guide*).
9. Set the `ejb3-toplink-sessions.xml` file primary project to your `toplink-ejb-jar.xml` file (see "Configuring a Primary Mapping Project" in the *Oracle TopLink Developer's Guide*).
10. Optionally, configure any other TopLink session-level options (see "Configuring a Server Session" in the *Oracle TopLink Developer's Guide*).
11. Save your TopLink Workbench sessions configuration file.
12. Package the `ejb3-toplink-sessions.xml` and `toplink-ejb-jar.xml` file in the META-INF directory of the EJB-JAR that contains your EJB 3.0 entities.

Note: Alternatively, you can use JDeveloper to create the `ejb3-toplink-sessions.xml` and `toplink-ejb-jar.xml` file (see "Using EJB Development Tools". on page 2-1).

Migrating a 10.1.3.0 TopLink JPA Preview Application to 10.1.3.1 TopLink Essentials JPA

In 10.1.3.0 release, OC4J uses the TopLink JPA preview persistence provider based on a subset of the functionality specified in the EJB 3.0 public review draft.

In 10.1.3.1 release, OC4J uses the TopLink Essentials JPA persistence provider, the JPA persistence provider for the EJB 3.0 Reference Implementation, to provide full JPA support according to the final EJB 3.0 specification.

You must make code changes to your JPA preview-based application before using it with TopLink Essentials and the final EJB 3.0 API in OC4J 10.1.3.1.

In general, you should do the following:

1. Undeploy your preview-based application.
2. Upgrade OC4J from 10.1.3.0 to 10.1.3.1.
3. Make the necessary post-upgrade configuration file changes (see "Changes in OC4J Configuration Files" on page 3-6).
4. Migrate your code to use the new EJB 3.0 API.

   The following sections describe the important differences between the TopLink JPA preview and full TopLink JPA to help you identify where changes must be made:

   - "Changes in javax.persistence" on page 3-6
   - "Changes in oracle.toplink.essentials.platform.database" on page 3-10
   - "Changes in Interceptor Support" on page 3-10
   - "Acquiring an Entity Manager" on page 3-10
   - "New JAR Files" on page 3-11
5. Redeploy your updated application.

Changes in OC4J Configuration Files

After applying the 10.1.3.1.0 patch set to a 10.1.3.0.0 OC4J, you must manually edit OC4J configuration files as follows:

1. Edit the `<ORACLE_HOME>/j2ee/home/config/server.xml` file and add the following:

```
<shared-library name="oracle.persistence" version="1.0"
library-compatible="true">
    <code-source path="../../../toplink/jlib/toplink-essentials.jar"/>
</shared-library>
```

2. Edit the `<ORACLE_HOME>/j2ee/home/config/system-application.xml` file and add the following to the `<imported-shared-libraries>` element:

```
<import-shared-library name="oracle.persistence"/>
```

Changes in javax.persistence

Table 3–2 lists the additions, deletions, and changes made to the `javax.persistence` package between 10.1.3.0 and 10.1.3.1. If your application uses any of these classes, consult the latest EBJ 3.0 specification and JPA Javadoc for details.

Table 3–2 Changes to javax.persistence

10.1.3.0	10.1.3.1	Description
`AccessMode`	deleted	In 10.1.3.1 release, EJB 3.0 entities do not require local or remote interfaces. Because all entity access is by way of an `EntityManager`, clients need not be concerned about whether access is local or remote.
`AccessType`	deleted	In 10.1.3.1 release, the EJB 3.0 persistence specification requires the use of a single access type in an entity hierarchy. The placement of the mapping annotations determines the access type in effect.
N/A	`AssociationOverride`	Added in 10.1.3.1 release as part of the changes made to annotations for inheritance. For more information, see `http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#AssociationOverride`.

Table 3–2 (Cont.) Changes to javax.persistence

10.1.3.0	10.1.3.1	Description
N/A	`AssociationOverrides`	Added in 10.1.3.1 release as part of the changes made to annotations for inheritance. For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#AssociationOverrides.
`Basic`	`Basic`	In 10.1.3.1 release, attribute `temporalType` is omitted. For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#Basic.
N/A	`DiscriminatorValue`	Added in 10.1.3.1 release as part of the changes made to annotations for inheritance. For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#InheritanceAnnotations
`EmbeddableSuperclass`	deleted	Added in 10.1.3.1 release as part of the changes made to annotations for inheritance. For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#InheritanceAnnotations
`Entity`	`Entity`	In 10.1.3.1 release, attribute `access` is omitted.
N/A	`EntityExistsException`	Added in 10.1.3.1 release.
`EntityListener`	deleted	In 10.1.3.1 release, use `EntityListners` instead. For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#EntityListeners.
N/A	`EntityListeners`	Added in 10.1.3.1 release. For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#EntityListeners.
`EntityManager`	`EntityManager`	In 10.1.3.1 release, `EntityManager` method `getUserTransaction` is named `getTransaction`. New methods added in 10.1.3.1 include the following: ▪ `setFlushMode` ▪ `getFlushMode` ▪ `lock` ▪ `clear` ▪ `joinTransaction` ▪ `getDelegate` In 10.1.3.1 release, methods of this class throw additional exceptions such as `EntityExistsException` and `IllegalStateException`, and, in some cases, methods throw `IllegalStateException` instead of `IllegalArgumentException`.
`EntityNotFoundException`	`EntityNotFoundException`	In 10.1.3.1 release, this exception extends `PersistenceException` instead of `RuntimeException`.
`EntityTransaction`	`EntityTransaction`	New methods added in 10.1.3.1 release include the following: ▪ `setRollbackOnly` ▪ `getRollbackOnly`
`EntityType`	deleted	In 10.1.3.1 release, the EJB 3.0 specification removes the concept of BMP. To develop and deploy a Java EE 5 BMP application, you must use the EJB 2.1 API.

Table 3–2 (Cont.) Changes to javax.persistence

10.1.3.0	10.1.3.1	Description
N/A	`Enumerated`	Added in 10.1.3.1 release to support direct mappings of enumerated types. For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#Enumerated.
N/A	`EnumType`	Added in 10.1.3.1 release to support direct mappings of enumerated types. For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#Enumerated.
N/A	`ExcludeDefaultListeners`	Added in 10.1.3.1 release to support managing life cycle callback default listeners. For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#ExcludeDefaultListeners.
N/A	`ExcludeSuperclassListeners`	Added in 10.1.3.1 release to support managing life cycle callback superclass listeners. For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#ExcludeSuperclassListeners.
`FlushMode`	deleted	In 10.1.3.1 release, to set the flush mode, use the `EntityManager` method `setFlushMode` to set the `FlushModeType`.
N/A	`GeneratedValue`	Added in 10.1.3.1 release to support automatic primary key (identity) generation. For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#GeneratedValue.
N/A	`GenerationType`	Added in 10.1.3.1 release to support automatic primary key (identity) generation. For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#GeneratedValue.
`GeneratorType`	deleted	In 10.1.3.1 release, use `GeneratedValue` attribute `strategy` to set the `GenerationType`.
`Inhertiance`	`Inheritance`	In 10.1.3.1 release, the following attributes are omitted: ▪ `discriminatorType` ▪ `discriminatorValue` For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#InheritanceAnnotations.
`JoinColumn`	`JoinColumn`	In 10.1.3.1 release, attribute `secondaryTable` is changed to `table`. For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#JoinColumn.
`JoinTable`	`JoinTable`	In 10.1.3.1 release, attribute `table` (type `Table`) is changed to `name` (type `String`). The following attributes are added: ▪ `catalog` ▪ `schema` ▪ `uniqueConstraints` For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#JoinTable.

Table 3–2 (Cont.) Changes to javax.persistence

10.1.3.0	10.1.3.1	Description
LobType	deleted	In 10.1.3.1 release, use direct mapping annotation @Lob to specify a Lob type. A Lob may be either a binary or character type. The persistence provider infers the Lob type from the type of the persistent field or property. For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#Lob.
N/A	LockModeType	Added in 10.1.3.1 release.
N/A	MappedSuperclass	Added in 10.1.3.1 release as part of the changes made to annotations for inheritance. For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#MappedSuperclass.
NamedNativeQuery	NamedNativeQuery	In 10.1.3.1 release, attribute queryString changed to query and attribute hints was added. For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#NamedNativeQuery.
NamedQuery	NamedQuery	In 10.1.3.1 release, attribute queryString changed to query and attribute hints was added. For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#NamedQuery.
NoResultException	NoResultException	In 10.1.3.1 release, this exception extends PersistenceException instead of RuntimeException.
N/A	OptimisticLockException	Added in 10.1.3.1 release.
PersistenceContext	PersistenceContext	In 10.1.3.1 release, attribute properties was added. For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#PersistenceContext.
N/A	PersistenceProperty	For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#PersistenceProperty.
N/A	QueryHint	For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#QueryHint.
N/A	RollbackException	Added in 10.1.3.1 release.
SecondaryTable	SecondaryTable	In 10.1.3.1 release, attribute pkJoin was changed to pkJoinColumns. For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#SecondaryTable.
SequenceGenerator	SequenceGenerator	In 10.1.3.1 release, the default for attribute initialValue changed from 0 to 1. For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#SequenceGenerator.
N/A	SqlResultSetMappings	For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#SqlResultSetMappings.
Table	Table	In 10.1.3.1 release, attribute specified is omitted. For more information, see http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#Table.

Table 3–2 (Cont.) Changes to javax.persistence

10.1.3.0	10.1.3.1	Description
`TableGenerator`	`TableGenerator`	In 10.1.3.1 release, the following attributes were added: ▪ `catalog` ▪ `schema` ▪ `uniqueConstraints` For more information, see `http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#TableGenerator`.
N/A	`Temporal`	For more information, see `http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#Temporal`.
`TemporalType`	`TemporalType`	In 10.1.3.1 release, enum value `NONE` is omitted. For more information, see `http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#Temporal`.
`TransactionRequiredException`	`TransactionRequiredException`	In 10.1.3.1 release, this exception extends `PersistenceException` instead of `RuntimeException`.

Changes in oracle.toplink.essentials.platform.database

In 10.1.3.1 release, the following new classes were added:

- `DerbyPlatform`
- `JavaDBPlatform`
- `PostgreSQLPlatform`

Changes in Interceptor Support

If you are using TopLink JPA with OC4J, be aware of the fact that interceptors changed substantially between the TopLink JPA preview and the final EJB 3.0 specification.

In the final EJB 3.0 specification, interceptors and life cycle event listeners are merged and both use `javax.interceptors.Interceptors`. This will affect any code that uses interceptors or life cycle event listeners, in particular session beans and message-driven beans.

For more information, see the following:

- "What is the Life Cycle of an Enterprise Bean?" on page 1-5
- "Understanding EJB 3.0 Interceptors" on page 2-10.

Acquiring an Entity Manager

In 10.1.3.0 release, you use the `java.persistence.setup.config` property to identify a class with the list of entities that an entity manager manages.

In 10.1.3.1 release, this property is obsolete. Instead, you must define managed entity classes in a persistence unit as specified in the EJB 3.0 specification.

In 10.1.3.0 release, you inject an entity manager using the `@Resource` annotation.

In 10.1.3.1 release, you inject an entity manager using the `@PersistenceContext` annotation:

```
@PersistenceContext protected EntityManager entityManager;
```

In 10.1.3.1 release, OC4J supports the use of `@Resource` to inject an entity manager for backward compatibility. However, Oracle recommends that you use the

`@PersistenceContext` annotation instead to be compliant with the EJB 3.0 specification.

For more information, see the following:

- "What is the persistence.xml File?" on page 2-8
- "Acquiring an EntityManager" on page 29-8
- `http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#EntityManagerAnnotations`).
- `http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-extensions.html#persistence-xml`).

New JAR Files

In 10.1.3.0 release, OC4J uses the `persistence-preview.jar` and `toplink.jar` file to provide the JPA preview implementation.

In 10.1.3.1 release, OC4J uses the `persistence.jar` and the `toplink-essentials.jar` and `toplink-essentials-agent.jar` files to provide the full JPA implementation.

In your IDE, ensure that any library definitions you may have associated with your projects include only the 10.1.3.1 JPA libraries and exclude the old 10.1.3.0 libraries.

For more information about TopLink JAR files, see "JPA Persistence JAR Files" on page 3-2.

EJB 2.1 Support

In this release, OC4J supports the functionality specified in the EJB 2.1 final release specification (`http://java.sun.com/products/ejb/docs.html`).

This section describes the following:

- What JDK is Required?
- How do you Define an EJB 2.1 Module?
- How Does OC4J Manage Persistence in an EJB 2.1 Application?

What JDK is Required?

If you are using EJB 2.1, then you must use JDK 1.4 or higher.

How do you Define an EJB 2.1 Module?

By default, module version - `ejb-jar.xml` file `<ejb-jar>` element `version` attribute - is set to `2.x`.

Typically, this value changes only if you explicitly set it to `3.0` or omit the `ejb-jar.xml` file.

The CMP version - `ejb-jar.xml` file `<cmp-version>` element - is independent of the EJB module version. For EJB 2.x CMP entity beans, you set `<cmp-version>` to `2.x`.

Note that it is valid to have an EJB 3.0 module that uses both EJB 2.x CMP entity beans and EJB 3.0 entities.

For more information, see "Understanding EJB Persistence Services" on page 2-12.

How Does OC4J Manage Persistence in an EJB 2.1 Application?

OC4J delegates persistence operations to a persistence manager. In this release, OC4J uses the TopLink persistence manager by default (see "TopLink EJB 2.1 Persistence Manager" on page 3-12).

The Orion persistence manager is deprecated. Oracle recommends that you use OC4J and the TopLink persistence manager for new development. Using the migration tool (see "Migrating to the TopLink EJB 2.1 Persistence Manager" on page 3-13), you can easily migrate an existing OC4J application that uses EJB 2.0 entity beans with the Orion persistence manager to use EJB 2.0 entity beans with the TopLink persistence manager. For more information about the Orion persistence manager, see the *Oracle Containers for J2EE Orion CMP Developer's Guide*.

TopLink EJB 2.1 Persistence Manager

Oracle TopLink is an advanced, object-persistence and object-transformation framework that provides development tools and run-time capabilities that reduce development and maintenance efforts, and increase enterprise application functionality.

In this release, OC4J uses TopLink as the persistence manager for EJB 2.1 entity beans with container-managed persistence. For more information about the TopLink persistence manager, see "What is TopLink?" in the *Oracle TopLink Developer's Guide*.

OC4J provides JAR files for the classes that make up the TopLink EJB 2.1 persistence manager implementation. For more information about persistence JAR files, see "EJB 2.1 Persistence JAR Files" on page 3-12.

For EJB 2.1 projects, you use the TopLink Workbench (see "Understanding the TopLink Workbench" in the *Oracle TopLink Developer's Guide*) to configure persistence properties in the `toplink-ejb-jar.xml` file (see "What is the toplink-ejb-jar.xml File?" on page 2-6). When you migrate an Orion CMP application to TopLink persistence (see "Migrating to the TopLink EJB 2.1 Persistence Manager" on page 3-13), the TopLink migration tool automatically creates a TopLink Workbench project for you.

You can customize this configuration at run time using a TopLink customization class (see "Customizing the TopLink EJB 2.1 Persistence Manager" on page 3-13).

EJB 2.1 Persistence JAR Files

OC4J uses the TopLink JAR files that Table 3–3 lists to provide the TopLink EJB 2.1 persistence manager implementation. These JAR files are located in the `<ORACLE_HOME>/toplink/jlib` directory.

Table 3–3 TopLink JAR Files

JAR File	Contents
`antlr.jar`	This JAR contains the
`toplink.jar`	This JAR contains all the classes that comprise the TopLink API, including classes with Oracle JDBC dependencies. If you want to use an Oracle JDBC driver version different than the default version installed with OC4J, see "Associating TopLink With an Oracle JDBC Driver" on page 20-4.
`toplink-agent.jar`	This JAR contains the classes that TopLink uses to perform byte-code weaving on EJB 2.1 entity bean classes to enable transparent one-to-one and many-to-one indirection without requiring the use of a `ValueHolder`. You invoke `toplink-agent.jar` by adding `-javaagent:toplink-agent.jar` to your application's JVM command line – do not include this jar on the classpath of a TopLink application. This JAR is optional. It should never be placed on the classpath and should only be used as part of `-javaagent`.
`toplink-oc4j.jar`	This JAR contains the classes that TopLink uses to integrate with Oracle Containers for J2EE. This JAR file is only used in OC4J; the container is preconfigured to use `toplink-oc4j.jar`. In a non-OC4J application, use `toplink.jar`.

Customizing the TopLink EJB 2.1 Persistence Manager

At run time, you can access TopLink persistence manager API to take advantage of advanced TopLink features.

To access the TopLink persistence manager API in an EJB 2.1 CMP application, you can include a TopLink customization class in your deployment JAR.

This optional Java class implements `oracle.toplink.ejb.cmp.DeploymentCustomization` to allow deployment customization of TopLink mapping and run-time configuration. At deployment time, the TopLink runtime creates a new instance of this class and invokes its methods `beforeLoginCustomization` (before the TopLink runtime logs in to the session) and `afterLoginCustomization` (after the TopLink runtime logs in to the session), passing in the TopLink session as a parameter.

Use your implementation of the `beforeLoginCustomization` method to configure TopLink session attributes including cache coordination, parameterized SQL, native SQL, batch writing/batch size, byte-array/string binding, login, event listeners, table qualifier, and sequencing.

For EJB 2.1, you can use a TopLink customization class to access TopLink persistence manager API not accessible from the TopLink Workbench GUI.

For more information, see the following:

- "Configuring `pm-properties`" in the *Oracle TopLink Developer's Guide*
- *Oracle TopLink API Reference*

Migrating to the TopLink EJB 2.1 Persistence Manager

Using the TopLink migration tool, you can easily migrate an existing OC4J application that uses EJB 2.0 entity beans with the Orion persistence manager to use EJB 2.0 entity beans with the TopLink persistence manager.

For more information on using the TopLink migration tool, see "Migrating OC4J Orion Persistence to OC4J TopLink Persistence" in the *Oracle TopLink Developer's Guide*.

Part II

EJB 3.0 Session Beans

This part provides procedural information on implementing and configuring EJB 3.0 session beans. For conceptual information, see Part I, "EJB Overview".

This part contains the following chapters:

- Chapter 4, "Implementing an EJB 3.0 Session Bean"
- Chapter 5, "Using an EJB 3.0 Session Bean"

4

Implementing an EJB 3.0 Session Bean

This chapter explains how to implement an EJB 3.0 session bean, including the following:

- Implementing an EJB 3.0 Stateless Session Bean
- Implementing an EJB 3.0 Stateful Session Bean

For more information, see the following:

- "What is a Session Bean?" on page 1-27
- "Using an EJB 3.0 Session Bean" on page 5-1

Implementing an EJB 3.0 Stateless Session Bean

EJB 3.0 greatly simplifies the development of stateless session beans, removing many complex development tasks. For example:

- The bean class can be a plain old Java object (POJO); it does not need to implement `javax.ejb.SessionBean`.
- The business interface is optional.

  Home (`javax.ejb.EJBHome` and `javax.ejb.EJBLocalHome`) and component (`javax.ejb.EJBObject` and `javax.ejb.EJBLocalObject`) business interfaces are not required.

  The EJB 3.0 local or remote client of a session bean written to the EJB 3.0 API accesses a session bean through its business interface. The business interface of an EJB 3.0 session bean is an ordinary Java interface, regardless of whether or not local or remote access is provided for the bean.
- Annotations are used for many features.
- A `SessionContext` is not required: you can simply use `this` to resolve a session bean to itself.

For more information, see the following:

- "What is a Stateless Session Bean?" on page 1-28
- "Adapting an EJB 3.0 Stateless Session Bean for an EJB 2.1 Client" on page 4-4

Note: You can download an EJB 3.0 stateless session bean code example from:
http://www.oracle.com/technology/tech/java/oc4j/10131/how_to/how-to-ejb30-stateless-ejb/doc/how-to-ejb30-stateless-ejb.html.

To implement an EJB 3.0 stateless session bean, do the following:

1. Create the stateless session bean class.

   You can create a plain old Java object (POJO) and define it as a stateless session bean with the `@Stateless` annotation.

   Note: OC4J ignores the `@Stateless` attribute `mappedName`. For more information, see "OC4J Support for Annotation Attribute mappedName" on page 1-27.

   ---

2. Implement your business methods.

   Note: A stateless session bean does not need a remove method.

   ---

3. Optionally, define life cycle callback methods using the appropriate annotations.

   You do not need to define life cycle methods: OC4J provides an implementation for all such methods. Define a method of your stateless session bean class as a life cycle callback method only if you want to take some action of your own at a particular point in the stateless session bean's life cycle.

   For more information, see "Configuring a Life Cycle Callback Interceptor Method on an EJB 3.0 Session Bean" on page 5-4.

4. Optionally, define OC4J-proprietary deployment options.

   In an EJB 3.0 application, you can do this by annotating your stateless session bean class with the OC4J-proprietary `oracle.j2ee.ejb.@StatelessDeployment` annotation (see "Configuring OC4J-Proprietary Deployment Options on an EJB 3.0 Session Bean" on page 5-10).

5. Complete the configuration of your session bean (see "Using an EJB 3.0 Session Bean" on page 5-1).

Implementing an EJB 3.0 Stateful Session Bean

EJB 3.0 greatly simplifies the development of stateful session beans, removing many complex development tasks. For example:

- The bean class can be a POJO; it does not need to implement `javax.ejb.SessionBean`.
- The business interface is optional.

  Home (`javax.ejb.EJBHome` and `javax.ejb.EJBLocalHome`) and component (`javax.ejb.EJBObject` and `javax.ejb.EJBLocalObject`) business interfaces are not required.

The EJB 3.0 local or remote client of a session bean written to the EJB 3.0 API accesses a session bean through its business interface. The business interface of an EJB 3.0 session bean is an ordinary Java interface, regardless of whether or not local or remote access is provided for the bean.

- Annotations are used for many features.
- A `SessionContext` is not required: you can simply use `this` to resolve a session bean to itself.

For more information, see the following:

- "What is a Stateless Session Bean?" on page 1-28
- "Adapting an EJB 3.0 Stateful Session Bean for an EJB 2.1 Client" on page 4-5

Note: You can download an EJB 3.0 stateful session bean code example from: `http://www.oracle.com/technology/tech/java/oc4j/10131/how_to/how-to-ejb30-stateful-ejb/doc/how-to-ejb30-stateful-ejb.html`.

To implement an EJB 3.0 stateful session bean, do the following:

1. Create the stateful session bean class.

   You can create a POJO and define it as a stateful session bean with the `@Stateful` annotation.

   Note: OC4J ignores the `@Stateful` attribute `mappedName`.

   ---

2. Implement your business methods.

   To define a method of your stateful session bean class as a `remove` method, use the `@Remove` annotation.

3. Optionally, define life cycle callback methods using the appropriate annotations.

   You do not need to define life cycle methods: OC4J provides an implementation for all such methods. Define a method of your stateful session bean class as a life cycle callback method only if you want to take some action of your own at a particular point in the stateful session bean's life cycle.

   For more information, see "Configuring a Life Cycle Callback Interceptor Method on an EJB 3.0 Session Bean" on page 5-4.

4. Optionally, define OC4J-proprietary deployment options.

   In an EJB 3.0 application, you can do this by annotating your stateful session bean class with the OC4J-proprietary `oracle.j2ee.ejb.@StatefulDeployment` annotation (see "Configuring OC4J-Proprietary Deployment Options on an EJB 3.0 Session Bean" on page 5-10).

5. Complete the configuration of your session bean (see "Using an EJB 3.0 Session Bean" on page 5-1).

Adapting an EJB 3.0 Stateless Session Bean for an EJB 2.1 Client

By associating an EJB 3.0 stateless session bean with EJB 2.1 home and component interfaces (see "Using Annotations" on page 4-4), you can adapt an EJB 3.0 stateless session bean so that an EJB 2.1 client can access it.

You can use this technique to manage the incremental migration of an EJB 2.1 application to EJB 3.0 or to give existing EJB 2.1 clients access to new development that you implement using EJB 3.0.

For more information on EJB 2.1 home and component interfaces, see the following:

- "Implementing the Home Interfaces" on page 11-6
- "Implementing the Component Interfaces" on page 11-8

Using Annotations

To adapt an EJB 3.0 stateless session bean for an EJB 2.1 client, do the following:

1. Associate the EJB 2.1 home interfaces with the EJB 3.0 stateless session bean.

   Use the `@RemoteHome` annotation for remote home interfaces, and the `@LocalHome` annotation for local home interfaces:

   ```
   @Stateless
   @RemoteHome (value=Ejb21RemoteHome1.class)
   @LocalHome (value=Ejb21LocalHome.class)
   public class MyStatelessSB {
   ...
   }
   ```

   Note: You may associate a stateless session bean with at most one remote and one local home interface.

2. Consider the requirements for supporting the home interface's `create` methods.

   An EJB 3.0 stateless session bean does not require an `ejbCreate` method, even when it has a home interface. Alternatively, you may define a post-construct life cycle callback method (see "Configuring a Life Cycle Callback Interceptor Method on an EJB 3.0 Session Bean" on page 5-4).

3. Associate the EJB 2.1 component interfaces with the EJB 3.0 stateless session bean.

   Use the `@Remote` annotation for remote component interfaces, and the `@Local` annotation for local component interfaces:

   ```
   @Stateless
   @Remote (value={Ejb21Remote1.class, Ejb21Remote2.class})
   @Local (value={Ejb21Local.class})
   public class MyStatelessSB {
   ...
   }
   ```

   Note: You may associate a stateless session bean with one or more remote and local component interfaces.

Adapting an EJB 3.0 Stateful Session Bean for an EJB 2.1 Client

By associating an EJB 3.0 stateful session bean with EJB 2.1 home and component interfaces (see "Using Annotations" on page 4-5), you can adapt an EJB 3.0 stateful session bean so that an EJB 2.1 client can access it.

You can use this technique to manage the incremental migration of an EJB 2.1 application to EJB 3.0 or to give existing EJB 2.1 clients access to new development that you implement using EJB 3.0.

For more information on EJB 2.1 home and component interfaces, see: the following

- "Implementing the Home Interfaces" on page 11-6
- "Implementing the Component Interfaces" on page 11-8

Using Annotations

To adapt an EJB 3.0 stateful session bean for an EJB 2.1 client, do the following:

1. Associate the EJB 2.1 home interfaces with the EJB 3.0 stateful session bean.

   Use the `@RemoteHome` annotation for remote home interfaces, and the `@LocalHome` annotation for local home interfaces:

```
@Stateful
@RemoteHome (value=Ejb21RemoteHome1.class)
@LocalHome (value=Ejb21LocalHome.class)
public class MyStatefulSB {
...
}
```

   Note: You may associate a stateful session bean with at most one remote and one local home interface.

2. Consider the requirements for supporting the home interface's `create` methods.

   For each `create<METHOD>` in the home interfaces, implement an initialization method in your EJB 3.0 stateful session bean with the same signature (number, order, and type of arguments), and annotate the method with `@Init`:

```
@Stateful
@RemoteHome (value=Ejb21RemoteHome1.class)
@LocalHome (value=Ejb21LocalHome.class)
public class MyStatefulSB {
    private String message;
    private String name;
...
    // Corresponds to home interface method create()

    @Init
    public void initDefault() throws CreateException {
        this.message = "Default Message";
        this.name = "Default Name";
    }

    // Corresponds to home interface method createWithMessage(String)

    @Init
    public void initWithMsg(String message) throws CreateException {
```

```
        this.message = message;
    }

    // Corresponds to home interface method createWithName(String)
    // Use @Init attribute value to disambiguate createWithName(String)
    // from createWithMessage(String).

    @Init(value="createWithName")
    public void initWithName(String message) throws CreateException {
        this.name = name;
    }

...
}
```

Initialization methods may have any method name. OC4J matches a home interface `create<METHOD>` to a stateful session bean initialization method by signature. Alternatively, you can use `@Init` attribute `value` to explicitly specify the name of the home interface `create<METHOD>`. This is useful when two or more home interface `create<METHOD>` methods have the same signature.

Initialization methods are invoked after the post-construct life cycle method is invoked, if present (see "Configuring a Life Cycle Callback Interceptor Method on an EJB 3.0 Session Bean" on page 5-4).

3. Associate the EJB 2.1 component interfaces with the EJB 3.0 stateful session bean.

   Use the `@Remote` annotation for remote component interfaces, and the `@Local` annotation for local component interfaces:

```
@Stateful
@Remote (value={Ejb21Remote1.class, EJB21Remote2.class})
@Local (value={Ejb21Local.class})
public class MyStatefulSB {
...
}
```

> **Note:** You may associate a stateful session bean with one or more remote and local component interfaces.

5

Using an EJB 3.0 Session Bean

This chapter describes the various options that you must configure in order to use an EJB 3.0 session bean.

Table 5–1 lists these options and indicates which are basic (applicable to most applications), and which are advanced (applicable to more specialized applications) The table also indicates which options are applicable to stateless session beans, and which are applicable to stateful session beans.

For more information, see the following:

- "What is a Session Bean?" on page 1-27
- "Implementing an EJB 3.0 Session Bean" on page 4-1

Table 5–1 Configurable Options for an EJB 3.0 Session Bean

Options	Stateless	Stateful	Type
"Configuring Passivation" on page 5-1		✔	Advanced
"Configuring Passivation Criteria" on page 5-2		✔	Advanced
"Configuring Passivation Location" on page 5-3		✔	Advanced
"Configuring Bean Instance Pool Size" on page 31-4	✔	✔	Basic
"Configuring Bean Instance Pool Timeouts for Session Beans" on page 31-6	✔	✔	Advanced
"Configuring a Transaction Timeout for a Session Bean" on page 21-6	✔	✔	Advanced
"Configuring a Life Cycle Callback Interceptor Method on an EJB 3.0 Session Bean" on page 5-4	✔	✔	Basic
"Configuring a Life Cycle Callback Interceptor Method on an Interceptor Class of an EJB 3.0 Session Bean" on page 5-5	✔	✔	Basic
"Configuring an Around Invoke Interceptor Method on an EJB 3.0 Session Bean" on page 5-6	✔	✔	Advanced
"Configuring an Around Invoke Interceptor Method on an Interceptor Class of an EJB 3.0 Session Bean" on page 5-7	✔	✔	Advanced
"Configuring an Interceptor Class for an EJB 3.0 Session Bean" on page 5-8	✔	✔	Advanced
"Configuring OC4J-Proprietary Deployment Options on an EJB 3.0 Session Bean" on page 5-10	✔	✔	Advanced

Configuring Passivation

You can enable and disable passivation for stateful session beans using the `server.xml` file (see "Using Deployment XML" on page 5-2).

You may choose to disable passivation for any of the following reasons:

- Incompatible object types: if you cannot represent the nontransient attributes of your stateful session bean with object types supported by passivation (see "What Object Types can be Passivated?" on page 1-33), you can exchange increased memory consumption for the use of other object types by disabling passivation.
- Performance: if you determine that passivation is a performance problem in your application, you can exchange increased memory consumption for improved performance by disabling passivation.
- Secondary storage limitations: if you cannot provide sufficient secondary storage (see "Configuring Passivation Location" on page 5-3), you can exchange increased memory consumption for reduced secondary storage requirements by disabling passivation.

For more information, see the following:

- "When Does Stateful Session Bean Passivation Occur?" on page 1-32
- "Configuring Passivation Criteria" on page 5-2
- "Configuring Passivation Location" on page 5-3

Using Deployment XML

For an EJB 3.0 stateful session bean, you configure passivation in the `server.xml` file as you would for an EJB 2.1 stateful session bean (see "Using Deployment XML" on page 12-2).

Configuring Passivation Criteria

You can specify under what conditions OC4J passivates an EJB 3.0 stateful session bean using OC4J-proprietary annotations (see "Using Annotations" on page 5-2) or using the `orion-ejb-jar.xml` file (see "Using Deployment XML" on page 5-3).

Configuration in the `orion-ejb-jar.xml` file overrides the corresponding configuration made using OC4J-proprietary annotations.

For more information, see the following:

- "When Does Stateful Session Bean Passivation Occur?" on page 1-32
- "Configuring Passivation" on page 5-1
- "Configuring Passivation Location" on page 5-3

Using Annotations

You can specify OC4J-proprietary deployment options for an EJB 3.0 stateful session bean using the `@StatefulDeployment` OC4J-proprietary annotation. Example 5–1 shows how to configure passivation criteria for an EJB 3.0 stateless session bean using the following `@StatefulDeployment` annotation attributes:

- `idletime`
- `memoryThreshold`
- `maxInstances`
- `maxInstancesThreshold`
- `passivateCount`
- `resourceCheckInterval`

For more information on these `@StatefulDeployment` attributes, see Table A–1. For more information on the `@StatefulDeployment` annotation, see "Configuring OC4J-Proprietary Deployment Options on an EJB 3.0 Session Bean" on page 5-10.

Example 5–1 Configuring Passivation Criteria Using @StatefulDeployment

```
import javax.ejb.Stateless;
import oracle.j2ee.ejb.StatelessDeployment;

@Stateless
@StatefulDeployment(
    idletime=100,
    memoryThroshold=90,
    maxInstances=10,
    maxInstancesThreshold=80,
    passivateCount=3,
    resourceCheckInterval=90
)
public class HelloWorldBean implements HelloWorld {
    public void sayHello(String name) {
        System.out.println("Hello "+name +" from first EJB3.0");
    }
}
```

Using Deployment XML

For an EJB 3.0 stateful session bean, you configure passivation criteria in the `orion-ejb-jar.xml` file as you would for an EJB 2.1 stateful session bean (see "Using Deployment XML" on page 12-2).

Configuring Passivation Location

You can specify the directory and file name to which OC4J serializes an EJB 3.0 stateful session bean when passivated using OC4J-proprietary annotations (see "Using Annotations" on page 5-3) or using the `orion-ejb-jar.xml` file (see "Using Deployment XML" on page 5-4).

For more information, see the following:

- "Where is a Passivated Stateful Session Bean Stored?" on page 1-34
- "Configuring Passivation" on page 5-1
- "Configuring Passivation Criteria" on page 5-2

Using Annotations

You can specify OC4J-proprietary deployment options for an EJB 3.0 stateful session bean using the `@StatefulDeployment` OC4J-proprietary annotation. Example 5–1 shows how to configure the passivation location for an EJB 3.0 stateless session bean using the `@StatefulDeployment` annotation `persistenceFileName` attribute.

For more information on this `@StatefulDeployment` attribute, see Table A–1. For more information on the `@StatefulDeployment` annotation, see "Configuring OC4J-Proprietary Deployment Options on an EJB 3.0 Session Bean" on page 5-10.

Example 5–2 Configuring Passivation Location Using @StatefulDeployment

```
import javax.ejb.Stateless;
import oracle.j2ee.ejb.StatelessDeployment;
```

```
@Stateless
@StatefulDeployment(
    persistenceFileNazme="C:\sfsb\sfsb.persistence",
)
public class HelloWorldBean implements HelloWorld {
    public void sayHello(String name) {
        System.out.println("Hello "+name +" from first EJB3.0");
    }
}
```

Using Deployment XML

For an EJB 3.0 stateful session bean, you configure passivation location in the `orion-ejb-jar.xml` file as you would for an EJB 2.1 stateful session bean (see "Using Deployment XML" on page 12-3).

Configuring a Life Cycle Callback Interceptor Method on an EJB 3.0 Session Bean

You can specify an EJB 3.0 session bean class method as a callback interceptor method for any of the following life cycle events (see "Using Annotations" on page 5-4):

- Post-construct
- Pre-destroy
- Pre-passivate (stateful session beans only)
- Post-activate (stateful session beans only)

> **Note:** Do not specify pre-passivate or post-activate life cycle callback methods on a stateless session bean.

The session bean class life cycle callback method must have the following signature:

```
void <METHOD>()
```

You can also specify one or more life cycle callback methods on an interceptor class that you associate with an EJB 3.0 session bean (see "Configuring a Life Cycle Callback Interceptor Method on an Interceptor Class of an EJB 3.0 Session Bean" on page 5-5).

For more information, see the following:

- "What is the Stateless Session Bean Life Cycle?" on page 1-28
- "What is the Life Cycle of a Stateful Session Bean?" on page 1-30
- "Life Cycle Callback Methods on a Bean Class" on page 1-6

Using Annotations

You can specify an EJB 3.0 session bean class method as a life cycle callback method using any of the following annotations:

- `@PostConstruct`
- `@PreDestroy`
- `@PrePassivate` (stateful session beans only)
- `@PostActivate` (stateful session beans only)

Example 5–3 shows how to use the `@PostConstruct` annotation to specify EJB 3.0 stateful session bean class method `initialize` as a life cycle callback method.

Example 5–3 @PostConstruct

```
@Stateful
public class CartBean implements Cart {
    private ArrayList items;

    @PostConstruct
    public void initialize() {
        items = new ArrayList();
    }
...
}
```

Configuring a Life Cycle Callback Interceptor Method on an Interceptor Class of an EJB 3.0 Session Bean

You can designate an interceptor method on an interceptor class of an EJB 3.0 session bean as a life cycle callback interceptor method.

To configure a life cycle callback interceptor method on an interceptor class, you must do the following:

1. Create an interceptor class.

   This can be any POJO class.

2. Implement the life cycle callback interceptor method.

   Callback methods defined on a bean's interceptor class have the following signature:

   ```
   Object <METHOD>(InvocationContext)
   ```

3. Associate a life cycle event with the callback interceptor method (see "Using Annotations" on page 5-5).

   A life cycle event can only be associated with one callback interceptor method, but a life cycle callback interceptor method may be used to interpose on multiple callback events. For example, `@PostConstruct` and `@PreDestroy` may appear only once in an interceptor class, but you may associate both `@PostConstruct` and `@PreDestroy` with the same callback interceptor method.

4. Associate the interceptor class with your EJB 3.0 session bean (see "Configuring an Interceptor Class for an EJB 3.0 Session Bean" on page 5-8).

For more information, see the following:

- "What is the Stateless Session Bean Life Cycle?" on page 1-28
- "What is the Life Cycle of a Stateful Session Bean?" on page 1-30
- "Life Cycle Callback Interceptor Methods on an EJB 3.0 Interceptor Class" on page 1-6

Using Annotations

You can specify an interceptor class method as an EJB 3.0 session bean life cycle callback method using any of the following annotations:

- `@PostConstruct`
- `@PreDestroy`
- `@PrePassivate` (stateful session beans only)
- `@PostActivate` (stateful session beans only)

Example 5–4 shows an interceptor class for a stateful session bean. It designates method `myPrePassivateInterceptorMethod` as the life cycle callback interceptor method for the pre-passivate life cycle event using the `@PrePassivate` annotation. It also designates method `myPostConstructInterceptorMethod` as the life cycle callback interceptor method for both the post-construct and post-activate life cycle events using the `@PostConstruct` and `@PostActivate` annotations. OC4J invokes the appropriate life cycle method only when the appropriate life cycle event occurs. OC4J invokes all other non-life cycle interceptor methods (such as `myInterceptorMethod`) each time you invoke a session bean business method (see "Configuring an Interceptor Class for an EJB 3.0 Session Bean" on page 5-8).

Example 5–4 Interceptor Class

```
public class MyStatefulSessionBeanInterceptor {
    ...
    protected void myInterceptorMethod (InvocationContext ctx) {
        ...
        ctx.proceed();
        ...
    }

    @PostConstruct
    @PostActivate
    protected void myPostConstructInterceptorMethod (InvocationContext ctx) {
        ...
        ctx.proceed();
        ...
    }
    @PrePassivate
    protected void myPrePassivateInterceptorMethod (InvocationContext ctx) {
        ...
        ctx.proceed();
        ...
    }
}
```

Configuring an Around Invoke Interceptor Method on an EJB 3.0 Session Bean

You can specify one nonbusiness method as the interceptor method for a stateless or stateful session bean. Each time a client invokes a session bean business method, OC4J intercepts the invocation and invokes the interceptor method. The client invocation proceeds only if the interceptor method returns `InvocationContext.proceed()`.

An interceptor method has the following signature:

```
Object <METHOD>(InvocationContext) throws Exception
```

An interceptor method may have public, private, protected, or package level access, but must not be declared as final or static.

You can specify this method on the EJB 3.0 session bean class (see "Using Annotations" on page 5-7) or on an interceptor class that you associate with an EJB 3.0 session bean

(see "Configuring an Around Invoke Interceptor Method on an Interceptor Class of an EJB 3.0 Session Bean" on page 5-7).

For more information, see "Understanding EJB 3.0 Interceptors" on page 2-10.

Using Annotations

Example 5–5 shows how to designate a method of a session bean class as an interceptor method using the `@AroundInvoke` annotation. Each time a client invokes a business method of this stateless session bean, OC4J intercepts the invocation and invokes the interceptor method `myInterceptor`. The client invocation proceeds only if the interceptor method returns `InvocationContext.proceed()`.

Example 5–5 @AroundInvoke in an EJB 3.0 Session Bean

```
@Stateless
public class HelloWorldBean implements HelloWorld  {
    public void sayHello()  {
        System.out.println("Hello!");
    }

    @AroundInvoke
    protected Object myInterceptor(InvocationContext ctx) throws Exception {
        Principal p = ctx.getEJBContext().getCallerPrincipal;
        if (!userIsValid(p)) {
            throw new SecurityException(
                "Caller: '" + p.getName() +
                "' does not have permissions for method " + ctx.getMethod()
            );
        }
        return ctx.proceed();
    }
}
```

Configuring an Around Invoke Interceptor Method on an Interceptor Class of an EJB 3.0 Session Bean

You can specify one nonbusiness method as the interceptor method for a stateless or stateful session bean. Each time a client invokes a session bean business method, OC4J intercepts the invocation and invokes the interceptor method. The client invocation proceeds only if the interceptor method returns `InvocationContext.proceed()`.

You can specify this method on an interceptor class that you associate with an EJB 3.0 session bean or on the EJB 3.0 session bean class itself (see "Configuring an Around Invoke Interceptor Method on an EJB 3.0 Session Bean" on page 5-6).

To configure an interceptor method on an interceptor class, you must do the following:

1. Create an interceptor class.

   This can be any POJO class.

2. Implement the interceptor method.

   An interceptor method has the following signature:

   ```
   Object <METHOD>(InvocationContext) throws Exception
   ```

   An interceptor method may have public, private, protected, or package level access but must not be declared as final or static.

3. Designate the method as the interceptor method (see "Using Annotations" on page 5-8).
4. Associate the interceptor class with your EJB 3.0 session bean (see "Configuring an Interceptor Class for an EJB 3.0 Session Bean" on page 5-8).

For more information, see "Understanding EJB 3.0 Interceptors" on page 2-10.

Using Annotations

Example 5–6 shows how to specify interceptor class method `myInterceptor` as the interceptor method of an EJB 3.0 session bean using the `@AroundInvoke` annotation. After you associate this interceptor class with a session bean ("Configuring an Interceptor Class for an EJB 3.0 Session Bean" on page 5-8), each time you invoke a session bean business method, OC4J intercepts the invocation and invokes the `myInterceptor` method. The client invocation proceeds only if this method returns `InvocationContext.proceed()`.

Example 5–6 Interceptor Class

```
public class MyInterceptor {
    ...
    @AroundInvoke
    protected Object myInterceptor(InvocationContext ctx) throws Exception {
        Principal p = ctx.getEJBContext().getCallerPrincipal;
        if (!userIsValid(p)) {
            throw new SecurityException(
                "Caller: '" + p.getName() +
                "' does not have permissions for method " + ctx.getMethod()
            );
        }
        return ctx.proceed();
    }

    @PreDestroy
    public void myPreDestroyMethod (InvocationContext ctx) {
        ...
        ctx.proceed();
        ...
    }
}
```

Configuring an Interceptor Class for an EJB 3.0 Session Bean

An interceptor class is a class, distinct from the bean class itself, whose methods are invoked in response to business method invocations and life cycle events on the bean. You can associate a bean class can with any number of interceptor classes.

You can associate an interceptor class with an EJB 3.0 stateless or stateful session bean.

To configure an EJB 3.0 session bean with an interceptor class, you must do the following:

1. Create an interceptor class (see "Creating an Interceptor Class" on page 5-9).

   This can be any POJO class.
2. Implement interceptor methods in the interceptor class.

   An interceptor method has the following signature:

   ```
   Object <METHOD>(InvocationContext) throws Exception
   ```

An interceptor method may have public, private, protected, or package level access, but must not be declared as final or static.

You can annotate an interceptor method as a life cycle callback (see "Configuring a Life Cycle Callback Interceptor Method on an Interceptor Class of an EJB 3.0 Session Bean" on page 5-5) or as an `AroundInvoke` method (see "Configuring an Around Invoke Interceptor Method on an Interceptor Class of an EJB 3.0 Session Bean" on page 5-7).

3. Associate the interceptor class with your EJB 3.0 session bean (see "Associating an Interceptor Class With a Session Bean" on page 5-10).
4. Optionally configure the session bean to use singleton interceptors (see "Specifying Singleton Interceptors in a Session Bean" on page 5-10).

For more information, see "Understanding EJB 3.0 Interceptors" on page 2-10.

Using Annotations

This section describes the following:

- Creating an Interceptor Class
- Associating an Interceptor Class With a Session Bean
- Specifying Singleton Interceptors in a Session Bean

Creating an Interceptor Class

Example 5–7 shows how to specify an `AroundInvoke` interceptor method and a life cycle callback interceptor method in an interceptor class for an EJB 3.0 session bean. After you associate this interceptor class with a session bean (see Example 5–8), each time you invoke a session bean business method, OC4J invokes the `AroundInvoke` method `myInterceptor`. When the appropriate life cycle event occurs, OC4J invokes the corresponding life cycle callback interceptor method such as `myPreDestroyMethod`.

Example 5–7 Interceptor Class

```
public class MyInterceptor {
    ...
    @AroundInvoke
    protected Object myInterceptor(InvocationContext ctx) throws Exception {
        Principal p = ctx.getEJBContext().getCallerPrincipal;
        if (!userIsValid(p)) {
            throw new SecurityException(
                "Caller: '" + p.getName() +
                "' does not have permissions for method " + ctx.getMethod()
            );
        }
        return ctx.proceed();
    }

    @PreDestroy
    public void myPreDestroyMethod (InvocationContext ctx) {
        ...
        ctx.proceed();
        ...
    }
}
```

Associating an Interceptor Class With a Session Bean

You can associate an interceptor class with an EJB 3.0 session bean using the `@Interceptors` annotation. Example 5–8 shows how to associate the interceptor class from Example 5–7 with an EJB 3.0 session bean class.

Note that the life cycle method for `@PostConstruct` is a method of the EJB 3.0 session bean class itself (for more information, see "Configuring a Life Cycle Callback Interceptor Method on an EJB 3.0 Session Bean" on page 5-4), while the life cycle method for `@PreDestroy` is a life cycle callback interceptor method on the interceptor class associated with this session bean (see "Configuring a Life Cycle Callback Interceptor Method on an Interceptor Class of an EJB 3.0 Session Bean" on page 5-5).

Example 5–8 Associating an Interceptor Class With an EJB 3.0 Session Bean

```
@Stateful
@Interceptors(MyInterceptor.class)
public class CartBean implements Cart {
    private ArrayList items;

    @PostConstruct
    public void initialize() {
        items = new ArrayList();
    }
    ...
}
```

Specifying Singleton Interceptors in a Session Bean

As Example 5–9 shows, you can configure OC4J to use singleton interceptor classes by setting the `@StatelessDeployment` or `@StatefulDeployment` attribute `interceptorType` to `singleton`. All instances of this session bean will share the same instance of `MyInterceptor`. The `MyInterceptor` class must be stateless.

For more information about this attribute, see Table A–1. For more information on singleton interceptors, see "Singleton Interceptors" on page 2-12.

Example 5–9 Specifying a Singleton Interceptor Class with an EJB 3.0 Stateful Session Bean

```
@Stateful
@StatefulDeployment(interceptorType="singleton")
@Interceptors(MyInterceptor.class)
public class CartBean implements Cart {
    private ArrayList items;

    @PostConstruct
    public void initialize() {
        items = new ArrayList();
    }
    ...
}
```

Configuring OC4J-Proprietary Deployment Options on an EJB 3.0 Session Bean

You can configure OC4J-proprietary deployment options for an EJB 3.0 session bean using OC4J-proprietary annotations (see "Using Annotations" on page 5-11) or using the `orion-ejb-jar.xml` file (see "Using Deployment XML" on page 5-11).

Configuration in the `orion-ejb-jar.xml` file overrides the corresponding configuration made using OC4J-proprietary annotations.

Using Annotations

You can specify OC4J-proprietary deployment options for an EJB 3.0 session bean using the following OC4J-proprietary annotations:

- `@StatelessDeployment`: for stateless session beans.
- `@StatefulDeployment`: for stateful session beans.

Example 5–10 shows how to configure OC4J-proprietary deployment options for an EJB 3.0 stateless session bean using the `@StatelessDeployment` annotation.

For more information on `@StatelessDeployment` attributes, see Table A–1.

Example 5–10 @StatelessDeployment

```
import javax.ejb.Stateless;
import oracle.j2ee.ejb.StatelessDeployment;

@Stateless
@StatelessDeployment(
    minInstances=5,
    poolCacheTimeout=90
)
public class HelloWorldBean implements HelloWorld {
    public void sayHello(String name) {
        System.out.println("Hello "+name +" from first EJB3.0");
    }
}
```

Example 5–11 shows how to configure OC4J-proprietary deployment options for an EJB 3.0 stateful session bean using the `@StatefulDeployment` annotation.

For more information on `@StatefulDeployment` attributes, see Table A–1.

Example 5–11 @StatefulDeployment

```
import javax.ejb.Stateful
import oracle.j2ee.ejb.StatefulDeployment;

@Stateful
@StatefulDeployment(
    idletime=100
    passivateCount=3
)
public class CartBean implements Cart {
    private ArrayList items;
    ...
}
```

Using Deployment XML

You can specify OC4J-proprietary deployment options using the `orion-ejb-jar.xml` file element `<session-deployment>` as Example 5–12 shows.

For more information on the `<session-deployment>` element, see "<session-deployment>" on page A-4.

Example 5–12 orion-ejb-jar.xml File <session-deployment> Element

```
<?xml version="1.0" encoding="utf-8"?>
<orion-ejb-jar
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/orion-ejb-jar-10_
0.xsd"
    deployment-version="10.1.3.1.0"
    deployment-time="10b1fb5cdd0"
    schema-major-version="10"
    schema-minor-version="0"
>
    <enterprise-beans>
        <session-deployment
            name="MBeanServerEjb"
            call-timeout="0"
            location="MBeanServerEjb"
            local-location="admin_ejb_MBeanServerEjbLocal"
            timeout="0"
            ...
        >
        </session-deployment>
    ...
    </enterprise-beans>
    ...
</orion-ejb-jar>
```

Part III

JPA Entities

This part provides procedural information on implementing and configuring JPA entities and JPA entity queries. For conceptual information, see Part I, "EJB Overview".

This part contains the following chapters:

- Chapter 6, "Implementing a JPA Entity"
- Chapter 7, "Using Java Persistence API"
- Chapter 8, "Implementing JPA Queries"

6

Implementing a JPA Entity

This chapter explains how to implement a JPA entity.

For more information, see the following:

- "What is a JPA Entity?" on page 1-34
- "Using Java Persistence API" on page 7-1

Implementing a JPA Entity

EJB 3.0 greatly simplifies the development of enterprise beans, removing many complex development tasks. For example:

- The bean class can be a POJO; it does not need to implement `javax.ejb.EntityBean`.
- The business interface is optional. It can be a plain old Java interface (POJI).

  Home (`javax.ejb.EJBHome` and `javax.ejb.EJBLocalHome`) and component (`javax.ejb.EJBObject` and `javax.ejb.EJBLocalObject`) business interfaces are not required.
- Annotations are used for many features, including container-managed relationships (object-relational mapping).
- An `EntityContext` is not required: you can simply use `this` to resolve an entity to itself.

For more information, see "What is a JPA Entity?" on page 1-34.

> **Note:** You can download a JPA entity code example from: `http://www.oracle.com/technology/tech/java/oc4j/10131/how_to/how-to-ejb30-entity-ejb/doc/how-to-ejb30-entity-ejb.html`.

To implement a JPA entity, do the following:

1. Create the entity bean class.

   You can create a POJO and define it as an entity bean with container-managed persistence using the `@Entity` annotation.

   All data members are by default considered container-managed persistent fields, unless annotated with `@Transient`.

2. Define how OC4J persists your entity bean class to a database using the `@Table` and `@Column` annotations.

   If you do not have an existing database schema, you can delegate table and column definition to OC4J by omitting these annotations: at deployment time, OC4J will create default table and column names based on class and data member names.

   For more information, see "Configuring Table and Column Information" on page 7-6.

3. Define one data member as the primary key field with the `@Id` annotation.

   You can annotate the data member itself or its getter method. For more information, see "Configuring a JPA Entity Primary Key" on page 7-1.

4. Define container-managed relationships using the appropriate object-relational mapping annotations, such as `@OneToOne`.

   For more information, see "Configuring a Container-Managed Relationship Field for a JPA Entity" on page 7-9.

5. Optionally, define finders and queries using the `@NamedQuery` annotation.

   At run time, you can use the predefined finders (see "Predefined TopLink Finders" on page 1-53) and default finders (see "Default TopLink Finders" on page 1-54) that the TopLink persistence manager provides.

   For more information, see "Implementing JPA Queries" on page 8-1.

6. Optionally, define life cycle callback methods using the appropriate annotations.

   You do not need to define life cycle methods: OC4J provides an implementation for all such methods. Define a method of your entity bean class as a life cycle callback method only if you want to take some action of your own at a particular point in the entity bean's life cycle.

   For more information, see "Configuring a Life Cycle Callback Method on a JPA Entity" on page 7-16.

7. Complete the configuration of your entity bean (see "Using Java Persistence API" on page 7-1).

7

Using Java Persistence API

This chapter describes the various options that you can configure in order to use a JPA entity.

Table 7–1 lists these options and indicates which are basic (applicable to most applications) and which are advanced (applicable to more specialized applications).

For more information, see the following:

- "What is a JPA Entity?" on page 1-34
- "Implementing a JPA Entity" on page 6-1

Table 7–1 Configurable Options for a JPA Entity

Options	Type
"Configuring a JPA Entity Primary Key" on page 7-1	Basic
"Configuring Table and Column Information" on page 7-6	Basic
"Configuring a Container-Managed Relationship Field for a JPA Entity" on page 7-9	Basic
"Configuring a Basic Mapping" on page 7-10	Basic
"Configuring a Large Object Mapping" on page 7-10	Advanced
"Configuring a Serialized Object Mapping" on page 7-11	Advanced
"Configuring an One-to-One Mapping" on page 7-11	Basic
"Configuring a Many-to-One Mapping" on page 7-12	Basic
"Configuring an One-to-Many Mapping" on page 7-12	Basic
"Configuring a Many-to-Many Mapping" on page 7-13	Basic
"Configuring an Aggregate Mapping" on page 7-14	Advanced
"Configuring Optimistic Lock Version Field" on page 7-15	Advanced
"Implementing JPA Queries" on page 8-1	Basic
"Configuring Inheritance for a JPA Entity" on page 7-19	Advanced
"Configuring Lazy Loading" on page 7-16	Basic
"Configuring a Life Cycle Callback Method on a JPA Entity" on page 7-16	Advanced
"Configuring a Life Cycle Callback Listener Method on an Entity Listener Class of a JPA Entity" on page 7-17	Advanced

Configuring a JPA Entity Primary Key

Every JPA entity must have a primary key.

You can specify a primary key as a single primitive, or JDK object type entity field (see "Configuring a JPA Entity Simple Primary Key Field" on page 7-2).

You can specify a composite primary key made up of one or more primitive, or JDK object types using a separate composite primary key class (see "Configuring a JPA Entity Composite Primary Key Class" on page 7-2).

You can either assign primary key values yourself, or you can associate a primary key field with a primary key value generator (see "Configuring JPA Entity Automatic Primary Key Generation" on page 7-5).

Configuring a JPA Entity Simple Primary Key Field

The simplest primary key is one you specify as a single primitive or JDK object type entity field (see "Using Annotations" on page 7-2).

Note: For a JPA entity primary key field code example, see:
`http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30mappingannotations/doc/how-to-ejb30-mapping-annotations.html#id`

Using Annotations

Example 7–1 shows how to use the `@Id` annotation to specify an entity field as the primary key. In this example, primary key values are generated using a table generator (see "Configuring JPA Entity Automatic Primary Key Generation" on page 7-5).

Example 7–1 Primary Key Using @Id

```
@Id(generate=TABLE, generator="ADDRESS_TABLE_GENERATOR")
@TableGenerator(
    name="ADDRESS_TABLE_GENERATOR",
    tableName="EMPLOYEE_GENERATOR_TABLE",
    pkColumnValue="ADDRESS_SEQ"
)
@Column(name="ADDRESS_ID")
public Integer getId() {
    return id;
}
```

Configuring a JPA Entity Composite Primary Key Class

A composite primary key is usually made up of two or more primitive or JDK object types. Composite primary keys typically arise when mapping from legacy databases when the database key is comprised of several columns. You can specify such a composite primary key with a separate composite primary key class (see "Using Annotations" on page 7-3)

A composite primary key class has the following characteristics:

- It is a POJO class.
- It must be public and must have a public no-argument constructor.
- If you use property-based access, the properties of the primary key class must be public or protected.
- It must be serializable.
- It must define `equals` and `hashCode` methods.

  The semantics of value equality for these methods must be consistent with the database equality for the database types to which the key is mapped.

You can make the composite primary key class either an embedded class owned by the entity class, or a nonembedded class whose fields you map to multiple fields or properties of the entity class. In the latter case, the names of primary key fields or properties in the composite primary key class and those of the entity class must correspond and their types must be the same.

Using Annotations

Example 7–2 shows a typical embeddable composite primary key class. Example 7–3 shows how to configure a JPA entity with this embedded composite primary key class using the `@EmbeddedId` annotation.

Example 7–2 Embeddable Composite Primary Key Class

```
@Embeddable
public class EmployeePK implements Serializable {
    private String name;
    private long id;

    public EmployeePK() {
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public int hashCode() {
        return (int) name.hashCode() + id;
    }

    public boolean equals(Object obj) {
        if (obj == this) return true;
        if (!(obj instanceof EmployeePK)) return false;
        if (obj == null) return false;
        EmployeePK pk = (EmployeePK) obj;
        return pk.id == id && pk.name.equals(name);
    }
}
```

Example 7–3 JPA Entity With an Embedded Composite Primary Key Class

```
@Entity
public class Employee implements Serializable {
    EmployeePK primaryKey;

    public Employee() {
    }

    @EmbeddedId
    public EmployeePK getPrimaryKey() {
        return primaryKey;
```

```
    }

    public void setPrimaryKey(EmployeePK pk) {
        primaryKey = pk;
    }

    ...
}
```

Example 7–5 shows a nonembedded composite primary key class. In this class, fields `empName` and `birthDay` must correspond in name and type to properties in the entity class. Example 7–5 shows how to configure a JPA entity with this nonembedded composite primary key class using the `@IdClass` annotation. Because entity class fields `empName` and `birthDay` are used in the primary key, you must also annotate them using the `@Id` annotation.

Example 7–4 Non-Embedded Composite Primary Key Class

```
public class EmployeePK implements Serializable {
    private String empName;
    private Date birthDay;

    public EmployeePK() {
    }

    public String getName() {
        return empName;
    }

    public void setName(String name) {
        empName = name;
    }

    public long getDateOfBirth() {
        return birthDay;
    }

    public void setDateOfBirth(Date date) {
        birthDay = date;
    }

    public int hashCode() {
        return (int) empName.hashCode();
    }

    public boolean equals(Object obj) {
        if (obj == this) return true;
        if (!(obj instanceof EmployeePK)) return false;
        if (obj == null) return false;
        EmployeePK pk = (EmployeePK) obj;
        return pk.birthDay == birthDay && pk.empName.equals(empName);
    }
}
```

Example 7–5 JPA Entity With a Mapped Composite Primary Key Class

```
@IdClass(EmployeePK.class)
@Entity
public class Employee {
    @Id String empName;
    @Id Date birthDay;
...
}
```

Configuring JPA Entity Automatic Primary Key Generation

Typically, you associate a primary key field (see "Configuring a JPA Entity Simple Primary Key Field") with a primary key value generator so that when an entity instance is created, a new, unique primary key value is assigned automatically.

Table 7–2 lists the types of primary key value generators that you can define.

Table 7–2 JPA Entity Primary Key Value Generators

Type	Description	For more information, see ...
Generated Id Table	A database table that the container uses to store generated primary key values for entities. Typically shared by multiple entity types that use table-based primary key generation. Each entity type will typically use its own row in the table to generate the primary key values for that entity class. Primary key values are positive integers.	"Table Sequencing" in the *Oracle TopLink Developer's Guide*
Table Generator	A primary key generator, which you can reference by name, defined at one of the package, class, method, or field level. The level at which you define it will depend upon the desired visibility and sharing of the generator. No scoping or visibility rules are actually enforced. Oracle recommends that you define the generator at the level for which it will be used. This generator is based on a database table.	"Table Sequencing" in the *Oracle TopLink Developer's Guide*
Sequence Generator	A primary key generator which you can reference by name, defined at one of the package, class, method, or field level. The level, at which you define it, will depend upon the desired visibility and sharing of the generator. No scoping or visibility rules are actually enforced. Oracle recommends that you define the generator at the level for which it will be used. This generator is based on a sequence object that the database server provides.	"Native Sequencing With an Oracle Database Platform" in the *Oracle TopLink Developer's Guide* "Native Sequencing With a Non-Oracle Database Platform" in the *Oracle TopLink Developer's Guide*

Note: For an EJB 3.0 automatic primary key generation code example, see:
`http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30mappingannotations/doc/how-to-ejb30-mapping-annotations.html#sequencing`

Using Annotations

Example 7–6 shows how to use the `@TableGenerator` annotation to specify a primary key value generator based on a database table. The TopLink JPA persistence provider will attempt to create this table at deployment time: if it cannot, then you must follow your database documentation to ensure that this table exists before deployment. When a new instance of `Address` is created, a new value for entity field `id` is obtained from `ADDRESS_GENERATOR_TABLE`. In this case, you must set the `@GeneratedValue` annotation attribute `strategy` to `TABLE` and `generator` to `ADDRESS_TABLE_GENERATOR`.

Example 7–6 GeneratedValue Strategy Table: @TableGenerator

```
@Entity
@Table(name="EJB_ADDRESS")
public class Address implements Serializable {
...
    @TableGenerator(
        name="ADDRESS_TABLE_GENERATOR",
        tableName="ADDRESS_GENERATOR_TABLE",
        pkColumnValue="ADDRESS_SEQ"
    )
    @Id @GeneratedValue(strategy="TABLE", generator="ADDRESS_TABLE_GENERATOR")
```

```
    @Column(name="ADDRESS_ID")
    public Integer getId() {
        return id;
    }
...
}
```

Example 7–7 shows how to use the `@SequenceGenerator` annotation to specify a primary key value generator based on a sequence object provided by the database. The TopLink JPA persistence provider will attempt to create this object at deployment time: if it cannot, then you must follow your database documentation to ensure that this sequence object exists before deployment. When a new instance of `Address` is created, a new value for entity field `id` is obtained from database sequence object `ADDRESS_SEQ`. In this case, you must set the `@GeneratedValue` annotation attribute `strategy` to `SEQUENCE` and `generator` to `ADDRESS_SEQUENCE_GENERATOR`.

Example 7–7 GeneratedValue Strategy Sequence: @SequenceGenerator

```
@Entity
@Table(name="EJB_ADDRESS")
public class Address implements Serializable {
...
    @SequenceGenerator(
        name="ADDRESS_SEQUENCE_GENERATOR",
        sequenceName="ADDRESS_SEQ"
    )
    @Id @GeneratedValue(strategy="SEQUENCE", generator="ADDRESS_SEQUENCE_GENERATOR")
    @Column(name="ADDRESS_ID")
    public Integer getId() {
        return id;
    }
...
}
```

Example 7–8 shows how to use the `@GeneratedValue` annotation to specify a primary key value generator based on a primary key identity column (autonumber column). When a new instance of `Address` is persisted, the database assigns a value to the identity column. In this case, the TopLink JPA persistence provider re-reads the inserted row and updates the in-memory `Address` entity to set `id` to this value.

Example 7–8 @GeneratedValue Strategy Identity

```
@Entity
@Table(name="EJB_ADDRESS")
public class Address implements Serializable {
...
    @Id @GeneratedValue(strategy="IDENTITY")
    public Integer getId() {
        return id;
    }
...
}
```

Configuring Table and Column Information

You can define the characteristics of the database table into which the TopLink entity manager persists your entity, including the following:

- Configuring the Primary Table
- Configuring a Secondary Table

- Configuring a Column
- Configuring a Join Column

This is particularly important if you have an existing database schema.

If you do not have an existing database schema, you can delegate table and column definition to OC4J by omitting this configuration: at deployment time, OC4J will create default table and column names based on class and data member names.

Note: You can download a JPA entity table and column code example from:
http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30mappingannotations/doc/how-to-ejb30-mapping-annotations.html.

Configuring the Primary Table

The primary table is the table into which the TopLink entity manager persists your entity: in particular, it is the table that stores the entity's primary key (see "Configuring a JPA Entity Primary Key" on page 7-1). Optionally, you can also specify one or more secondary tables (see "Configuring a Secondary Table" on page 7-7), if the entity's persistent data is stored across multiple tables.

You define the primary table at the entity class level.

Using Annotations

Example 7–9 shows how to use the `@Table` annotation to define the primary table for the `Employee` class. The TopLink entity manager will persist instances of this entity to a table named `EJB_EMPLOYEE`.

Example 7–9 @Table

```
@Entity
@Table(name="EJB_EMPLOYEE")
public class Employee implements Serializable {
...
}
```

Configuring a Secondary Table

Specifying one or more secondary tables indicates that the entity's persistent data is stored across multiple tables. You must first specify a primary table (see "Configuring the Primary Table" on page 7-7) before you can specify any secondary tables.

You define a secondary table at the entity class level.

If you specify one or more secondary tables, you can specify the secondary table name in the column definition (see "Configuring a Column" on page 7-8) for persistent fields that are stored in that table. This enables the distribution of entity persistent fields across multiple tables.

Using Annotations

Example 7–10 shows how to use the `@SecondaryTable` annotation to specify that some of the entity's persistent data is stored in a table named `EJB_SALARY`.

Example 7–10 @SecondaryTable

```
@Entity
@Table(name="EJB_EMPLOYEE")
@SecondaryTable(name="EJB_SALARY")
public class Employee implements Serializable {
...
}
```

Configuring a Column

The column is, by default, the name of the column in the primary table (see "Configuring the Primary Table" on page 7-7), into which the TopLink entity manager stores the field's value.

You define the column at one of the property (getter or setter method) or field level of your entity.

If you specified one or more secondary tables (see "Configuring a Secondary Table" on page 7-7), you can specify the secondary table name in the column definition. This enables the distribution of entity persistent fields across multiple tables.

Using Annotations

Example 7–11 shows how to use the `@Column` annotation to specify column `F_NAME` in the primary table for field `firstName`.

Example 7–11 @Column for the Primary Table

```
@Column(name="F_NAME")
public String getFirstName() {
    return firstName;
}
```

Example 7–12 shows how to use the `@Column` annotation to specify column `SALARY` in secondary table `EMP_SALARY` for field `salary`.

Example 7–12 @Column for a Secondary Table

```
@Column(name="SALARY", secondaryTable="EMP_SALARY")
public String getSalary() {
    return salary;
}
```

Configuring a Join Column

A join column specifies a mapped, foreign key column for joining an entity association or a secondary table.

You can define a join column with the following:

- a secondary table (see Example 7–13);
- an one-to-one mapping (see Example 7–14);
- a many-to-one mapping (see Example 7–15);
- an one-to-many mapping (see Example 7–16);

Using Annotations

Example 7–13 shows how to use the `@JoinColumn` annotation to specify a join column with a secondary table. For more information, see "Configuring a Secondary Table" on page 7-7.

Example 7–13 @JoinColumn With a Secondary Table

```
@Entity
@Table(name="EJB_EMPLOYEE")
@SecondaryTable(name="EJB_SALARY")
@JoinColumn(name="EMP_ID", referencedColumnName="EMP_ID")
public class Employee implements Serializable {
...
}
```

Example 7–14 shows how to use the `@JoinColumn` annotation to specify a join column with an one-to-one mapping. For more information, see "Configuring an One-to-One Mapping" on page 7-11.

Example 7–14 @JoinColumn With an One-to-One Mapping

```
@OneToOne(cascade=ALL, fetch=LAZY)
@JoinColumn(name="ADDR_ID")
public Address getAddress() {
    return address;
}
```

Example 7–15 shows how to use the `@JoinColumn` annotation to specify a join column with a many-to-one mappiong. For more information, see "Configuring a Many-to-One Mapping" on page 7-12.

Example 7–15 @JoinColumn With a Many-to-One Mapping

```
@ManyToOne(cascade=PERSIST, fetch=LAZY)
@JoinColumn(name="MANAGER_ID", referencedColumnName="EMP_ID")
public Employee getManager() {
    return manager;
}
```

Example 7–16 shows how to use the `@JoinColumn` annotation to specify a join column with an one-to-many mapping. Fore more information, see "Configuring an One-to-Many Mapping" on page 7-12.

Example 7–16 @JoinColumn With an One-to-Many Mapping

```
@OneToMany(cascade=PERSIST)
@JoinColumn(name="MANAGER_ID", referencedColumnName="EMP_ID")
public Collection getManagedEmployees() {
    return managedEmployees;
}
```

Configuring a Container-Managed Relationship Field for a JPA Entity

In a JPA entity, you define container-managed relationship (CMR) fields (see "What are JPA Entity Container-Managed Relationship Fields?" on page 1-36) as follows:

- "Configuring a Basic Mapping" on page 7-10
- "Configuring a Large Object Mapping" on page 7-10
- "Configuring a Serialized Object Mapping" on page 7-11

- "Configuring an One-to-One Mapping" on page 7-11
- "Configuring a Many-to-One Mapping" on page 7-12
- "Configuring an One-to-Many Mapping" on page 7-12
- "Configuring a Many-to-Many Mapping" on page 7-13

Note: You can download a JPA entity container-managed relationship field code example from: http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30mappingannotations/doc/how-to-ejb30-mapping-annotations.html.

Configuring a Basic Mapping

Use a basic mapping to map an field that contains a primitive or JDK object value. For example, use a basic mapping to store a `String` attribute in a `VARCHAR` column.

You define a basic mapping at one of the property (getter or setter method) or field level of your entity.

Optionally, you can define the strategy for fetching data from the database (see "Configuring Lazy Loading" on page 7-16.

For more information, see "Understanding Direct-to-Field Mapping" in the *Oracle TopLink Developer's Guide*.

Note: For an EJB 3.0 basic mapping code example, see: http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30mappingannotations/doc/how-to-ejb30-mapping-annotations.html#basic.

Using Annotations

Example 7–17 shows how to use the `@Basic` annotation to specify a basic mapping for field `firstName`.

Example 7–17 @Basic

```
@Basic()
@Column(name="F_NAME")
public String getFirstName() {
    return firstName;
}
```

Configuring a Large Object Mapping

Use a large object (LOB) mapping to specify that a persistent property or field should be persisted as a LOB to a database-supported LOB type. A LOB may be either a binary (BLOB) or character (CLOB) type.

You define a large object mapping at one of the property (getter or setter method) or field level of your entity.

For more information, see the following:

- "Understanding Direct-to-Field Mapping" in the *Oracle TopLink Developer's Guide*

- "Using a Converter Mapping" in the *Oracle TopLink Developer's Guide*
- "Type Conversion Converter" in the *Oracle TopLink Developer's Guide*

Using Annotations

Example 7–18 shows how to use the `@Lob` annotation to specify a large object mapping for field `image`.

Example 7–18 @Lob

```
@Lob(fetch=EAGER, type=BLOB)
@Column(name="IMAGE")
public Byte[] getImage() {
    return image;
}
```

Configuring a Serialized Object Mapping

Use a serialized object mapping to specify that a persistent property should be persisted as a serialized stream of bytes.

You define a serialized object at one of the property (getter or setter method) or field level of your entity.

For more information, see the following:

- "Understanding Direct-to-Field Mapping" in the *Oracle TopLink Developer's Guide*
- "Using a Converter Mapping" in the *Oracle TopLink Developer's Guide*
- "Serialized Object Converter" in the *Oracle TopLink Developer's Guide*

Using Annotations

Example 7–19 shows how to use the `@Serialized` annotation to specify a serialized object mapping for field `picture`.

Example 7–19 @Serialized

```
@Serialized(fetch=EAGER)
@Column(name="PICTURE")
public Byte[] getPicture() {
    return picture;
}
```

Configuring an One-to-One Mapping

Use an one-to-one mapping to represent simple pointer references between two Java objects. In Java, a single pointer stored in an attribute represents the mapping between the source and target objects. Relational database tables implement these mappings using foreign keys.

You define an one-to-one mapping at one of the property (getter or setter method) or field level of your entity.

For more information, see "Understanding One-to-One Mapping" in the *Oracle TopLink Developer's Guide*.

Note: For an EJB 3.0 basic mapping code example, see: http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30mappingannotations/doc/how-to-ejb30-mapping-annotations.html#onetoone.

Using Annotations

Example 7–20 shows how to use the `@OneToOne` annotation to specify an one-to-one mapping for field `address`.

Example 7–20 @OneToOne

```
@OneToOne(cascade=ALL, fetch=LAZY)
@JoinColumn(name="ADDR_ID")
public Address getAddress() {
    return address;
}
```

Configuring a Many-to-One Mapping

Use a many-to-one mapping to represent simple pointer references between two Java objects. In Java, a single pointer stored in an attribute represents the mapping between the source and target objects. Relational database tables implement these mappings using foreign keys.

You define a many-to-one mapping at one of the property (getter or setter method) or field level of your entity.

For more information, see "Understanding One-to-One Mapping" in the *Oracle TopLink Developer's Guide*.

Note: For an EJB 3.0 basic mapping code example, see: http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30mappingannotations/doc/how-to-ejb30-mapping-annotations.html#manytoone.

Using Annotations

Example 7–21 shows how to use the `@ManyToOne` annotation to specify a many-to-one mapping for field `manager`.

Example 7–21 @ManyToOne

```
@ManyToOne(cascade=PERSIST, fetch=LAZY)
@JoinColumn(name="MANAGER_ID", referencedColumnName="EMP_ID")
public Employee getManager() {
    return manager;
}
```

Configuring an One-to-Many Mapping

Use an one-to-many mapping to represent the relationship between a single source object and a collection of target objects. This relationship is a good example of something that is simple to implement in Java using a `Vector` (or other collection types) of target objects, but difficult to implement using relational databases.

You define a one-to-many mapping at one of the property (getter or setter method) or field level of your entity.

For more information, see "Understanding One-to-Many Mapping" in the *Oracle TopLink Developer's Guide*.

Note: For an EJB 3.0 basic mapping code example, see: http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30mappingannotations/doc/how-to-ejb30-mapping-annotations.html#onetomany.

Using Annotations

Example 7–22 shows how to use the `@OneToMany` annotation to specify an one-to-many mapping for field `managedEmployees`.

Example 7–22 @OneToMany

```
@OneToMany(cascade=PERSIST)
@JoinColumn(name="MANAGER_ID", referencedColumnName="EMP_ID")
public Collection getManagedEmployees() {
    return managedEmployees;
}
```

Configuring a Many-to-Many Mapping

Use a many-to-many mapping to represent the relationships between a collection of source objects and a collection of target objects. This mapping requires the creation of an intermediate table (the association table) for managing the associations between the source and target records.

You define a many-to-many mapping at one of the property (getter or setter method) or field level of your entity.

For more information, see "Understanding Many-to-Many Mapping" in the *Oracle TopLink Developer's Guide*.

Note: For an EJB 3.0 basic mapping code example, see: http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30mappingannotations/doc/how-to-ejb30-mapping-annotations.html#manytomany.

Using Annotations

Example 7–23 shows how to use the `@ManyToMany` annotation to specify a many-to-many mapping for field `projects` and how to use the `@JoinTable` annotation to specify an association table.

Example 7–23 @ManyToMany

```
@ManyToMany(cascade=PERSIST)
@JoinTable(
    name="EJB_PROJ_EMP",
    joinColumns=@JoinColumn(name="EMP_ID", referencedColumnName="EMP_ID"),
    inverseJoinColumns=@JoinColumn(name="PROJ_ID", referencedColumnName="PROJ_ID")
)
public Collection getProjects() {
```

```
    return projects;
}
```

Configuring an Aggregate Mapping

Two entities–an owning (parent or source) entity and an owned (child or target) entity–are related by aggregation if there is a strict one-to-one relationship between them and all the attributes of the owned entity can be retrieved from the same table(s) as the owning entity. This means that if the owning entity exists, then the owned entity must also exist and if the owning entity is destroyed, then the owned entity is also destroyed.

An aggregate mapping lets you associate data members in the owned entity with fields in the owning entity's underlying database tables.

In the owning entity, you designate the owned field or setter as embedded.

In owned entity, you designate the class as embeddable and associate it with the owning entity's table name.

In the owning entity, you can override any column specifications (see "Configuring a Column" on page 7-8) made in the owned entity.

For more information, see "Understanding Aggregate Mapping" in the *Oracle TopLink Developer's Guide*.

Note: For an EJB 3.0 basic mapping code example, see: `http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30mappingannotations/doc/how-to-ejb30-mapping-annotations.html#embedded`.

Using Annotations

Example 7–24 shows how to use the `@Embedded` annotation to specify an aggregate mapping for field `period`. This field contains an instance of `EmploymentPeriod`. Example 7–25 shows how to use the `@Embeddable` annotation to specify the `EmploymentPeriod` entity class as being eligible for use in an aggregate mapping and how to use the `@Table` annotation (see "Configuring the Primary Table" on page 7-7) to associate this class with the owning entity's table.

Example 7–24 @Embedded

```
@Entity
@Table(name="EJB_EMPLOYEE")
public class Employee implements Serializable {
...
    @Embedded
    public EmploymentPeriod getPeriod() {
        return period;
    }
...
}
```

Example 7–25 @Embeddable

```
@Embeddable
@Table(name="EJB_EMPLOYEE")
public class EmploymentPeriod implements Serializable {
```

```
    private Date startDate;
    private Date endDate;
...
}
```

You can use the `@AttributeOverride` in the owning entity (see Example 7–26) to override the column definitions made in the owned entity (see Example 7–27).

Example 7–26 @Embedded and @AttributeOverride

```
@Entity
@Table(name="EJB_EMPLOYEE")
public class Employee implements Serializable {
...
    @Embedded({
            @AttributeOverride(name="startDate", column=@Column("EMP_START")),
            @AttributeOverride(name="endDate", column=@Column("EMP_END"))}
    )
    public EmploymentPeriod getPeriod() {
        return period;
    }
...
}
```

Example 7–27 @Embeddable and @Column

```
@Embeddable
@Table(name="EJB_EMPLOYEE")
public class EmploymentPeriod implements Serializable {
    @Column("START_DATE")
    private Date startDate;

    @Column("END_DATE")
    private Date endDate;
...
}
```

Configuring Optimistic Lock Version Field

You can specify an entity field to function as a version field for use in a TopLink optimistic version locking policy. OC4J uses this version field to ensure integrity when reattaching (see "Detaching and Merging an Entity Bean Instance" on page 29-17) and for overall optimistic concurrency control.

You define the optimistic lock version field at one of the property (getter or setter method) or field level of your entity.

For more information, see "Optimistic Version Locking Policies" in the *Oracle TopLink Developer's Guide*.

Using Annotations

Example 7–28 shows how to use the `@Version` annotation to define an optimistic version locking policy using column `VERSION`.

Example 7–28 @Version

```
@Version
@Column(name="VERSION")
public int getVersion() {
    return version;
```

```
}
```

Configuring Lazy Loading

For all EJB 3.0 mapping types (basic and relationship mappings), you can define the strategy for fetching data from the database as one of the following:

- `FetchType.LAZY`: when the entity is retrieved, the persistent field value is not retrieved. The value is retrieved if and when the field is accessed.
- `FetchTyep.EAGER`: when the entity is retrieved, the persistent field value is also retrieved.

By default, all persistent fields are fetched eagerly.

If you are using finders in your EJB 3.0 application, you can configure lazy loading at the finder level. This is an Oracle-specific option that you configure using the EJB 2.1 `orion-ejb-jar.xml` file. For more information, see "Configuring Lazy Loading on Finder Methods" on page 14-14.

Using Annotations

Example 7–29 shows how to use the `@Basic` annotation to define a fetch strategy of `LAZY`.

Example 7–29 @Basic Fetch Attribute

```
@Basic(fetch=FetchType.LAZY)
@Column(name="F_NAME")
public String getFirstName() {
    return firstName;
}
```

Configuring a Life Cycle Callback Method on a JPA Entity

You can specify a JPA entity class method as a callback method for any of the following life cycle events:

- Pre-persist
- Post-persist
- Pre-remove
- Post-remove
- Pre-update
- Post-update
- Post-load

The entity class method must have the following signature:

```
int <METHOD>()
```

The entity class method can have any method name as long as it does not begin with `ejb`.

For more information, see the following:

- "What is the JPA Entity Life Cycle?" on page 1-37

- "Life Cycle Callback Methods on a Bean Class" on page 1-6
- "Descriptor Event Manager" in the *Oracle TopLink Developer's Guide*
- "Configuring a Domain Object Method as an Event Handler" in the *Oracle TopLink Developer's Guide*

> **Note:** For an EJB 3.0 life cycle callback method code example, see: http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30mappingannotations/doc/how-to-ejb30-mapping-annotations.html#callbacks.

Using Annotations

You can specify a JPA entity class method as a life cycle callback method using any of the following annotations:

- `@PrePersist`
- `@PostPersist`
- `@PreRemove`
- `@PostRemove`
- `@PreUpdate`
- `@PostUpdate`
- `@PostLoad`

Example 7–30 shows how to use the `@PrePersist` annotation to specify JPA entity class method `initialize` as a life cycle callback method.

Example 7–30 @PrePersist

```
@Entity
@Table(name="EJB_PROJECT")
public class Project implements Serializable {
    ...
    @Id()
    @Column(name="PROJECT_ID", primaryKey=true)
    public Integer getId() {
         return id;
    }

    ...

    @PrePersist
    public int initialize() {
        ...
    }
}
```

Configuring a Life Cycle Callback Listener Method on an Entity Listener Class of a JPA Entity

You can designate an entity listener method on an entity listener class of a JPA entity as a life cycle callback method.

To configure a life cycle callback listener method on an entity listener class, you must do the following:

1. Create an entity listener class.

   This can be any POJO class.

2. Implement the life cycle callback listener method in the entity listener class.

   Callback methods defined on a JPA entity listener class have the following signature:

   ```
   void <METHOD>(Object)
   ```

   You may specify an argument type of `Object` or the type of the JPA entity class that you will associate the entity listener class with.

3. Associate a life cycle event with the callback listener method.

   You may associate a life cycle event with one and only one callback listener method, but you may associate a given callback listener method with more than one life cycle event.

   For more information, see the following:

   - "Using Annotations" on page 7-18

4. Associate the interceptor class with your JPA entity.

   For more information, see the following:

   - "Using Annotations" on page 7-18

For more information, see the following:

- "What is the JPA Entity Life Cycle?" on page 1-37
- "Life Cycle Callback Listener Methods on a JPA Entity Listener Class" on page 1-6

Using Annotations

You can specify a JPA entity listener method as a life cycle callback method using any of the following annotations:

- `@PrePersist`
- `@PostPersist`
- `@PreRemove`
- `@PostRemove`
- `@PreUpdate`
- `@PostUpdate`
- `@PostLoad`

Example 7–31 shows how to use the `@PostConstruct` and `@PreDestroy` annotation to specify JPA entity listener methods `myPostConstruct` and `myPreDestroy` as life cycle callback methods, respectively.

Example 7–31 @PrePersist Life Cycle Listener Callback Method

```
public class MyProjectEntityListener {
    ...
    @PostConstruct
    public void myPostConstruct (Project obj) { // or just Object
        ...
    }
```

```
    @PreDestroy
    public void myPreDestroy (Project obj)  { // or just Object
        ...
    }
}
```

You can associate an entity listener class with a JPA entity using the `@EntityListeners` annotation. Example 7–32 shows how to associate the entity listener class from Example 7–31 with a JPA entity class.

Note that the life cycle method for `@PrePersist` is a method of the JPA entity class itself (for more information, see "Configuring a Life Cycle Callback Method on a JPA Entity" on page 7-16).

Example 7–32 Associating an Entity Listener Class With a JPA Entity

```
@Entity
@EntityListeners(MyProjectEntityListener.class)
@Table(name="EJB_PROJECT")
public class Project implements Serializable {
    ...
    @Id
    @Column(name="PROJECT_ID", primaryKey=true)
    public Integer getId() {
        return id;
    }

    ...

    @PrePersist
    public int initialize() {
        ...
    }
}
```

Configuring Inheritance for a JPA Entity

OC4J supports the following inheritance strategies for mapping a class or class hierarchy to a relational database schema:

- Joined Subclass
- Single Table for Each Class Hierarchy

You can configure either approach using annotations (see "Using Annotations" on page 7-20).

> **Note:** For an EJB 3.0 inheritance code example, see: `http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30inheritance/doc/how-to-ejb30-inheritance.html`.

Joined Subclass

In this strategy, fields that are specific to a subclass are mapped to a different table than the fields that are common to the parent class, and a join is performed to instantiate the subclass.

The root of the class hierarchy is represented by a single table. Each subclass is represented by a separate table that contains the columns that are specific to the

subclass (not inherited from its superclass), as well as the column that represent the subclass's primary key. If the subclass does not have any additional state over its superclass, a separate table is not required.

If the subclass table has primary key column, it serves as a foreign key to the primary key of the superclass table. If the subclass table primary key column name is the same as that of the primary key column of the superclass table, OC4J infers this relationship. If the subclass table primary key column name is not the same as that of the primary key column of the superclass table (or, if the subclass table does not have primary key column), you must specify a subclass table column to use to join the primary table of an entity subclass to the primary table of its superclass.

The primary table of the superclass also has a column that serves as a discriminator column, that is, a column whose value identifies the specific subclass to which the instance that is represented by the row belongs.

For more information, see "Configuring Joined Subclass Inheritance With Annotations" on page 7-20.

Single Table for Each Class Hierarchy

In this strategy, all the classes in a hierarchy are mapped to a single table. The table has a column that serves as a discriminator column. Each subclass that adds additional state maps to this new state only in this single table. Such columns are only used by that subclass.

For more information, see "Configuring Single Table Inheritance With Annotations" on page 7-21.

Using Annotations

This section describes the following:

- Configuring Joined Subclass Inheritance With Annotations
- Configuring Single Table Inheritance With Annotations

Configuring Joined Subclass Inheritance With Annotations

The following examples show how to configure inheritance using a joined subclass approach (see "Joined Subclass" on page 7-19): Example 7–33 shows how to use the `@Inheritance` annotation in the base class `Project`. Example 7–34 and Example 7–35 show how to use the `@Inheritance` annotation in derived classes `LargeProject` and `SmallProject`, respectively.

The primary table is `EJB_PROJECT` to which both `Project` and `SmallProject` are mapped. `EJB_PROJECT` has a discriminator column called `PROJ_TYPE` that represents `Project`, `LargeProject` and `SmallProject` with values `P`, `L` and `S`, respectively. `LargeProject` adds additional state to `Project`, so is mapped to its own table, `EJB_LPROJECT`, which contains fields specific to `LargeProject`, such as `BUDGET`. Note that `EJB_LPROJECT` does not have a primary key column; instead it has a foreign key (`PROJ_ID`) that has the same name as the primary key of `EJB_PROJECT`.

Note that in Example 7–34, because the `LargeProject` class primary key column name (`LARGE_PROJECT_ID`) is not the same as that of the primary key column of the superclass table (`ID`), you must use the `@InheritanceJoinColumn` annotation to specify the column used to join the `LargeProject` primary table to the primary table of its superclass.

Example 7–33 @Inheritance: Base Class Project in Joined Subclass Inheritance

```
@Entity
@Table(name="EJB_PROJECT")
@Inheritance(strategy=JOINED, discriminatorValue="P")
@DiscriminatorColumn(name="PROJ_TYPE")
public class Project implements Serializable {
...
    @Id()
    @Column(name="PROJECT_ID", primaryKey=true)
    public Integer getId() {
        return id;
    }
...
}
```

Example 7–34 @Inheritance: Derived Class LargeProject in Joined Subclass Inheritance

```
@Entity
@Table(name="EJB_LPROJECT")
@Inheritance(discriminatorValue="L")
@InheritanceJoinColumn(name="LARGE_PROJECT_ID")
public class LargeProject extends Project {
...
    @Id()
    @Column(name="LARGE_PROJECT_ID", primaryKey=true)
    public Integer getProjectId() {
        return projectId;
    }
...
}
```

Example 7–35 @Inheritance: Derived Class SmallProject in Joined Subclass Inheritance

```
@Entity
@Table(name="EJB_PROJECT")
@Inheritance(discriminatorValue="S")
public class SmallProject extends Project {
...
}
```

Configuring Single Table Inheritance With Annotations

The following examples show how to configure inheritance using a single table for each class hierarchy approach (see "Single Table for Each Class Hierarchy" on page 7-20): Example 7–33 shows how to use the `@Inheritance` annotation in the base class `Project`. Example 7–34 and Example 7–35 show how the `@Inheritance` annotation is not needed in derived classes `LargeProject` and `SmallProject`, respectively.

The primary table is `EJB_PROJECT`, to which both `Project` and `SmallProject` are mapped. The `EJB_PROJECT` table would contain all the columns for `Project` and an additional column (`BUDGET`) used only by `LargeProject`.

Example 7–36 @Inheritance: Base Class Project in Single Table Inheritance

```
@Entity
@Table(name="EJB_PROJECT")
@Inheritance(strategy=SINGLE_TABLE, discriminatorValue="P")
@DiscriminatorColumn(name="PROJ_TYPE")
public class Project implements Serializable {
...
```

```
}
```

Example 7–37 @Inheritance: Derived Class LargeProject in Single Table Inheritance

```
@Entity
@Inheritance(discriminatorValue="L")
public class LargeProject extends Project {
...
}
```

Example 7–38 @Inheritance: Derived Class SmallProject in Single Table Inheritance

```
@Entity
@Inheritance(discriminatorValue="S")
public class SmallProject extends Project {
...
}
```

8

Implementing JPA Queries

This section describes how to create predefined, static queries that you can access at run time, including the following:

- Implementing a JPA Named Query
- Implementing a JPA Dynamic Query
- Configuring TopLink Query Hints in a JPA Query

For more information, see "How do you Query for a JPA Entity?" on page 1-39.

Implementing a JPA Named Query

A named query is a predefined query that you create and associate with a container-managed entity (see "Using Annotations" on page 8-1). At deployment time, OC4J stores named queries on the `EntityManager`. At run time, you can use the `EntityManager` to acquire, configure, and execute a named query.

For more information, see the following:

- "Acquiring an EntityManager" on page 29-8
- "Creating a Named Query With the EntityManager" on page 29-13
- "Executing a Query" on page 29-15

Using Annotations

Example 8–1 shows how to use the `@NamedQuery` annotation to define a Java persistence query language query that you can acquire by name `findAllEmployeesByFirstName` at run time using the `EntityManager`.

Example 8–1 Implementing a Query Using @NamedQuery

```
@Entity
@NamedQuery(
    name="findAllEmployeesByFirstName",
    queryString="SELECT OBJECT(emp) FROM Employee emp WHERE emp.firstName = 'John'"
)
public class Employee implements Serializable {
...
}
```

Example 8–2 shows how to use the `@NamedQuery` annotation to define a Java persistence query language query that takes a parameter named `firstname`. Example 8–3 shows how you use the `EntityManager` to acquire this query and use `Query` method `setParameter` to set the `firstname` parameter. For more

information on using the EntityManager with named queries, see "Querying for a JPA Entity Using the EntityManager" on page 29-13.

Optionally, you can configure your named query with query hints to use JPA persistence provider vendor extensions (see "Configuring TopLink Query Hints in a JPA Query" on page 8-3).

Example 8–2 Implementing a Query With Parameters Using @NamedQuery

```
@Entity
@NamedQuery(
    name="findAllEmployeesByFirstName",
    queryString="SELECT OBJECT(emp) FROM Employee emp WHERE emp.firstName = :firstname"
)
public class Employee implements Serializable {
...
}
```

Example 8–3 Setting Parameters in a Named Query

```
Query queryEmployeesByFirstName = em.createNamedQuery("findAllEmployeesByFirstName");
queryEmployeeByFirstName.setParameter("firstName", "John");
Collection employees = queryEmployessByFirstName.getResultList();
```

Implementing a JPA Dynamic Query

Using `EntityManager` methods `createQuery` or `createNativeQuery`, you can create a `Query` object dynamically at run time (see "Using Java" on page 8-2). Using the `Query` methods `getResultList`, `getSingleResult`, or `executeUpdate` you can execute the query (see "Executing a Query" on page 29-15).

Optionally, you can configure your named query with query hints to use JPA persistence provider vendor extensions (see "Configuring TopLink Query Hints in a JPA Query" on page 8-3).

For more information, see the following:

- "Acquiring an EntityManager" on page 29-8
- "Creating a Dynamic Java Persistence Query Language Query With the Entity Manager" on page 29-14
- "Creating a Dynamic TopLink Expression Query With the EntityManager" on page 29-14
- "Creating a Dynamic Native SQL Query With the EntityManager" on page 29-15
- "Executing a Query" on page 29-15

Using Java

Example 8–4 shows how to create a dynamic EJB QL query with parameters and how to execute the query. In this example, the query returns multiple results so `Query` method `getResultList` is used.

Example 8–4 Implementing and Executing a Dynamic Query

```
Query queryEmployeeByFirstName = entityManager.createQuery(
    "SELECT OBJECT(emp) FROM Employee emp WHERE emp.firstName = :firstname"
);
```

```
queryEmployeeByFirstName.setParameter("firstName", "Joan");

Collection employees = queryEmployeeByFirstName.getResultList();
```

Configuring TopLink Query Hints in a JPA Query

Table 8–1 lists the TopLink EJB 3.0 JPA persistence provider query hints you can specify when you construct a JPA query, as Example 8–5 shows, or when you specify a JPA query using the `@QueryHint` annotation, as Example 8–6 shows. When you set a hint, you can set the value using the corresponding `public static final` field in the appropriate configuration class in `oracle.toplink.essentials.config` as follows:

- `PessimisticLock`
- `TopLinkQueryHints`
- `HintValues`

> **Note:** To access these classes, put the appropriate OC4J persistence JAR on your classpath (see "JPA Persistence JAR Files" on page 3-2).

Example 8–5 Specifying a TopLink JPA Query Hint

```
import oracle.toplink.essentials.config.HintValues;
import oracle.toplink.essentials.config.TopLinkQueryHints;

Customer customer = (Customer)entityMgr.createNamedQuery("findCustomerBySSN").
setParameter("SSN", "123-12-1234").setHint(TopLinkQueryHints.BIND_PARAMETERS,
HintValues.PERSISTENCE_UNIT_DEFAULT).getSingleResult();
```

Example 8–6 Specifying a TopLink JPA Query Hint With @QueryHint

```
import oracle.toplink.essentials.config.HintValues;
import oracle.toplink.essentials.config.TopLinkQueryHints;

@Entity
@NamedQuery(
    name="findAllEmployees",
    query="SELECT * FROM EMPLOYEE WHERE MGR=1"
    hints={
        @QueryHint={name=TopLinkQueryHints.BIND_PARAMETERS, value=HintValues.PERSISTENCE_
UNIT_DEFAULT}
    }
)
public class Employee implements Serializable {
    ...
}
```

Table 8–1 TopLink JPA Query Hints

Hint	Usage	Default

`toplink.jdbc.bind-parameters`	Control whether or not the query uses parameter binding. For more information, see "Parameterized SQL (Binding) and Prepared Statement Caching" in the *Oracle TopLink Developer's Guide*.
	Valid values: `oracle.toplink.essentials.config.HintValues`
	■ `true`: bind all parameters.
	■ `false`: do not bind all parameters.
	■ `PersistenceUnitDefault`: use the parameter binding setting made in your TopLink session's database login.
	For more information, see "Configuring JDBC Options" in the *Oracle TopLink Developer's Guide*.
	Example: JPA Query API
	`import oracle.toplink.essentials.config.HintValues;`
	`import oracle.toplink.essentials.config.TopLinkQueryHints;`
	`query.setHint(TopLinkQueryHints.BIND_PARAMETERS, HintValues.TRUE);`
	Example: `@QueryHint`
	`import oracle.toplink.essentials.config.HintValues;`
	`import oracle.toplink.essentials.config.TargetDatabase;`
	`@QueryHint(name=TopLinkQueryHints.BIND_PARAMETERS,`
	`value=HintValues.PERSISTENCE_UNIT_DEFAULT);`
		`PersistenceUnitDefault`

`toplink.pessimistic-lock`	Control whether or not pessimistic locking is used.
	Valid values:
	`oracle.toplink.essentials.config.PessimisticLock`
	■ `NoLock`: pessimistic locking is not used.
	■ `Lock`: TopLink issues a `SELECT.... FOR UPDATE`.
	■ `LockNoWait`: TopLink issues a `SELECT.... FOR UPDATE NO WAIT`.
	Example: JPA Query API
	`import oracle.toplink.essentials.config.PessimisticLock;`
	`import oracle.toplink.essentials.config.TopLinkQueryHints;`
	`query.setHint(TopLinkQueryHints.PESSIMISTIC_LOCK,`
	`PessimisticLock.LockNoWait);`
	Example: `@QueryHint`
	`import oracle.toplink.essentials.config.PessimisticLock;`
	`import oracle.toplink.essentials.config.TopLinkQueryHints;`
	`@QueryHint(name=TopLinkQueryHints.PESSIMISTIC_LOCK,`
	`value=PessimisticLock.LockNoWait);`
		`NoLock`

`toplink.refresh`	Control whether or not to update the TopLink session cache with objects that the query returns.
	Valid values: `oracle.toplink.essentials.config.HintValues`
	■ `true`: refresh cache.
	■ `false`: do not refresh cache.
	Example: JPA Query API
	`import oracle.toplink.essentials.config.HintValues;`
	`import oracle.toplink.essentials.config.TopLinkQueryHints;`
	`query.setHint(TopLinkQueryHints.REFRESH, HintValues.TRUE);`
	Example: `@QueryHint`
	`import oracle.toplink.essentials.config.HintValues;`
	`import oracle.toplink.essentials.config.TopLinkQueryHints;`
	`@QueryHint(name=TopLinkQueryHints.REFRESH, value=HintValues.TRUE);`
		`false`

Part IV

EJB 3.0 Message-Driven Beans

This part provides procedural information on implementing and configuring EJB 3.0 message-driven beans. For conceptual information, see Part I, "EJB Overview".

This part contains the following chapters:

9

Implementing an EJB 3.0 Message-Driven Bean

This chapter explains how to implement an EJB 3.0 message-driven bean (MDB).

For more information, see the following:

- "What is a Message-Driven Bean?" on page 1-56
- "Using an EJB 3.0 Message-Driven Bean" on page 10-1

Implementing an EJB 3.0 MDB

EJB 3.0 greatly simplifies the development of enterprise beans, removing many complex development tasks. For example:

- The bean class can be a POJO; it does not need to implement `javax.ejb.MessageDrivenBean`.
- Annotations are used for many features, including the message destination and topic (or queue) factory.
- You can use injection to acquire a `MessageDrivenEntityContext`.

For more information, see "What is a Message-Driven Bean?" on page 1-56.

> **Note:** You can download an EJB 3.0 message-driven bean code example from: `http://www.oracle.com/technology/tech/java/oc4j/10131/how_to/how-to-ejb30-mdb/doc/how-to-ejb30-mdb.html`.

To implement an EJB 3.0 message-driven bean, do the following:

1. Configure your message service provider.

   For more information, see the following:

   - "What Message Service Providers Can you use With Your MDB?" on page 2-21
   - "Configuring Message Services" on page 23-1

2. Create the message-driven bean class.

   You can create a POJO and define it as a message-driven bean with the `@MessageDriven` annotation.

   > **Note:** OC4J ignores the `@MessageDriven` attribute `mappedName`.

3. Configure message service provider information as follows:

   You can define this information with the `@ActivationConfigProperty` annotation.

   For more information, see the following:

   - "Configuring an EJB 3.0 MDB to Access a Message Service Provider Using J2CA" on page 10-1
   - "Configuring an EJB 3.0 MDB to Access a Message Service Provider Directly" on page 10-3

4. Add a data member for the `MessageDrivenContext`.

   You can use resource injection to easily initialize this data member without getter and setter methods.

5. Implement the appropriate message listener interface as follows:

   For a JMS message-driven bean, implement the `javax.jms.MessageListener` interface to provide the `onMessages` method with the following signature:

   ```
   public void onMessage(javax.jms.Message message)
   ```

   This method processes the incoming message. Most MDBs receive messages from a queue or a topic, then invoke an entity bean to process the request contained within the message.

   In this method, you can use the `MessageDrivenContext` to acquire and configure a `javax.ejb.TimerService` if you implemented the `TimedObject` interface (see step 6).

6. Optionally, implement the `javax.ejb.TimedObject` interface.

   Implement the `ejbTimeout` method with the following signature:

   ```
   public void ejbTimeout(javax.ejb.Timer timer)
   ```

7. Optionally, define life cycle callback methods using the appropriate annotations.

   You do not need to define life cycle methods: OC4J provides an implementation for all such methods. Define a method of your message-driven bean class as a life cycle callback method only if you want to take some action of your own at a particular point in the message-driven bean's life cycle.

   For more information, see "Configuring a Life Cycle Callback Interceptor Method on an EJB 3.0 MDB" on page 10-11.

8. Optionally, define OC4J-proprietary deployment options.

   In an EJB 3.0 application, you can do this by annotating your message-driven bean class with the OC4J-proprietary `oracle.j2ee.ejb.@MessageDrivenDeployment` annotation (see "Configuring OC4J-Proprietary Deployment Options on an EJB 3.0 MDB" on page 10-17).

9. Complete the configuration of your message-driven bean (see "Using an EJB 3.0 Message-Driven Bean" on page 10-1).

10

Using an EJB 3.0 Message-Driven Bean

This chapter describes the various options that you must configure in order to use an EJB 3.0 message-driven bean.

Table 10–1 lists these options and indicates which are basic (applicable to most applications) and which are advanced (applicable to more specialized applications).

For more information, see the following:

- "What is a Message-Driven Bean?" on page 1-56
- "Implementing an EJB 3.0 Message-Driven Bean" on page 9-1

Table 10–1 Configurable Options for an EJB 3.0 Message-Driven Bean

Options	Type
"Configuring an EJB 3.0 MDB to Access a Message Service Provider Using J2CA" on page 10-1	Basic
"Configuring an EJB 3.0 MDB to Access a Message Service Provider Directly" on page 10-3	Basic
"Configuring an MDB for Fast Undeploy on Windows Operating System" on page 18-5	Advanced
"Configuring an MDB for Oracle RAC Failover" on page 18-6	Advanced
"Configuring Bean Instance Pool Size" on page 31-4	Basic
"Configuring a Transaction Timeout for a Message-Driven Bean" on page 21-7	Advanced
"Configuring Parallel Message Processing" on page 10-5	Advanced
"Configuring Maximum Delivery Count" on page 10-7	Advanced
Configuring Connection Failure Recovery for an EJB 3.0 MDB on page 10-9	Advanced
"Configuring a Life Cycle Callback Interceptor Method on an EJB 3.0 MDB" on page 10-11	Basic
"Configuring a Life Cycle Callback Interceptor Method on an Interceptor Class of an EJB 3.0 MDB" on page 10-11	Advanced
"Configuring an Around Invoke Interceptor Method on an EJB 3.0 MDB" on page 10-13	Advanced
"Configuring an Around Invoke Interceptor Method on an Interceptor Class of an EJB 3.0 MDB" on page 10-14	Advanced
"Configuring an Interceptor Class for an EJB 3.0 MDB" on page 10-15	Advanced
"Configuring OC4J-Proprietary Deployment Options on an EJB 3.0 MDB" on page 10-17	Advanced

Configuring an EJB 3.0 MDB to Access a Message Service Provider Using J2CA

You can configure an EJB 3.0 MDB to access a message service provider using a J2CA resource adapter, such as the Oracle JMS Connector.

You can do this using annotations (see "Using Annotations" on page 10-2) or deployment XML (see "Using Deployment XML" on page 10-3).

> **Note:** Oracle recommends that you access a message service provider using a J2CA resource adapter such as the Oracle JMS Connector. For more information, see "Restrictions When Accessing a Message Service Provider Without a J2CA Resource Adapter" on page 2-25.

OC4J supports both XA factories for two-phase commit (2PC) transactions, and non-XA factories for transactions that do not require 2PC.

For more information, see:

- "Oracle JMS Connector: J2EE Connector Architecture (J2CA)-Based Provider" on page 2-21
- "Message Service Configuration Options: Annotations or XML? Attributes or Activation Configuration Properties?" on page 2-26
- "How do You Participate in a Global or Two-Phase Commit (2PC) Transaction?" on page 2-20

Using Annotations

To configure an EJB 3.0 MDB to access a JMS message service provider using a J2CA resource adapter:

1. Specify the name of the resource adapter.

   You may use either the OC4J-proprietary `@MessageDrivenDeployment` annotation resourceAdapter attribute (as Example 10–1 shows) or the equivalent `orion-ejb-jar.xml` file `<message-driven-deployment>` element `resource-adapter` attribute (see "Using Deployment XML" on page 10-3).

2. Specify the required activation configuration properties.

   You may specify activation configuration properties using any combination of `@MessageDrivenDeployment` and `@MessageDriven` annotation (as Example 10–1 shows) and deployment XML (see "Using Deployment XML" on page 10-3).

   For more information, see:

   - "J2CA Activation Configuration Properties" on page B-1
   - "Message Service Configuration Options: Annotations or XML? Attributes or Activation Configuration Properties?" on page 2-26

Example 10–1 shows how to configure a message-driven bean to use the Oracle JMS resource adapter named `OracleASjms`. It assumes that you defined connection factory `OracleASjms/MyQCF` in `oc4j-ra.xml` file and destination name `OracleASjms/MyQueue` in `oc4j-connectors.xml` file when you configured your message service provider. You can define either XA-enabled factories for two-phase commit (2PC) support, or non-XA factories if 2PC support is not required. For more information on configuring a J2CA message service provider, see "Configuring a J2CA Resource Adapter for use With Your Message Service Provider" on page 23-1.

Example 10–1 @MessageDriven and @MessageDrivenDeployment Annotation for a J2CA Message Service Provider

```
import javax.ejb.MessageDriven;
import oracle.j2ee.ejb.MessageDrivenDeployment;
import javax.ejb.ActivationConfigProperty;
```

```
import javax.jms.Message;
import javax.jms.MessageListener;

@MessageDriven(
    activationConfig = {
        @ActivationConfigProperty(
            propertyName="ConnectionFactoryJndiName", propertyValue="OracleASjms/MyQCF"),
        @ActivationConfigProperty(
            propertyName="DestinationName", propertyValue="OracleASjms/MyQueue"),
        @ActivationConfigProperty(
            propertyName="DestinationType", propertyValue="javax.jms.Queue"),
        @ActivationConfigProperty(
            propertyName="messageSelector", propertyValue="RECIPIENT = 'simple_jca_test'")
    })

// associate MDB with the resource adapter
@MessageDrivenDeployment(resourceAdapter = "OracleASjms")

public class JCAQueueMDB implements MessageListener {
    public void onMessage(Message msg) {
        ...
    }
}
```

The actual names you use depend on your message service provider installation. For more information, see "J2CA Message Service Provider Connection Factory Names" on page 23-2.

Using Deployment XML

To configure an EJB 3.0 MDB to access a JMS message service provider using a J2CA resource adapter by using deployment XML, you must use both `ejb-jar.xml` and `orion-ejb.jar.xml` files, as you would for an EJB 2.1 MDB (see "Using Deployment XML" on page 18-2).

You can override annotation configuration (see "Using Annotations" on page 10-2), if present, with this deployment XML configuration.

Configuring an EJB 3.0 MDB to Access a Message Service Provider Directly

You can configure an EJB 3.0 MDB to access a message service provider directly (without a J2CA resource adapter).

You can do this by using annotations (see "Using Annotations" on page 10-4) or deployment XML (see "Using Deployment XML" on page 10-5).

> **Note:** Oracle recommends that you access a message service provider using a J2CA resource adapter such as the Oracle JMS Connector. For more information, see:
>
> - "Restrictions When Accessing a Message Service Provider Without a J2CA Resource Adapter" on page 2-25.
> - "Configuring an EJB 3.0 MDB to Access a Message Service Provider Using J2CA" on page 10-1

OC4J supports both XA factories for two-phase commit (2PC) transactions and non-XA factories for transactions that do not require 2PC.

For more information, see:

- "OEMS JMS: In-Memory or File-Based Provider" on page 2-23
- "OEMS JMS Database: Advanced Queueing (AQ)-Based Provider" on page 2-24
- "Message Service Configuration Options: Annotations or XML? Attributes or Activation Configuration Properties?" on page 2-26
- "How do You Participate in a Global or Two-Phase Commit (2PC) Transaction?" on page 2-20

Using Annotations

To configure an EJB 3.0 MDB to access a JMS message service provider using a J2CA resource adapter:

1. Specify the required activation configuration properties.

   You may specify activation configuration properties using any combination of `@MessageDrivenDeployment` annotation, `@MessageDriven` annotation, and deployment XML.

   For more information, see:

   - "J2CA Activation Configuration Properties" on page B-1
   - "Message Service Configuration Options: Annotations or XML? Attributes or Activation Configuration Properties?" on page 2-26

Example 10–11 shows how to configure a message-driven bean to access a JMS message service provider directly (without a J2CA resource adapter). It assumes that you defined connection factory `jms/MyQCF` and queue `jms/MyQueue` when you configured your message service provider. You can define either XA-enabled factories for two-phase commit (2PC) support or non-XA factories if 2PC support is not required. For more information on configuring a message service provider, see "Configuring Message Services" on page 23-1.

Example 10–2 @MessageDriven Annotation for a Non-J2CA Message Service Provider

```
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.Message;
import javax.jms.TextMessage;
import javax.jms.MessageListener;

@MessageDriven(
    messageListenerInterface=MessageListener.class,
    activationConfig = {
        @ActivationConfigProperty(
            propertyName="connectionFactoryJndiName", propertyValue="jms/MyQCF"),
        @ActivationConfigProperty(
            propertyName="destinationName", propertyValue="jms/MyQueue"),
        @ActivationConfigProperty(
            propertyName="destinationType", propertyValue="javax.jms.Queue"),
        @ActivationConfigProperty(
            propertyName="messageSelector", propertyValue="RECIPIENT = 'simple_test'")
    })

public class QueueMDB implements MessageListener {
    public void onMessage(Message msg) {
        ...
    }
}
```

The actual names you use depend on your message service provider installation. For more information, see the following:

- "OEMS JMS Destination and Connection Factory Names" on page 23-3
- "OEMS JMS Database Destination and Connection Factory Names" on page 23-6

Using Deployment XML

To configure an EJB 3.0 MDB to access a JMS message service provider directly (without a J2CA resource adapter) by using deployment XML, you must use both `ejb-jar.xml` and `orion-ejb.jar.xml` files, as you would for an EJB 2.1 MDB (see "Using Deployment XML" on page 18-4).

You can override annotation configuration (see "Using Annotations" on page 10-4), if present, with this deployment XML configuration.

Configuring Parallel Message Processing

By default, OC4J uses one receiver thread to poll for messages from the message location.

Having more than one receiver thread allows messages to be received in parallel which can improve performance.

If your message location is a Topic, the number of receiver threads is fixed to one.

If your message location is a Queue, you can configure the number of receiver threads using OC4J-proprietary annotations (see "Using Annotations" on page 10-5) or using the `orion-ejb-jar.xml` file (see "Using Deployment XML" on page 10-7).

Note that the minimum number of bean instances in the MDB pool should be at least the same as the number of receiver threads to avoid blocking receiver threads from acquiring a bean instance from the pool to process messages.

For more information, see:

- "Message Service Configuration Options: Annotations or XML? Attributes or Activation Configuration Properties?" on page 2-26
- "Configuring Bean Instance Pool Size" on page 31-4

Using Annotations

How you configure this option depends on how you access your message-service provider you are using:

- Accessing a Message Service Provider Using a J2CA Resource Adapter
- Accessing a Message Service Provider Without Using a J2CA Resource Adapter

Accessing a Message Service Provider Using a J2CA Resource Adapter

If you access your message-service provider using a J2CA resource adapter, set activation configuration property `ReceiverThreads`, as Example 10–3 shows.

For more information on `ReceiverThreads`, see Table B–2.

Example 10–3 Configuring Parallel Message Processing for a J2CA Adapter Message Service Provider

```
import javax.ejb.MessageDriven;
import oracle.j2ee.ejb.MessageDrivenDeployment;
```

```
import javax.ejb.ActivationConfigProperty;
import javax.jms.Message;
import javax.jms.MessageListener;

@MessageDriven(
    activationConfig = {
        @ActivationConfigProperty(propertyName="ReceiverThreads", propertyValue="3"),
        ...
    }
)

@MessageDrivenDeployment(
    resourceAdapter = "OracleASjms",
    ...
)

public class JCAQueueMDB implements MessageListener {
    public void onMessage(Message msg) {
        ...
    }
}
```

Accessing a Message Service Provider Without Using a J2CA Resource Adapter

If you access your message-service provider directly (without using aJ2CA resource adapter), set OC4J-proprietary annotation `@MessageDrivenDeployment` attribute listenerThreads, as Example 10–4 shows.

For more information on this `@MessageDrivenDeployment` attribute, see Table A–3. For more information on the `@MessageDrivenDeployment` annotation, see "Configuring OC4J-Proprietary Deployment Options on an EJB 3.0 MDB" on page 10-17.

Note: Oracle recommends that you access a message service provider using a J2CA resource adapter such as the Oracle JMS Connector. For more information, see:

- "Restrictions When Accessing a Message Service Provider Without a J2CA Resource Adapter" on page 2-25.
- "Configuring an EJB 3.0 MDB to Access a Message Service Provider Using J2CA" on page 10-1

Example 10–4 Configuring Parallel Message Processing for a Non-J2CA Adapter Message Service Provider

```
import javax.ejb.MessageDriven;
import oracle.j2ee.ejb.MessageDrivenDeployment;
import javax.jms.Message;
import javax.jms.MessageListener;

@MessageDriven(
     ...
)

@MessageDrivenDeployment(
    listenerThreads=3
)

public class QueueMDB implements MessageListener {
    public void onMessage(Message msg) {
        ...
```

```
    }
}
```

Using Deployment XML

For an EJB 3.0 message-driven bean, you configure parallel message processing in the `orion-ejb-jar.xml` file as you would for an EJB 2.1 message-driven bean (see "Using Deployment XML" on page 18-7).

Configuring Maximum Delivery Count

You can configure the maximum number of times OC4J will attempt the immediate redelivery of a message to the message-driven bean's message listener method (for example, the `onMessage` method for a JMS message listener) if that method returns failure (fails to invoke an acknowledgment operation, throws an exception, or both).

After this number of redeliveries, the message is deemed undeliverable and is handled according to the policies of your message service provider. For example, OEMS JMS will put the message on its exception queue (`jms/Oc4jJmsExceptionQueue`).

You can configure the maximum delivery count using OC4J-proprietary annotations (see "Using Annotations" on page 10-7) or using the `orion-ejb-jar.xml` file (see "Using Deployment XML" on page 10-8).

For more information, see "Message Service Configuration Options: Annotations or XML? Attributes or Activation Configuration Properties?" on page 2-26.

Using Annotations

How you configure this option depends on the type of message-service provider you are using:

- Accessing a Message Service Provider Using a J2CA Resource Adapter
- Accessing a Message Service Provider Without Using a J2CA Resource Adapter

Accessing a Message Service Provider Using a J2CA Resource Adapter

If you access your message-service provider using a J2CA resource adapter, set activation configuration property `MaxDeliveryCnt`, as Example 10–5 shows.

For more information on `MaxDeliveryCnt`, see Table B–2.

Example 10–5 Configuring Maximum Delivery Count for a J2CA Adapter Message Service Provider

```
import javax.ejb.MessageDriven;
import oracle.j2ee.ejb.MessageDrivenDeployment;
import javax.ejb.ActivationConfigProperty;
import javax.jms.Message;
import javax.jms.MessageListener;

@MessageDriven(
    activationConfig = {
        @ActivationConfigProperty(propertyName="MaxDeliveryCnt", propertyValue="3"),
        ...
    }
)

@MessageDrivenDeployment(
    resourceAdapter = "OracleASjms",
```

```
    ...
)

public class JCAQueueMDB implements MessageListener {
    public void onMessage(Message msg) {
        ...
    }
}
```

Accessing a Message Service Provider Without Using a J2CA Resource Adapter

If you access your message-service provider directly (without using a J2CA resource adapter), set OC4J-proprietary annotation `@MessageDrivenDeployment` attribute maxDeliveryCount, as Example 10–6 shows.

For more information on this `@MessageDrivenDeployment` attribute, see Table A–3. For more information on the `@MessageDrivenDeployment` annotation, see "Configuring OC4J-Proprietary Deployment Options on an EJB 3.0 MDB" on page 10-17.

Note: Oracle recommends that you access a message service provider using a J2CA resource adapter such as the Oracle JMS Connector. For more information, see:

- "Restrictions When Accessing a Message Service Provider Without a J2CA Resource Adapter" on page 2-25.
- "Configuring an EJB 3.0 MDB to Access a Message Service Provider Using J2CA" on page 10-1

Example 10–6 Configuring Maximum Delivery Count for a Non-J2CA Adapter Message Service Provider

```
import javax.ejb.MessageDriven;
import oracle.j2ee.ejb.MessageDrivenDeployment;
import javax.jms.Message;
import javax.jms.MessageListener;

@MessageDriven(
    ...
)

@MessageDrivenDeployment(
    maxDeliveryCount=3
)

public class QueueMDB implements MessageListener {
    public void onMessage(Message msg) {
        ...
    }
}
```

Using Deployment XML

For an EJB 3.0 message-driven bean, you configure the maximum delivery count in the `orion-ejb-jar.xml` file as you would for an EJB 2.1 message-driven bean (see "Using Deployment XML" on page 18-8).

Configuring Connection Failure Recovery for an EJB 3.0 MDB

You can configure how a message-driven bean's listener thread responds to connection failures due to such events as network and JMS server outages.

These options are applicable to only container-managed transactions in a message-driven bean.

You can configure connection failure recovery options using OC4J-proprietary annotations (see "Using Annotations" on page 10-9) or using the `orion-ejb-jar.xml` file (see "Using Deployment XML" on page 10-10).

For more information, see:

- "Understanding OC4J EJB Application Clustering Services" on page 2-29
- "Message Service Configuration Options: Annotations or XML? Attributes or Activation Configuration Properties?" on page 2-26

Using Annotations

How you configure this option depends on the type of message-service provider you are using:

- Accessing a Message Service Provider Using a J2CA Resource Adapter
- Accessing a Message Service Provider Without Using a J2CA Resource Adapter

Accessing a Message Service Provider Using a J2CA Resource Adapter

If you access your message-service provider using a J2CA resource adapter, the Oracle JMS Connector does an infinite retry of polling for the JMS resource and this retry interval can be configured in the activation config property, `EndpointFailureRetryInterval` as Example 10–5 shows.

Note that the recovery of message after retry does not guarantee message ordering, and messages can be lost or duplicated when MDB subscription to the JMS topic is non-durable.

For more information, see `EndpointFailureRetryInterval` in Table B–2.

Example 10–7 Configuring Connection Failure Recovery for a J2CA Adapter Message Service Provider

```
import javax.ejb.MessageDriven;
import oracle.j2ee.ejb.MessageDrivenDeployment;
import javax.ejb.ActivationConfigProperty;
import javax.jms.Message;
import javax.jms.MessageListener;

@MessageDriven(
    activationConfig = {
        @ActivationConfigProperty(
            propertyName="EndpointFailureRetryInterval",
            propertyValue="20000"
        ),
        ...
    }
)

@MessageDrivenDeployment(
    resourceAdapter = "OracleASjms",
    ...
)
```

```
public class JCAQueueMDB implements MessageListener {
    public void onMessage(Message msg) {
        ...
    }
}
```

Accessing a Message Service Provider Without Using a J2CA Resource Adapter

If you access your message-service provider directly (without using a J2CA resource adapter), set OC4J-proprietary annotation `@MessageDrivenDeployment` attributes dequeueRetryCount and dequeueRetryInterval as Example 10–8 shows.

For more information on this `@MessageDrivenDeployment` attribute, see Table A–3. For more information on the `@MessageDrivenDeployment` annotation, see "Configuring OC4J-Proprietary Deployment Options on an EJB 3.0 MDB" on page 10-17.

Note: Oracle recommends that you access a message service provider using a J2CA resource adapter such as the Oracle JMS Connector. For more information, see:

- "Restrictions When Accessing a Message Service Provider Without a J2CA Resource Adapter" on page 2-25.
- "Configuring an EJB 3.0 MDB to Access a Message Service Provider Using J2CA" on page 10-1

Example 10–8 Configuring Connection Failure Recovery for a Non-J2CA Adapter Message Service Provider

```
import javax.ejb.MessageDriven;
import oracle.j2ee.ejb.MessageDrivenDeployment;
import javax.jms.Message;
import javax.jms.MessageListener;

@MessageDriven(
        ...
)

@MessageDrivenDeployment(
    dequeueRetryCount=3,
    dequeueRetryInterval=90
)

public class QueueMDB implements MessageListener {
    public void onMessage(Message msg) {
        ...
    }
}
```

Using Deployment XML

For an EJB 3.0 message-driven bean, you configure the dequeue retry in the `orion-ejb-jar.xml` file as you would for an EJB 2.1 message-driven bean (see "Using Deployment XML" on page 18-8).

Configuring a Life Cycle Callback Interceptor Method on an EJB 3.0 MDB

You can specify an EJB 3.0 message-driven bean class method as a callback method for any of the following life cycle events (see "Using Annotations" on page 10-11):

- Post-construct
- Pre-destroy

The message-driven bean class life cycle callback method must have the following signature:

```
void <METHOD>()
```

You can also specify one or more life cycle callback methods on an interceptor class that you associate with an EJB 3.0 message-driven bean (see "Configuring a Life Cycle Callback Interceptor Method on an Interceptor Class of an EJB 3.0 MDB" on page 10-11).

For more information, see the following:

- "What is the Life Cycle of a Message-Driven Bean?" on page 1-57
- "Life Cycle Callback Methods on a Bean Class" on page 1-6

Using Annotations

You can specify an EJB 3.0 message-driven bean class method as a life cycle callback method using any of the following annotations:

- `@PostConstruct`
- `@PreDestroy`

Example 10–9 shows how to use the `@PostConstruct` annotation to specify EJB 3.0 message-driven bean class method `initialize` as a life cycle callback method.

Example 10–9 @PostConstruct in an EJB 3.0 Message-Driven Bean

```
@MessageDriven
public class MessageLogger implements MessageListener {
    @Resource javax.ejb.MessageDrivenContext mc;

    public void onMessage(Message message) {
    ....
    }

    @PostConstruct
    public void initialize() {
        // Initialization logic
    }
...
}
```

Configuring a Life Cycle Callback Interceptor Method on an Interceptor Class of an EJB 3.0 MDB

You can designate an interceptor method on an interceptor class of an EJB 3.0 message-driven bean as a life cycle callback interceptor method.

To configure a life cycle callback interceptor method on an interceptor class, you must do the following:

1. Create an interceptor class.

   This can be any POJO class.

   An interceptor class must have a public no-argument constructor.

2. Implement the life cycle callback interceptor method.

   Callback methods defined on a bean's interceptor class have the following signature:

   ```
   Object <METHOD>(InvocationContext)
   ```

3. Associate a life cycle event with the callback interceptor method.

   A life cycle event can only be associated with one callback interceptor method, but a life cycle callback interceptor method may be used to interpose on multiple callback events. For example, `@PostConstruct` and `@PreDestroy` may appear only once in an interceptor class but you may associate both `@PostConstruct` and `@PreDestroy` with the same callback interceptor method.

   For more information, see the following:

   - "Using Annotations" on page 10-12

4. Associate the interceptor class with your EJB 3.0 message-driven bean (see "Configuring an Interceptor Class for an EJB 3.0 MDB" on page 10-15).

For more information, see the following:

- "What is the Life Cycle of a Message-Driven Bean?" on page 1-57
- "Life Cycle Callback Interceptor Methods on an EJB 3.0 Interceptor Class" on page 1-6

Using Annotations

You can specify an interceptor class method as an EJB 3.0 message-driven bean life cycle callback method using any of the following annotations:

- `@PostConstruct`
- `@PreDestroy`

Example 10–10 shows an interceptor class using `@PostConstruct` and `@PreDestroy` annotations to identify `myPostConstructMethod` and `myPreDestroyMethod` as life cycle callback interceptor methods. OC4J invokes the appropriate life cycle method only when the appropriate life cycle event occurs. OC4J invokes all other non-life cycle interceptor methods (such as `myInterceptorMethod`) each time you invoke a message-driven bean business method (see "Configuring an Interceptor Class for an EJB 3.0 MDB" on page 10-15).

Example 10–10 Interceptor Class

```
public class MyInterceptor {
    ...
    public void myInterceptorMethod (InvocationContext ctx) {
        ...
        ctx.proceed();
        ...
    }

    @PostConstruct
    public void myPostContructMethod (InvocationContext ctx) {
        ...
```

```
        ctx.proceed();
        ...
    }

    @PreDestroy
    public void myPreDestroyMethod (InvocationContext ctx) {
        ...
        ctx.proceed();
        ...
    }
}
```

Configuring an Around Invoke Interceptor Method on an EJB 3.0 MDB

You can specify one nonbusiness method as the interceptor method for an EJB 3.0 message-driven bean. Each time the `onMessage` method is invoked, OC4J intercepts the invocation and invokes the interceptor method. The `onMessage` method invocation proceeds only if the interceptor method returns `InvocationContext.proceed()`.

An interceptor method has the following signature:

```
Object <METHOD>(InvocationContext) throws Exception
```

An interceptor method may have public, private, protected, or package level access, but must not be declared as final or static.

You can specify this method on the EJB 3.0 message-driven bean class (see "Using Annotations" on page 10-13) or on an interceptor class that you associate with an EJB 3.0 message-driven bean (see "Configuring an Around Invoke Interceptor Method on an Interceptor Class of an EJB 3.0 MDB" on page 10-14).

For more information, see "Understanding EJB 3.0 Interceptors" on page 2-10.

Using Annotations

Example 10–11 shows how to designate a method of a message-driven bean class as an interceptor method using the `@AroundInvoke` annotation. Each time the `onMessage` method is invoked, OC4J intercepts the invocation and invokes the interceptor method `myInterceptor`. The `onMessage` method invocation proceeds only if the interceptor method returns `InvocationContext.proceed()`.

Example 10–11 @AroundInvoke in an EJB 3.0 Message-Driven Bean

```
@MessageDriven
public class MessageLogger implements MessageListene {
    @Resource javax.ejb.MessageDrivenContext mc;

    public void onMessage(Message message) {
    ....
    }

    @AroundInvoke
    public Object myInterceptor(InvocationContext ctx) throws Exception  {
        if (!userIsValid(ctx. getEJBContext().getCallerPrincipal())) {
            throw new SecurityException(
                "Caller: '" + ctx.getEJBContext().getCallerPrincipal().getName() +
                "' does not have permissions for method " + ctx.getMethod()
            );
        }
        return ctx.proceed();
```

```
    }
}
```

Configuring an Around Invoke Interceptor Method on an Interceptor Class of an EJB 3.0 MDB

You can specify one nonbusiness method as the interceptor method for an EJB 3.0 message-driven bean. Each time the `onMessage` method is invoked, OC4J intercepts the invocation and invokes the interceptor method. The `onMessage` invocation proceeds only if the interceptor method returns `InvocationContext.proceed()`.

You can specify this method on an interceptor class that you associate with an EJB 3.0 MDB or on the EJB 3.0 MDB class itself (see "Configuring an Around Invoke Interceptor Method on an EJB 3.0 MDB" on page 10-13).

To configure an interceptor method on an interceptor class, you must do the following:

1. Create an interceptor class.

   This can be any POJO class.

2. Implement the interceptor method.

   An interceptor method has the following signature:

   ```
   Object <METHOD>(InvocationContext) throws Exception
   ```

   An interceptor method may have public, private, protected, or package level access, but must not be declared as final or static.

3. Designate the method as the interceptor method (see "Using Annotations" on page 10-14).

4. Associate the interceptor class with your EJB 3.0 MDB (see "Configuring an Interceptor Class for an EJB 3.0 MDB" on page 10-15).

For more information, see "Understanding EJB 3.0 Interceptors" on page 2-10.

Using Annotations

Example 10–12 shows how to specify interceptor class method `myInterceptor` as the interceptor method of an EJB 3.0 MDB using the `@AroundInvoke` annotation. After you associate this interceptor class with an MDB ("Configuring an Interceptor Class for an EJB 3.0 MDB" on page 10-15), each time the `onMessage` method is invoked, OC4J intercepts the invocation and invokes the interceptor method `myInterceptor`. The `onMessage` method invocation proceeds only if the interceptor method returns `InvocationContext.proceed()`.

Example 10–12 Interceptor Class

```
public class MyInterceptor {
    ...
    @AroundInvoke
    protected Object myInterceptor(InvocationContext ctx) throws Exception {
        Principal p = ctx.getEJBContext().getCallerPrincipal;
        if (!userIsValid(p)) {
            throw new SecurityException(
                "Caller: '" + p.getName() +
                "' does not have permissions for method " + ctx.getMethod()
            );
        }
```

```
        return ctx.proceed();
    }

    @PreDestroy
    public void myPreDestroyMethod (InvocationContext ctx) {
        ...
        ctx.proceed();
        ...
    }
}
```

Configuring an Interceptor Class for an EJB 3.0 MDB

An interceptor class is a class, distinct from the bean class itself, whose methods are invoked in response to business method invocations and life cycle events on the bean. You can associate a bean class can with any number of interceptor classes.

You can associate an interceptor class with an EJB 3.0 message-driven bean.

To configure an EJB 3.0 message-driven bean with an interceptor class, you must do the following:

1. Create an interceptor class (see "Creating an Interceptor Class" on page 10-15).

   This can be any POJO class.

2. Implement interceptor methods in the interceptor class.

   An interceptor method has the following signature:

   ```
   Object <METHOD>(InvocationContext) throws Exception
   ```

   An interceptor method may have public, private, protected, or package level access, but must not be declared as final or static.

   You can annotate an interceptor method as a life cycle callback (see "Configuring a Life Cycle Callback Interceptor Method on an Interceptor Class of an EJB 3.0 MDB" on page 10-11) or as an `AroundInvoke` method (see "Configuring an Around Invoke Interceptor Method on an Interceptor Class of an EJB 3.0 MDB" on page 10-14).

3. Associate the interceptor class with your EJB 3.0 message-driven bean (see "Associating an Interceptor Class With an MDB" on page 10-16).

4. Optionally configure the message-driven bean to use singleton interceptors (see "Specifying Singleton Interceptors in an MDB" on page 10-17).

Using Annotations

This section describes the following:

- Creating an Interceptor Class
- Associating an Interceptor Class With an MDB
- Specifying Singleton Interceptors in an MDB

Creating an Interceptor Class

Example 10–13 shows how to specify an `AroundInvoke` interceptor method and a life cycle callback interceptor method in an interceptor class for an EJB 3.0 message-driven bean. After you associate this interceptor class with a message-driven bean (see Example 10–14), each time the `onMessage` method is invoked, OC4J intercepts the

invocation and invokes the `AroundInvoke` method `myInterceptor`. When the appropriate life cycle event occurs, OC4J invokes the corresponding life cycle callback interceptor method such as `myPreDestroyMethod`.

Example 10–13 Interceptor Class

```
public class MyInterceptor {
    ...
    @AroundInvoke
    protected Object myInterceptor(InvocationContext ctx) throws Exception {
        Principal p = ctx.getEJBContext().getCallerPrincipal;
        if (!userIsValid(p)) {
            throw new SecurityException(
                "Caller: '" + p.getName() +
                "' does not have permissions for method " + ctx.getMethod()
            );
        }
        return ctx.proceed();
    }

    @PreDestroy
    public void myPreDestroyMethod (InvocationContext ctx) {
        ...
        ctx.proceed();
        ...
    }
}
```

Associating an Interceptor Class With an MDB

You can associate an interceptor class with an EJB 3.0 message-driven bean using the `@Interceptors` annotation. Example 10–14 shows how to associate the interceptor class from Example 10–13 with an EJB 3.0 message-driven bean class.

Note that the life cycle method for `@PostConstruct` is a method of the EJB 3.0 message-driven bean class itself (for more information, see "Configuring a Life Cycle Callback Interceptor Method on an EJB 3.0 MDB" on page 10-11) while the life cycle method for `@PreDestroy` is a life cycle callback interceptor method on the interceptor class associated with this message-driven bean (see "Configuring a Life Cycle Callback Interceptor Method on an Interceptor Class of an EJB 3.0 MDB" on page 10-11).

Example 10–14 Associating an Interceptor Class With an EJB 3.0 MDB

```
@MessageDriven
@Interceptors(MyInterceptor.class)
public class MessageLogger implements MessageListener {
    @Resource javax.ejb.MessageDrivenContext mc;

    public void onMessage(Message message) {
    ....
    }

    @PostConstruct
    public void initialize() {
        items = new ArrayList();
    }
    ...
}
```

Specifying Singleton Interceptors in an MDB

As Example 10–15 shows, you can configure OC4J to use singleton interceptor classes by setting the `@MessageDrivenDeployment` attribute interceptorType to `singleton`. All instances of this message-driven bean will share the same instance of `MyInterceptor`. The `MyInterceptor` class must be stateless.

For more information about this attribute, see Table A–3. For more information on singleton interceptors, see "Singleton Interceptors" on page 2-12.

Example 10–15 Specifying a Singleton Interceptor Class With an EJB 3.0 MDB

```
@MessageDriven
@MessageDrivenDeployment(interceptorType="singleton")
@Interceptors(MyInterceptor.class)
public class MessageLogger implements MessageListener {
    @Resource javax.ejb.MessageDrivenContext mc;

    public void onMessage(Message message) {
    ....
    }

    @PostConstruct
    public void initialize() {
        items = new ArrayList();
    }
    ...
}
```

Configuring OC4J-Proprietary Deployment Options on an EJB 3.0 MDB

You can configure OC4J-proprietary deployment options for an EJB 3.0 message-driven bean using OC4J-proprietary annotations (see "Using Annotations" on page 10-17) or using the `orion-ejb-jar.xml` file (see "Using Deployment XML" on page 10-18).

Configuration in the `orion-ejb-jar.xml` file overrides the corresponding configuration made with OC4J-proprietary annotations.

For more information, see "Message Service Configuration Options: Annotations or XML? Attributes or Activation Configuration Properties?" on page 2-26.

Using Annotations

You can specify OC4J-proprietary deployment options for an EJB 3.0 message-driven bean using the `@MessageDrivenDeployment` OC4J-proprietary annotation.

Example 10–16 shows how to configure OC4J-proprietary deployment options for an EJB 3.0 message-driven bean using the `@MessageDrivenDeployment` annotation. For more information on `@MessageDrivenDeployment` attributes, see Table A–3.

You can override `@MessageDriven` annotation `activationConfig` attribute settings (see "Configuring an EJB 3.0 MDB to Access a Message Service Provider Using J2CA" on page 10-1) by configuring activation configuration properties using `@MessageDrivenDeployment` attributes. You can also override annotation configuration using deployment XML (see "Using Deployment XML" on page 10-18).

Example 10–16 @MessageDrivenDeployment

```
import javax.ejb.MessageDriven;
import oracle.j2ee.ejb.MessageDrivenDeployment;
```

```
import javax.ejb.ActivationConfigProperty;
import javax.annotation.Resource;

@MessageDriven(
    activationConfig = {
        @ActivationConfigProperty(
            propertyName="messageListenerInterface",
            propertyValue="javax.jms.MessageListener"),
        @ActivationConfigProperty(
            propertyName="connectionFactoryJndiName",
            propertyValue="jms/TopicConnectionFactory"),
         @ActivationConfigProperty(
            propertyName="destinationName",
            propertyValue="jms/demoTopic"),
         @ActivationConfigProperty(
            propertyName="destinationType",
            propertyValue="javax.jms.Topic"),
         @ActivationConfigProperty(
            propertyName="messageSelector",
            propertyValue="RECIPIENT = 'MDB'")
    }
)
@MessageDrivenDeployment(
    maxInstances=10,
    poolCacheTimeout=30
)
public class MessageLogger implements MessageListener, TimedObject {
    @Resource javax.ejb.MessageDrivenContext mc;

    public void onMessage(Message message) {
    ...
    }

    public void ejbTimeout(Timer timer) {
    ...
    }
}
```

Using Deployment XML

You can specify OC4J-proprietary deployment options for a message-driven bean using the `orion-ejb-jar.xml` file element `<message-driven-deployment>` as Example 10–17 shows. For more information on the `<message-driven-deployment>` element, see "<message-driven-deployment>" on page A-17.

Example 10–17 orion-ejb-jar.xml File <message-driven-deployment> Element

```
<?xml version="1.0" encoding="utf-8"?>
<orion-ejb-jar
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/orion-ejb-jar-10_
0.xsd"
    deployment-version="10.1.3.1.0"
    deployment-time="10b1fb5cdd0"
    schema-major-version="10"
    schema-minor-version="0"
>
    <enterprise-beans>
        <message-driven-deployment
            name="MessageLogger"
            max-instances="10"
            cache-timeout="30"
            ...
```

```
        >
        </message-driven-deployment>
    ...
    </enterprise-beans>
    ...
</orion-ejb-jar>
```

Part V

EJB 2.1 Session Beans

This part provides procedural information on implementing and configuring EJB 2.1 session beans. For conceptual information, see Part I, "EJB Overview".

This part contains the following chapters:

- Chapter 11, "Implementing an EJB 2.1 Session Bean"
- Chapter 12, "Using an EJB 2.1 Session Bean"

11

Implementing an EJB 2.1 Session Bean

This chapter explains how to implement an EJB 2.1 session bean, including the following:

- "Implementing an EJB 2.1 Stateless Session Bean" on page 11-1
- "Implementing an EJB 2.1 Stateful Session Bean" on page 11-3

> **Note:** You can download EJB code examples from: `http://www.oracle.com/technology/tech/java/oc4j/demos`.

For more information, see the following:

- "What is a Session Bean?" on page 1-27
- "Using an EJB 2.1 Session Bean" on page 12-1

Implementing an EJB 2.1 Stateless Session Bean

Table 11–1 summarizes the important parts of an EJB 2.1 stateless session bean and the following procedure describes how to implement these parts. For a typical implementation, see "Using Java" on page 11-2. For more information, see "What is a Stateless Session Bean?" on page 1-28.

Table 11–1 Parts of an EJB 2.1 Stateless Session Bean

Part	Description
Home Interface (remote or local)	Extends `javax.ejb.EJBHome` and `javax.ejb.EJBLocalHome` and requires a single `create()` factory method, with no arguments, and a single `remove()` method.
Component Interface (remote or local)	Extends `javax.ejb.EJBObject` for the remote interface and `javax.ejb.EJBLocalObject` for the local interface. It defines the business logic methods, which are implemented in the bean implementation.
`TimedObject` Interface	Optionally implements the `javax.ejb.TimedObject` interface. For more information, see "Understanding EJB Timer Services" on page 2-31).
Bean implementation	Implements `SessionBean`. This class must be declared as public, contain a public, empty, default constructor, no `finalize`() method, and implements the methods defined in the component interface. Must contain a single `ejbCreate` method, with no arguments, to match the `create`() method in the home interface. Contains empty implementations for the container service methods, such as `ejbRemove`, and so on.

1. Create the home interfaces for the bean (see "Implementing the Home Interfaces" on page 11-6).

The remote home interface defines the `create` method that a client can invoke remotely to instantiate your bean. The local home interface defines the `create` method that a collocated bean can invoke locally to instantiate your bean.

a. To create the remote home interface, extend `javax.ejb.EJBHome` (see "Implementing the Remote Home Interface" on page 11-6).

b. To create the local home interface, extend `javax.ejb.EJBLocalHome` (see "Implementing the Local Home Interface" on page 11-7).

2. Create the component interfaces for the bean (see "Implementing the Component Interfaces" on page 11-8).

The remote component interface declares the business methods that a client can invoke remotely. The local interface declares the business methods that a collocated bean can invoke locally.

a. To create the remote component interface, extend `javax.ejb.EJBObject` (see "Implementing the Remote Component Interface" on page 11-8).

b. To create the local component interface, extend `javax.ejb.EJBLocalObject` (see "Implementing the Local Component Interface" on page 11-9).

3. Implement the stateless session bean as follows:

a. Implement a single `ejbCreate` method with no parameter that matches the home interface create method.

b. Implement the business methods that you declared in the home and component interfaces.

c. Implement the `javax.ejb.SessionBean` interface to implement the container callback methods it defines (see "Configuring a Life Cycle Callback Method for an EJB 2.1 Session Bean" on page 12-3).

d. Implement a `setSessionContext` method that takes an instance of `SessionContext` (see "Implementing the setSessionContext Method" on page 11-9).

For a stateless session bean, this method usually does nothing (does not actually add the `SessionContext` to the session bean's state).

4. Configure your `ejb-jar.xml` file to match your bean implementation (see "Using Deployment XML" on page 11-3).

Using Java

Example 11–1 shows a typical implementation of a stateless session bean.

Example 11–1 EJB 2.1 Stateless Session Bean Implementation

```
package hello;
import javax.ejb.*;

public class HelloBean implements SessionBean {

    /* ----------------------------------------
    * Begin business methods. The following methods
    * are called by the client code.
    * ---------------------------------------- */

    public String sayHello(String myName) throws EJBException {
        return ("Hello " + myName);
```

```
    }

    /* --------------------------------------
    * Begin private methods. The following methods
    * are used internally
    * ------------------------------------- */

...

    /* -----------------------------------------------------
    * Begin EJB-required methods. The following methods are called
    * by the container, and never called by client code
    * ----------------------------------------------------- */

    public void ejbCreate() throws CreateException {
        // when bean is created
    }

    public void setSessionContext(SessionContext ctx) {
    }

// Life Cycle Methods

    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }

    public void ejbCreate() {
    }

    public void ejbRemove() {
    }
}
```

Using Deployment XML

Example 11–2 shows the `ejb-jar.xml` session element corresponding to the stateless session bean shown in Example 11–1.

Example 11–2 ejb-jar.xml For a Stateless Session Bean

```
...
    <enterprise-beans>
        <session>
            <ejb-name>Hello</ejb-name>
            <home>hello.HelloHome</home>
            <remote>hello.Hello</remote>
            <ejb-class>hello.HelloBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
        </session>
    </enterprise-beans>
...
```

For more information on deployment files, see "Configuring Deployment Descriptor Files" on page 26-1.

Implementing an EJB 2.1 Stateful Session Bean

Table 11–2summarizes the important parts of an EJB 2.1 stateful session bean and the following procedure describes how to implement these parts. For a typical

implementation, see "Using Java" on page 11-5. For more information, see "What is a Stateful Session Bean?" on page 1-30.

Table 11–2 Parts of an EJB 2.1 Stateful Session Bean

Part	Description
Home Interface (remote or local)	Extends `javax.ejb.EJBHome` and `javax.ejb.EJBLocalHome` and requires one or more `create()` factory methods, and a single `remove()` method.
Component Interface (remote or local)	Extends `javax.ejb.EJBObject` for the remote interface and `javax.ejb.EJBLocalObject` for the local interface. It defines the business logic methods, which are implemented in the bean implementation.
Bean implementation	Implements `SessionBean`. This class must be declared as public, contain a public, empty, default constructor, no `finalize` method, and implement the methods defined in the remote interface. Must contain `ejbCreate` methods equivalent to the `create` methods defined in the home interface. That is, each `ejbCreate` method is matched–by its parameter signature–to a `create` method defined in the home interface. Implements the container service methods, such as `ejbRemove`, and so on. Also, optionally implements the `SessionSynchronization` interface for container-managed transactions, which includes `afterBegin`, `beforeCompletion`, and `afterCompletion`.

1. Create the home interfaces for the bean (see "Implementing the Home Interfaces" on page 11-6).

   The remote home interface defines the `create` method that a client can invoke remotely to instantiate your bean. The local home interface defines the `create` method that a collocated bean can invoke locally to instantiate your bean.

   a. To create the remote home interface, extend `javax.ejb.EJBHome` (see "Implementing the Remote Home Interface" on page 11-6).

   b. To create the local home interface, extend `javax.ejb.EJBLocalHome` (see "Implementing the Local Home Interface" on page 11-7).

2. Create the component interfaces for the bean (see "Implementing the Component Interfaces" on page 11-8).

   The remote component interface declares the business methods that a client can invoke remotely. The local interface declares the business methods that a collocated bean can invoke locally.

   a. To create the remote component interface, extend `javax.ejb.EJBObject` (see "Implementing the Remote Component Interface" on page 11-8).

   b. To create the local component interface, extend `javax.ejb.EJBLocalObject` (see "Implementing the Local Component Interface" on page 11-9).

3. Implement the stateless session bean as follows:

   a. Implement the `ejb<METHOD>` methods that match the home interface `create` methods.

      For a stateful session bean, provide `ejbCreate` methods with corresponding argument lists for each `create` method in the home interface.

   b. Implement the business methods that you declared in the home and component interfaces.

   c. Implement the `javax.ejb.SessionBean` interface to implement the container callback methods it defines (see "Configuring a Life Cycle Callback Method for an EJB 2.1 Session Bean" on page 12-3).

   d. Implement a `setSessionContext` method that takes an instance of `SessionContext` (see "Implementing the setSessionContext Method" on page 11-9).

For a stateful session bean, this method usually adds the `SessionContext` to the session bean's state.

4. Configure your `ejb-jar.xml` file to match your bean implementation (see "Using Deployment XML" on page 11-6).

Using Java

Example 11–3 shows a typical implementation of a stateful session bean.

Example 11–3 EJB 2.1 Stateful Session Bean Implementation

```
package hello;
import javax.ejb.*;

public class HelloBean implements SessionBean {
    /* ----------------------------------------
    * State
    * ---------------------------------------- */

    private SessionContext ctx;
    private Collection messages;
    private String defaultMessage = "Hello, World!";

    /* ----------------------------------------
    * Begin business methods. The following methods
    * are called by the client code.
    * ---------------------------------------- */

    public String sayHello(String myName) throws EJBException {
        return ("Hello " + myName);
    }

    public String sayHello() throws EJBException {
        return defaultMessage;
    }

    /* ----------------------------------------
    * Begin private methods. The following methods
    * are used internally.
    * ---------------------------------------- */

...

    /* ------------------------------------------------------
    * Begin EJB-required methods. The following methods are called
    * by the container, and never called by client code.
    * ------------------------------------------------------ */

    public void ejbCreate() throws CreateException {
        // when bean is created
    }

    public void ejbCreate(String message) throws CreateException {
        this.defaultMessage = message;
    }

    public void ejbCreate(Collection messages) throws CreateException {
        this.messages = messages;
    }

    public void setSessionContext(SessionContext ctx) {
        this.ctx = ctx;
    }
```

```
// Life Cycle Methods

    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }

    public void ejbCreate() {
    }

    public void ejbRemove() {
    }
}
```

Using Deployment XML

Example 11–4 shows the ejb-jar.xml session element corresponding to the stateful session bean shown in Example 11–3.

Example 11–4 ejb-jar.xml For a Stateful Session Bean

```
...
    <enterprise-beans>
        <session>
            <ejb-name>Hello</ejb-name>
            <home>hello.HelloHome</home>
            <remote>hello.Hello</remote>
            <ejb-class>hello.HelloBean</ejb-class>
            <session-type>Stateful</session-type>
            <transaction-type>Container</transaction-type>
        </session>
    </enterprise-beans>
...
```

For more information on deployment files, see "Configuring Deployment Descriptor Files" on page 26-1.

Implementing the Home Interfaces

The home interfaces (remote and local) are used to create the session bean instance; thus, they define the `create` method for your bean. As Table 11–3 shows, the type of create methods you define depends on the type of session bean you are creating:

Table 11–3 Home Interface Create Methods

Session Bean Type	Create Methods
Stateless Session Bean	Single `create` method only, with no parameters.
Stateful Session Bean	One or more create methods with whatever parameters define the bean's state.

For each `create` method, you define a corresponding `ejbCreate` method in the bean implementation.

Implementing the Remote Home Interface

A remote client invokes the EJB through its remote interface. The client invokes the `create` method that is declared within the remote home interface. The container passes the client call to the `ejbCreate` method–with the appropriate parameter signature–within the bean implementation. The requirements for developing the remote home interface include:

- The remote home interface must extend the `javax.ejb.EJBHome` interface.
- All `create` methods may throw the following exceptions:
  - `javax.ejb.CreateException`
  - `javax.ejb.RemoteException`
  - optional application exceptions
- All create methods should not throw the following exceptions:
  - `javax.ejb.EJBException`
  - `java.lang.RunTimeException`

Example 11–5 shows a remote home interface called `HelloHome` for a stateless session bean.

Example 11–5 Remote Home Interface for a Stateless Session Bean

```
package hello;

import javax.ejb.*;
import java.rmi.*;

public interface HelloHome extends EJBHome {
  public Hello create() throws CreateException, RemoteException;
}
```

Example 11–6 shows a remote home interface called `HelloHome` for a stateful session bean. You use the arguments passed into the various `create` methods to initialize the session bean's state.

Example 11–6 Remote Home Interface for a Stateful Session Bean

```
package hello;

import javax.ejb.*;
import java.rmi.*;

public interface HelloHome extends EJBHome {
  public Hello create() throws CreateException, RemoteException;
  public Hello create(String message) throws CreateException, RemoteException;
  public Hello create(Collection messages) throws CreateException, RemoteException;
}
```

Implementing the Local Home Interface

An EJB can be called locally from a client that exists in the same container. Thus, a collocated bean, JSP, or servlet invokes the `create` method that is declared within the local home interface. The container passes the client call to the `ejbCreate` method–with the appropriate parameter signature–within the bean implementation. The requirements for developing the local home interface include the following:

- The local home interface must extend the `javax.ejb.EJBLocalHome` interface.
- All `create` methods may throw the following exceptions:
  - `javax.ejb.CreateException`
  - `javax.ejb.RemoteException`
  - optional application exceptions

- All create methods should not throw the following exceptions:
  - `javax.ejb.EJBException`
  - `java.lang.RunTimeException`

Example 11–7 shows a local home interface called `HelloLocalHome` for a stateless session bean.

Example 11–7 Local Home Interface for a Stateless Session Bean

```
package hello;

import javax.ejb.*;

public interface HelloLocalHome extends EJBLocalHome {
 public HelloLocal create() throws CreateException;
}
```

Example 11–8 shows a local home interface called `HelloLocalHome` for a stateful session bean. You use the arguments passed into the various `create` methods to initialize the session bean's state.

Example 11–8 Local Home Interface for a Stateful Session Bean

```
package hello;

import javax.ejb.*;

public interface HelloLocalHome extends EJBLocalHome {
  public HelloLocal create() throws CreateException;
  public HelloLocal create(String message) throws CreateException;
  public HelloLocal create(Collection messages) throws CreateException;
}
```

Implementing the Component Interfaces

The component interfaces define the business methods of the bean that a client can invoke.

Implementing the Remote Component Interface

The remote interface defines the business methods that a remote client can invoke. The requirements for developing the remote component interface include: the following

- The remote component interface of the bean must extend the `javax.ejb.EJBObject` interface, and its methods must throw the `java.rmi.RemoteException` exception.
- You must declare the remote interface and its methods as `public` for remote clients.
- The remote component interface, all its method parameters, and return types must be serializable. In general, any object that is passed between the client and the EJB must be serializable, because RMI marshals and unmarshalls the object on both ends.
- Any exception can be thrown to the client, as long as it is serializable. Run-time exceptions, including `EJBException` and `RemoteException`, are transferred back to the client as remote run-time exceptions.

Example 11–9 shows a remote component interface called `Hello` with its defined methods, each of which will be implemented in the corresponding session bean.

Example 11–9 Remote Component Interface for EJB 2.1 Session Bean

```
package hello;

import javax.ejb.*;
import java.rmi.*;

public interface Hello extends EJBObject {
    public String sayHello(String myName) throws RemoteException;
    public String sayHello() throws RemoteException;
}
```

Implementing the Local Component Interface

The local component interface defines the business methods of the bean that a local (collocated) client can invoke. The requirements for developing the local component interface include the following:

- The local component interface of the bean must extend the `javax.ejb.EJBLocalObject` interface.
- You declare the local component interface and its methods as `public`.

Example 11–10 shows a local component interface called `HelloLocal` with its defined methods, each of which will be implemented in the corresponding session bean.

Example 11–10 Local Component Interface for EJB 2.1 Session Bean

```
package hello;

import javax.ejb.*;

public interface HelloLocal extends EJBLocalObject {
    public String sayHello(String myName);
    public String sayHello();
}
```

Implementing the setSessionContext Method

You use this method to obtain a reference to the context of the bean. A session bean has a session context that the container maintains and makes available to the bean. The bean may use the methods in the session context to make callback requests to the container.

The container invokes `setSessionContext` method, after it first instantiates the bean, to enable the bean to retrieve the session context. The container will never call this method from within a transaction context. If the bean does not save the session context at this point, the bean will never gain access to the session context.

When the container calls this method, it passes the reference of the `SessionContext` object to the bean. The bean can then store the reference for later use.

Example 11–11 shows a session bean saving the session context in the `sessctx` variable.

Example 11–11 Implementing the setSessionContext Method

```
import javax.ejb.*;

public class myBean implements SessionBean {
   SessionContext sessctx;
```

```
   public void setSessionContext(SessionContext ctx) {
      sessctx = ctx;   // session context is stored in instance variable
   }
   // other methods in the bean
}
```

12

Using an EJB 2.1 Session Bean

This chapter describes the various options that you must configure in order to use an EJB 2.1 session bean.

Table 12–1 lists these options and indicates which are basic (applicable to most applications) and which are advanced (applicable to more specialized applications).

For more information, see the following:

- "What is a Session Bean?" on page 1-27
- "Implementing an EJB 2.1 Session Bean" on page 11-1

Table 12–1 Configurable Options for an EJB 2.1 Session Bean

Options	Type
"Configuring Passivation" on page 12-1	Advanced
"Configuring Passivation Criteria" on page 12-2	Advanced
"Configuring Passivation Location" on page 12-3	Advanced
"Configuring Bean Instance Pool Size" on page 31-4	Basic
"Configuring Bean Instance Pool Timeouts for Session Beans" on page 31-6	Advanced
"Configuring a Transaction Timeout for a Session Bean" on page 21-6	Advanced
"Configuring a Life Cycle Callback Method for an EJB 2.1 Session Bean" on page 12-3	Basic

Configuring Passivation

You can enable and disable passivation for stateful session beans (see "Using Deployment XML" on page 12-2).

You may choose to disable passivation for any of the following reasons:

- Incompatible object types: if you cannot represent the nontransient attributes of your stateful session bean with object types supported by passivation (see "What Object Types can be Passivated?" on page 1-33), you can exchange increased memory consumption for the use of other object types by disabling passivation.
- Performance: if you determine that passivation is a performance problem in your application, you can exchange increased memory consumption for improved performance by disabling passivation.
- Secondary storage limitations: if you cannot provide sufficient secondary storage (see "Configuring Passivation Location" on page 12-3), you can exchange increased memory consumption for reduced secondary storage requirements by disabling passivation.

For more information, see the following:

- "When Does Stateful Session Bean Passivation Occur?" on page 1-32
- "Configuring Passivation Criteria" on page 12-2
- "Configuring Passivation Location" on page 12-3

Using Deployment XML

Table 12–2 lists the attributes, values, and defaults for configuring passivation in the `server.xml` file element `sfsb-config`.

Table 12–2 server.xml Element sfsb-config Passivation Configuration

Attribute	Values	Default
enable-passivation	`true`, `false`	`true`

Configuring Passivation Criteria

You can specify under what conditions OC4J passivates a stateful session bean (see "Using Deployment XML" on page 12-2).

For more information, see: the following

- "When Does Stateful Session Bean Passivation Occur?" on page 1-32
- "Configuring Passivation" on page 12-1
- "Configuring Passivation Location" on page 12-3

Using Deployment XML

Table 12–3 lists the attributes, values, and defaults for configuring passivation criteria in the `orion-ejb-jar.xml` file element `session-deployment`.

Table 12–3 orion-ejb-jar.xml Element session-deployment Passivation Criteria

Attribute	Values	Default
idletime	Positive, integer number of seconds before passivation occurs. To disable this criteria, specify a value of `never`.	`300`
memory-threshold	Percentage of JVM memory that can be consumed before passivation occurs. To disable this criteria, specify a value of never.	`80`
max-instances	Maximum positive integer number of bean instances allowed in memory: either instantiated or pooled. When this value is reached, OC4J attempts to passivate beans using the least recently used (LRU) algorithm. To allow an infinite number of bean instances, the `max-instances` attribute can be set to zero. Default is 0, which means infinite. This applies to both stateless and stateful session beans. To disable instance pooling, set `max-instances` to any negative number. This will create a new instance at the start of the EJB call and release it at the end of the call. See "Configuring Bean Instance Pool Size" on page 31-4 for more information.	`0` (unlimited)

Table 12–3 (Cont.) orion-ejb-jar.xml Element session-deployment Passivation Criteria

Attribute	Values	Default
max-instances-threshold	Percentage of `max-instances` number of beans that can be in memory before passivation occurs. Specify an integer that is translated as a percentage. If you define that the `max-instances` is 100 and the `max-instances-threshold` is 90%, then when the active bean instances is greater than or equal to 90, passivation of beans occurs. Default: 90%. To disable, specify never	`90`
passivate-count	Positive, integer number of beans to be passivated if any of the resource thresholds (`memory-threshold` or `max-instances-threshold`) have been reached. Passivation of beans is performed using the least recently used algorithm. To disable this option, specify a value of `0`.	One-third of `max-instances`
resource-check-interval	The frequency, as a positive, integer number of seconds, at which OC4J checks resource thresholds (`memory-threshold` or `max-instances-threshold`). To disable this option, specify a value of never.	`180`

Configuring Passivation Location

You can specify the directory and file name to which OC4J serializes a stateful session bean when passivated (see "Using Deployment XML" on page 12-3).

For more information, see the following:

- "Where is a Passivated Stateful Session Bean Stored?" on page 1-34
- "Configuring Passivation" on page 12-1
- "Configuring Passivation Criteria" on page 12-2

Using Deployment XML

Table 12–4 lists the attributes, values, and defaults for configuring passivation location in the `orion-ejb-jar.xml` file element `session-deployment`.

Table 12–4 orion-ejb-jar.xml Element session-deployment Passivation Location Configuration

Attribute	Values	Default
persistence-filename	Fully qualified path and file name of the file into which OC4J serializes bean instances during passivation.	`<OC4J_HOME>\j2ee\home\application-deployments\persistence`.

Configuring a Life Cycle Callback Method for an EJB 2.1 Session Bean

The following are the EJB 2.1 life cycle methods, as specified in the `javax.ejb.SessionBean` interface, that a session bean must implement (see "Using Java" on page 12-4):

- `ejbCreate`
- `ejbActivate` (stateful session beans only)
- `ejbPassivate` (stateful session beans only)
- `ejbRemove`
- `setSessionContext`

Note: Using EJB 2.1, you must implement all session bean callback methods. If you do not need to take any action, or if the callback method does not apply to your session bean, implement an empty method.

For more information, see the following:

- "What is the Stateless Session Bean Life Cycle?" on page 1-28
- "What is the Life Cycle of a Stateful Session Bean?" on page 1-30

Using Java

Example 12–1 shows how to implement an EBJ 2.1 session bean callback method.

Example 12–1 EJB 2.1 Session Bean Callback Method Implementation

```
public void ejbActivate() {
    // when bean is activated
}
```

Part VI

EJB 2.1 Entity Beans

This part provides procedural information on implementing and configuring EJB 2.1 entity beans and entity bean queries. For conceptual information, see Part I, "EJB Overview".

This part contains the following chapters:

13

Implementing an EJB 2.1 Entity Bean

This chapter explains how to implement an EJB 2.1 entity bean, including the following:

- "Implementing an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 13-1
- "Implementing an EJB 2.1 Entity Bean With Bean-Managed Persistence" on page 13-6

> **Note:** You can download EJB code examples from: `http://www.oracle.com/technology/tech/java/oc4j/demos`.

For more information, see the following:

- "What is an EJB 2.1 Entity Bean?" on page 1-41
- "Using an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 14-1
- "Using an EJB 2.1 Entity Bean With Bean-Managed Persistence" on page 15-1

Implementing an EJB 2.1 Entity Bean With Container-Managed Persistence

Table 13–1 summarizes the important parts of an EJB 2.1 entity bean with container-managed persistence and the following procedure describes how to implement these parts. For a typical implementation, see "Using Java" on page 13-3. For more information, see "What is an EJB 2.1 Entity Bean With Container-Managed Persistence?" on page 1-42.

Table 13–1 Parts of an EJB 2.1 Entity Bean With Container-Managed Persistence

Part	Description
Home Interface (remote or local)	Extends `javax.ejb.EJBHome` for the remote home interface, `javax.ejb.EJBLocalHome` for the local home interface, and requires a single `create` factory method, with no arguments, and a single `remove` method.
Component Interface (remote or local)	Extends `javax.ejb.EJBObject` for the remote interface and `javax.ejb.EJBLocalObject` for the local interface. It defines the business logic methods, which are implemented in the bean implementation.
Bean implementation	Implements `EntityBean`. This class must be declared as public, contain a public, empty, default constructor, no `finalize` method, and implements the methods defined in the component interface. Must contain one or more `ejbCreate` methods to match the `create` methods in the home interface. Contains empty implementations for the container service methods, such as `ejbRemove`, and so on.

1. Create the home interfaces for the bean (see "Implementing the EJB 2.1 Home Interfaces" on page 13-18).

   The remote home interface defines the `create` and `finder` methods that a client can invoke remotely to instantiate your bean. The local home interface defines the `create` and `finder` methods that a collocated bean can invoke locally to instantiate your bean.

   For more information about finders, see "Understanding Finder Methods" on page 1-53

   a. To create the remote home interface, extend `javax.ejb.EJBHome` (see "Implementing the Remote Home Interface" on page 13-18).

   b. To create the local home interface, extend `javax.ejb.EJBLocalHome` (see "Implementing the Local Home Interface" on page 13-19).

2. Create the component interfaces for the bean (see "Implementing the EJB 2.1 Component Interfaces" on page 13-19).

   The remote component interface declares the business methods that a client can invoke remotely. The local interface declares the business methods that a collocated bean can invoke locally.

   a. To create the remote component interface, extend `javax.ejb.EJBObject` (see "Implementing the Remote Component Interface" on page 13-19).

   b. To create the local component interface, extend `javax.ejb.EJBLocalObject` (see "Implementing the Local Component Interface" on page 13-20).

3. Define the primary key for the bean (see "Configuring a Primary Key for an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 14-2).

   The primary key identifies each entity bean instance and is a serializable class. You can use a simple data type class, such as `java.lang.String`, or define a complex class, such as one with two or more objects as components of the primary key.

4. Implement the entity bean with container-managed persistence as follows:

   a. Implement the abstract getter and setter methods that correspond to the getter and setter method(s) declared in the home interfaces.

      For an entity bean with container-managed persistence, the getter and setter methods are `public abstract`, because the container is responsible for their implementation.

   b. Implement the business methods that you declared in the home and component interfaces (if any). The signature for each of these methods must match the signature in the remote or local interface, except that the bean does not throw the `RemoteException`. Since both the local and the remote interfaces use the bean implementation, the bean implementation cannot throw the `RemoteException`.

      For an entity bean, these methods are often delegated to a session bean (see "What is a Session Bean?" on page 1-27).

   c. Implement any methods that are private to the bean or package used for facilitating the business logic. This includes private methods that your public methods use for completing the tasks requested of them.

   d. Implement the `ejbCreate` methods that correspond to the `create` method(s) declared in the home interfaces. The container invokes the

appropriate `ejbCreate` method when the client invokes the corresponding `create` method.

The return type of all `ebjCreate` methods is the type of the bean's primary key.

For an entity bean with container-managed persistence, provide create methods that allow the client to pass in values that the container will persist to your database.

e. Provide an empty implementation for each of the `javax.ejb.EntityBean` interface container callback methods.

For more information, see "Configuring a Life Cycle Callback Method for an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 14-15.

f. Implement a `setEntityContext` method (that takes an instance of `EntityContext`) and `unsetEntityContext` method (see "Implementing the setEntityContext and unsetEntityContext Methods" on page 13-20).

g. Optionally, define zero or more `public`, `abstract` select methods (see "Understanding Select Methods" on page 1-55) for use within the business methods of your entity bean.

5. Create the appropriate database schema (tables and columns) for the entity bean.

   For an entity bean with container-managed persistence, you can specify how persistence attributes should be stored in the database or you can configure the container to manage table creation for you.

   For more information, see the following:

   - "Configuring Table and Column Information" on page 14-4
   - "Configuring Automatic Database Table Creation" on page 14-5

6. Configure your `ejb-jar.xml` file to match your bean implementation and to reference a data source defined in your `data-sources.xml` file (see "Using Deployment XML" on page 13-5).

7. Complete the configuration of your entity bean (see "Using an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 14-1).

Using Java

Example 13–1 shows a typical implementation of an EJB 2.1 entity bean with container-managed persistence. Example 13–2 shows the corresponding remote home interface and Example 13–3 shows the corresponding remote component interface.

Example 13–1 Implementation of an EJB 2.1 Entity Bean With Container-Managed Persistence

```
package cmpapp;

import javax.ejb.*;
import java.rmi.*;

public abstract class EmployeeBean implements EntityBean {

    private EntityContext ctx;

    // container-managed persistent fields accessors
    public abstract Integer getEmpNo();
    public abstract void setEmpNo(Integer empNo);
```

```
    public abstract String getEmpName();
    public abstract void setEmpName(String empName);

    public abstract Float getSalary();
    public abstract void setSalary(Float salary);

    public void EmployeeBean() {
        // Empty constructor, don't initialize here but in the create().
        // passivate() may destroy these attributes in the case of pooling
    }

    public EmployeePK ejbCreate(Integer empNo, String empName, Float salary)
        throws CreateException {
        setEmpNo(empNo);
        setEmpName(empName);
        setSalary(salary);
        return new EmployeePK(empNo);
    }

    public void ejbPostCreate(Integer empNo, String empName, Float salary)
        throws CreateException {
        // when just after bean created
    }

    public void ejbStore() {
        // when bean persisted
    }

    public void ejbLoad() {
        // when bean loaded
    }

    public void ejbRemove() {
        // when bean removed
    }

    public void ejbActivate() {
        // when bean activated
    }

    public void ejbPassivate() {
        // when bean deactivated
    }

    public void setEntityContext(EntityContext ctx) {
        this.ctx = ctx;
    }

    public void unsetEntityContext() {
        this.ctx = null;
    }

}
```

Example 13–2 EJB 2.1 CMP Remote Home Interface

```
package cmpapp;

import java.rmi.*;
import java.util.*;
import javax.ejb.*;

public interface EmployeeHome extends EJBHome {
     public Employee create(Integer empNo, String empName, Float salary)
```

```
        throws CreateException, RemoteException;

    public Employee findByPrimaryKey(EmployeePK pk)
        throws FinderException, RemoteException;

    public Collection findByName(String empName)
        throws FinderException, RemoteException;

    public Collection findAll()
        throws FinderException, RemoteException;
}
```

Example 13–3 EJB 2.1 CMP Remote Component Interface

```
package cmpapp;

import javax.ejb.*;
import java.rmi.*;

public interface Employee extends EJBObject {
    // container-managed persistent fields accessors
    public Integer getEmpNo() throws RemoteException;
    public void setEmpNo(Integer empNo) throws RemoteException;

    public String getEmpName() throws RemoteException;
    public void setEmpName(String empName) throws RemoteException;

    public Float getSalary() throws RemoteException;
    public void setSalary(Float salary) throws RemoteException;
}
```

Using Deployment XML

Example 13–4 shows the `ejb-jar.xml` file `entity` element corresponding to the entity bean with container-managed persistence, shown in Example 13–1.

Example 13–4 ejb-jar.xml For an EJB 2.1 Entity Bean With Container-Managed Persistence

```
...
    <enterprise-beans>
        <entity>
            <description>no description</description>
            <display-name>EmployeeBean</display-name>
            <ejb-name>EmployeeBean</ejb-name>
            <home>cmpapp.EmployeeHome</home>
            <remote>cmpapp.Employee</remote>
            <ejb-class>cmpapp.EmployeeBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <cmp-version>2.x</cmp-version>
            <abstract-schema-name>EmployeeBean</abstract-schema-name>
            <prim-key-class>cmpapp.EmployeePK</prim-key-class>
            <reentrant>False</reentrant>
            <cmp-field><field-name>empNo</field-name></cmp-field>
            <cmp-field><field-name>empName</field-name></cmp-field>
            <cmp-field><field-name>salary</field-name></cmp-field>
            <query>
                <description></description>
                <query-method>
                <method-name>findAll</method-name>
                <method-params/>
                </query-method>
                <ejb-ql>Select OBJECT(e) From EmployeeBean e</ejb-ql>
```

```
            </query>
            <query>
                <description></description>
                <query-method>
                <method-name>findByName</method-name>
                <method-params>
                <method-param>java.lang.String</method-param>
                </method-params>
                </query-method>
                <ejb-ql>Select OBJECT(e) From EmployeeBean e where e.empName = ?1</ejb-ql>
            </query>
        </entity>
    </enterprise-beans>
...
```

Implementing an EJB 2.1 Entity Bean With Bean-Managed Persistence

Table 13–2 summarizes the important parts of an EJB 2.1 entity bean with bean-managed persistence. The following procedure describes how to implement these parts. For a typical implementation, see "Using Java" on page 13-8. For more information, see "What is an EJB 2.1 Entity Bean With Bean-Managed Persistence?" on page 1-46.

Table 13–2 Parts of an EJB 2.1 Entity Bean With Bean-Managed Persistence

Part	Description
Home Interface (remote or local)	Extends `javax.ejb.EJBHome` for the remote home interface, `javax.ejb.EJBLocalHome` for the local home interface, and requires a single `create` factory method, with no arguments, and a single `remove` method.
Component Interface (remote or local)	Extends `javax.ejb.EJBObject` for the remote interface and `javax.ejb.EJBLocalObject` for the local interface. It defines the business logic methods, which are implemented in the bean implementation.
Bean implementation	Implements `EntityBean`. This class must be declared as public, contain a public, empty, default constructor, no `finalize` method, and implements the methods defined in the component interface. Must contain one or more `ejbCreate` methods to match the `create` methods in the home interface. Contains complete implementations for the container service methods, such as `ejbStore`, `ejbLoad`, `ejbRemove`, and so on.

1. Create the home interfaces for the bean (see "Implementing the EJB 2.1 Home Interfaces" on page 13-18).

   The remote home interface defines the `create` method that a client can invoke remotely to instantiate your bean. The local home interface defines the `create` method that a collocated bean can invoke locally to instantiate your bean.

   a. To create the remote home interface, extend `javax.ejb.EJBHome` (see "Implementing the Remote Home Interface" on page 13-18).

   b. To create the local home interface, extend `javax.ejb.EJBLocalHome` (see "Implementing the Local Home Interface" on page 13-19).

2. Create the component interfaces for the bean (see "Implementing the EJB 2.1 Component Interfaces" on page 13-19).

   The remote component interface declares the business methods that a client can invoke remotely. The local interface declares the business methods that a collocated bean can invoke locally.

   a. To create the remote component interface, extend `javax.ejb.EJBObject` (see "Implementing the Remote Component Interface" on page 13-19).

b. To create the local component interface, extend `javax.ejb.EJBLocalObject` (see "Implementing the Local Component Interface" on page 13-20).

3. Define the primary key for the bean (see "Configuring a Primary Key for an EJB 2.1 Entity Bean With Bean-Managed Persistence" on page 15-1).

   The primary key identifies each entity bean instance and is a serializable class. You can use a simple data type class, such as `java.lang.String`, or define a complex class, such as one with two or more objects as components of the primary key.

4. Implement the entity bean with bean-managed persistence:

   a. Provide a complete implementation of the get and set methods that correspond to the get and set method(s) declared in the home interfaces.

      For an entity bean with bean-managed persistence, the getter and setter methods are public, because you are responsible for their implementation.

   b. Implement the business methods that you declared in the home and component interfaces (if any). The signature for each of these methods must match the signature in the remote or local interface, except that the bean does not throw the `RemoteException`. Since both the local and the remote interfaces use the bean implementation, the bean implementation cannot throw the `RemoteException`.

      For an entity bean, these methods are often delegated to a session bean (see "What is a Session Bean?" on page 1-27).

   c. Implement any methods that are private to the bean or package used for facilitating the business logic. This includes private methods that your public methods use for completing the tasks requested of them.

   d. Implement the `ejbCreate` methods that correspond to the `create` method(s) declared in the home interfaces. The container invokes the appropriate `ejbCreate` method when the client invokes the corresponding `create` method.

      The return type of all `ebjCreate` methods is the type of the bean's primary key.

      For an entity bean with bean-managed persistence, provide create methods that allow the client to pass in values that the container will persist to your database. You are responsible for providing an implementation that interacts with your database (usually through straight JDBC calls) to create an instance in the database.

      For more information, see "Implementing an ejbCreate Method for an EJB 2.1 Entity Bean With Bean-Managed Persistence" on page 13-15.

   e. Provide a complete implementation for each of the `javax.ejb.EntityBean` interface container callback methods (see "Configuring a Life Cycle Callback Method for an EJB 2.1 Entity Bean With Bean-Managed Persistence" on page 15-7).

      For an entity bean with bean-managed persistence, you are responsible for providing an implementation for each these methods that interacts with your database (usually through straight JDBC calls) to manage persistence in the database.

f. Implement a `setEntityContext` method that takes an instance of `EntityContext` and `unsetEntityContext` method (see "Implementing the setEntityContext and unsetEntityContext Methods" on page 13-20).

g. Implement the mandatory `findByPrimaryKey` finder method and, optionally, other finders (see "Configuring a Query for an EJB 2.1 Entity Bean With Bean-Managed Persistence" on page 15-5).

5. Create the appropriate database schema (tables and columns) for the entity bean.

   For an entity bean with bean-managed persistence, you are responsible for creating this schema in the database (defined in the `data-sources.xml` file) before your application attempts to create an instance of your entity bean with bean-managed persistence.

6. Configure your `ejb-jar.xml` file to match your bean implementation and to reference a data source defined in your `data-sources.xml` file (see "Using Deployment XML" on page 13-14).

7. Complete the configuration of your entity bean (see "Using an EJB 2.1 Entity Bean With Bean-Managed Persistence" on page 15-1).

Using Java

Example 13–5 shows a typical implementation of an EJB 2.1 entity bean with bean-managed persistence. Example 13–7 shows the corresponding home interface and Example 13–6 shows the corresponding remote interface.

Example 13–5 Implementation of an EJB 2.1 Entity Bean With Bean-Managed Persistence

```
package bmpapp;

import java.util.*;
import java.rmi.*;
import java.sql.*;
import javax.sql.*;
import javax.naming.*;
import javax.ejb.*;

public class EmployeeBean implements EntityBean {

    public Integer empNo;

    public EntityContext ctx;
    private Connection conn = null;
    private PreparedStatement ps = null;
    private EmployeePK pk;
    private static final String dsName = "jdbc/OracleDS";

    private static final String insertStatement =
        "INSERT INTO EMP (EMPNO, ENAME, SAL) VALUES (?, ?, ?)";
    private static final String updateStatement =
        "UPDATE EMP SET ENAME=?, SAL=? WHERE EMPNO=?";
    private static final String deleteStatement =
        "DELETE FROM EMP WHERE EMPNO=?";
    private static final String findAllStatement =
        "SELECT EMPNO, ENAME, SAL FROM EMP";
    private static final String findByPKStatement =
        "SELECT EMPNO, ENAME, SAL FROM EMP WHERE EMPNO = ?";
    private static final String findByNameStatement =
        "SELECT EMPNO, ENAME, SAL FROM EMP WHERE ENAME = ?";
    // or you can define a variable specific to orion to implement finder-method:
    // or use <finder-method/> in orion-ejb-jar.xml
```

```
public static final String findByNameQuery="full: " +
    "SELECT EMPNO, ENAME, SAL FROM EMP WHERE ENAME = $1";

public EmployeeBean() {
    // Empty constructor, don't initialize here but in the create().
    // passivate() may destroy these attributes in the case of pooling
}

public EmployeePK ejbCreate(Integer empNo, String empName, Float salary)
    throws CreateException {
    try {
        pk = new EmployeePK(empNo, empName, salary);
        conn = getConnection(dsName);
        ps = conn.prepareStatement(insertStatement);
        ps.setInt(1, empNo.intValue());
        ps.setString(2, empName);
        ps.setFloat(3, salary.floatValue());
        ps.executeUpdate();
        return pk;
    }
    catch (SQLException e) {
        System.out.println("Caught an exception 1 " + e.getMessage() );
        throw new CreateException(e.getMessage());
    }
    catch (NamingException e) {
        System.out.println("Caught an exception 1 " + e.getMessage() );
        throw new EJBException(e.getMessage());
    }
    finally {
        try {
            ps.close();
            conn.close();
        } catch (SQLException e) {
            throw new EJBException(e.getMessage());
        }
    }
}

public void ejbPostCreate(Integer empNo, String empName, Float salary)
    throws CreateException {
}

public EmployeePK ejbFindByPrimaryKey(EmployeePK pk)
    throws FinderException {
    if (pk == null || pk.empNo == null) {
        throw new FinderException("Primary key cannot be null");
    }
    try {
        conn = getConnection(dsName);
        ps = conn.prepareStatement(findByPKStatement);
        ps.setInt(1, pk.empNo.intValue());
        ps.executeQuery();
        ResultSet rs = ps.getResultSet();
        if (rs.next()) {
            pk.empNo = new Integer(rs.getInt(1));
            pk.empName = new String(rs.getString(2));
            pk.salary = new Float(rs.getFloat(3));
        }
        else {
            throw new FinderException("Failed to select this PK");
        }
    }
    catch (SQLException e) {
        throw new FinderException(e.getMessage());
    }
    catch (NamingException e) {
```

```
            System.out.println("Caught an exception 1 " + e.getMessage() );
            throw new EJBException(e.getMessage());
        }
        finally {
            try {
                ps.close();
                conn.close();
            }
        catch (SQLException e) {
                throw new EJBException(e.getMessage());
            }
        }
        return pk;
    }

    public Collection ejbFindAll() throws FinderException {
        //System.out.println("EmployeeBean.ejbFindAll(): begin");
        Vector recs = new Vector();
        try {
            conn = getConnection(dsName);
            ps = conn.prepareStatement(findAllStatement);
            ps.executeQuery();
            ResultSet rs = ps.getResultSet();
            int i = 0;
            while (rs.next()) {
                pk = new EmployeePK();
                pk.empNo = new Integer(rs.getInt(1));
                pk.empName = new String(rs.getString(2));
                pk.salary = new Float(rs.getFloat(3));
                recs.add(pk);
            }
        }
        catch (SQLException e) {
            throw new FinderException(e.getMessage());
        }
        catch (NamingException e) {
            System.out.println("Caught an exception 1 " + e.getMessage() );
            throw new EJBException(e.getMessage());
        }
        finally {
            try {
                ps.close();
                conn.close();
            }
            catch (SQLException e) {
                throw new EJBException(e.getMessage());
            }
        }
        return recs;
    }

    public Collection ejbFindByName(String empName)
        throws FinderException {
        //System.out.println("EmployeeBean.ejbFindByName(): begin");
        if (empName == null) {
            throw new FinderException("Name cannot be null");
        }
        Vector recs = new Vector();
        try {
            conn = getConnection(dsName);
            ps = conn.prepareStatement(findByNameStatement);
            ps.setString(1, empName);
            ps.executeQuery();
            ResultSet rs = ps.getResultSet();
            int i = 0;
            while (rs.next()) {
```

```
            pk = new EmployeePK();
            pk.empNo = new Integer(rs.getInt(1));
            pk.empName = new String(rs.getString(2));
            pk.salary = new Float(rs.getFloat(3));
            recs.add(pk);
        }
    }
    catch (SQLException e) {
        throw new FinderException(e.getMessage());
    }
    catch (NamingException e) {
        System.out.println("Caught an exception 1 " + e.getMessage() );
        throw new EJBException(e.getMessage());
    }
    finally {
        try {
            ps.close();
            conn.close();
        }
        catch (SQLException e) {
            throw new EJBException(e.getMessage());
        }
    }
    return recs;
}

public void ejbLoad() throws EJBException {
    //Container invokes this method to instruct the instance to
    //synchronize its state by loading it from the underlying database
    //System.out.println("EmployeeBean.ejbLoad(): begin");
    try {
        pk = (EmployeePK) ctx.getPrimaryKey();
        ejbFindByPrimaryKey(pk);
    }
    catch (FinderException e) {
        throw new EJBException (e.getMessage());
    }
}

public void ejbStore() throws EJBException {
    //Container invokes this method to instruct the instance to
    //synchronize its state by storing it to the underlying database
    //System.out.println("EmployeeBean.ejbStore(): begin");
    try {
        pk = (EmployeePK) ctx.getPrimaryKey();
        conn = getConnection(dsName);
        ps = conn.prepareStatement(updateStatement);
        ps.setString(1, pk.empName);
        ps.setFloat(2, pk.salary.floatValue());
        ps.setInt(3, pk.empNo.intValue());
        if (ps.executeUpdate() != 1) {
            throw new EJBException("Failed to update record");
        }
    }
    catch (SQLException e) {
        throw new EJBException(e.getMessage());
    }
    catch (NamingException e) {
        System.out.println("Caught an exception 1 " + e.getMessage() );
        throw new EJBException(e.getMessage());
    }
    finally {
        try {
            ps.close();
            conn.close();
        }
```

```
        catch (SQLException e) {
            throw new EJBException(e.getMessage());
        }
    }
}

public void ejbRemove() throws RemoveException {
    //Container invokes this method befor it removes the EJB object
    //that is currently associated with the instance
    //System.out.println("EmployeeBean.ejbRemove(): begin");
    try {
        pk = (EmployeePK) ctx.getPrimaryKey();
        conn = getConnection(dsName);
        ps = conn.prepareStatement(deleteStatement);
        ps.setInt(1, pk.empNo.intValue());
        if (ps.executeUpdate() != 1) {
            throw new RemoveException("Failed to delete record");
        }
    }
    catch (SQLException e) {
        throw new RemoveException(e.getMessage());
    }
    catch (NamingException e) {
        System.out.println("Caught an exception 1 " + e.getMessage() );
        throw new EJBException(e.getMessage());
    }
    finally {
        try {
            ps.close();
            conn.close();
        }
        catch (SQLException e) {
            throw new EJBException(e.getMessage());
        }
    }
}

public void ejbActivate() {
    // Container invokes this method when the instance is taken out
    // of the pool of available instances to become associated with
    // a specific EJB object
    //System.out.println("EmployeeBean.ejbActivate(): begin");
}

public void ejbPassivate() {
    // Container invokes this method on an instance before the instance
    // becomes disassociated with a specific EJB object
    //System.out.println("EmployeeBean.ejbPassivate(): begin");
}

public void setEntityContext(EntityContext ctx) {
    //Set the associated entity context
    //System.out.println("EmployeeBean.setEntityContext(): begin");
    this.ctx = ctx;
}

public void unsetEntityContext() {
    //Unset the associated entity context
    //System.out.println("EmployeeBean.unsetEntityContext(): begin");
    this.ctx = null;
}

/**
 * methods inherited from EJBObject
 */
public Integer getEmpNo() {
```

```
        pk = (EmployeePK) ctx.getPrimaryKey();
        return pk.empNo;
    }

    public String getEmpName() {
        pk = (EmployeePK) ctx.getPrimaryKey();
        return pk.empName;
    }

    public Float getSalary() {
        pk = (EmployeePK) ctx.getPrimaryKey();
        return pk.salary;
    }

    public void setEmpNo(Integer empNo) {
        pk = (EmployeePK) ctx.getPrimaryKey();
        pk.empNo = empNo;
    }

    public void setEmpName(String empName) {
        pk = (EmployeePK) ctx.getPrimaryKey();
        pk.empName = empName;
    }

    public void setSalary(Float salary) {
        pk = (EmployeePK) ctx.getPrimaryKey();
        pk.salary = salary;
    }

    public EJBHome getEJBHome() {
        return ctx.getEJBHome();
    }

    public Handle getHandle() throws RemoteException {
        return ctx.getEJBObject().getHandle();
    }

    public Object getPrimaryKey() throws RemoteException {
        return ctx.getEJBObject().getPrimaryKey();
    }

    public boolean isIdentical(EJBObject remote) throws RemoteException {
        return ctx.getEJBObject().isIdentical(remote);
    }

    public void remove() throws RemoveException, RemoteException{
        ctx.getEJBObject().remove();
    }

    /**
     * Private methods
     */
    private Connection getConnection(String dsName)
              throws SQLException, NamingException {
        DataSource ds = getDataSource(dsName);
        return ds.getConnection();
    }

    private DataSource getDataSource(String dsName) throws NamingException {
        DataSource ds = null;
        Context ic = new InitialContext();
        ds = (DataSource) ic.lookup(dsName);
        return ds;
    }
}
```

Example 13–6 EJB 2.1 BMP Remote Home Interface

```
package bmpapp;

import java.rmi.*;
import java.util.*;
import javax.ejb.*;

public interface EmployeeHome extends EJBHome {

    public Employee create(Integer empNo, String empName, Float salary)
        throws CreateException, RemoteException;

    public Employee findByPrimaryKey(EmployeePK pk)
        throws FinderException, RemoteException;

    public Collection findByName(String empName)
        throws FinderException, RemoteException;

    public Collection findAll()
        throws FinderException, RemoteException;
}
```

Example 13–7 EJB 2.1 BMP Remote Component Interface

```
package bmpapp;

import java.rmi.*;
import javax.ejb.*;

public interface Employee extends EJBObject {

    // getter remote methods
    public Integer getEmpNo() throws RemoteException;
    public String getEmpName() throws RemoteException;
    public Float getSalary() throws RemoteException;

    // setter remote methods
    public void setEmpNo(Integer empNo) throws RemoteException;
    public void setEmpName(String empName) throws RemoteException;
    public void setSalary(Float salary) throws RemoteException;
}
```

Using Deployment XML

Example 13–8 shows the `ejb-jar.xml` entity element corresponding to the entity bean with bean-managed persistence, shown in Example 13–5.

Example 13–8 ejb-jar.xml For an EJB 2.1 Entity Bean With Bean-Managed Persistence

```
...
    <enterprise-beans>
        <entity>
            <description>no description</description>
            <display-name>EmployeeBean</display-name>
            <ejb-name>EmployeeBean</ejb-name>
            <home>bmpapp.EmployeeHome</home>
            <remote>bmpapp.Employee</remote>
            <ejb-class>bmpapp.EmployeeBean</ejb-class>
            <persistence-type>Bean</persistence-type>
            <prim-key-class>bmpapp.EmployeePK</prim-key-class>
            <reentrant>False</reentrant>
            <resource-ref>
                <res-ref-name>jdbc/OracleDS</res-ref-name>
```

```
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Application</res-auth>
        </resource-ref>
    </entity>
</enterprise-beans>
...
```

Example 13–9 shows the `data-sources.xml` file `data-source` element `ejb-location` attribute that specifies the `res-ref-name` (`jdbc/OracleDS`) used in the `ejb-jar.xml` file shown in Example 13–8.

Example 13–9 data-sources.xml For an EJB 2.1 Entity Bean With Bean-Managed Persistence Data Source

```
<connection-pool name="Example Connection Pool">
    <!-- This is an example of a connection factory that emulates XA behavior. -->
    <connection-factory factory-class="oracle.jdbc.pool.OracleDataSource"
        user="scott"
        password="tiger"
        url="jdbc:oracle:thin:@//localhost:1521/oracle.regress.rdbms.dev.us.oracle.com">
    </connection-factory>
</connection-pool>

<managed-data-source name="OracleDS"
    connection-pool-name="Example Connection Pool"
    jndi-name="jdbc/OracleDS"/>
```

Implementing an ejbCreate Method for an EJB 2.1 Entity Bean With Bean-Managed Persistence

The `ejbCreate` method is responsible primarily for the creation of the primary key. This includes the following:

1. Creating the primary key.
2. Creating the persistent data representation for the key.
3. Initializing the key to a unique value and ensuring no duplication.
4. Returning this key to the container.

The container maps the key to the entity bean reference.

The following example shows the `ejbCreate` method for the employee example, which initializes the primary key, `empNo`. It should automatically generate a primary key that is the next available number in the employee number sequence. However, for this example to be simple, the `ejbCreate` method requires that the user provide the unique employee number.

> **Note:** For simplicity, the `try` blocks within the samples have been removed in this example.

In addition, because the full data for the employee is provided within this method, the data is saved within the context variables of this instance. After initialization, it returns this key to the container.

```
// The create methods takes care of generating a new empNo and returns
// its primary key to the container
public Integer ejbCreate (Integer empNo, String empName, Float salary)
   throws CreateException {
```

```
  // in this implementation, the client gives the employee number,
  // so only need to assign it, not create it
  this.empNo = empNo;
  this.empName = empName;
  this.salary = salary;

  // insert employee into database
  conn = getConnection(dsName);
  ps = conn.prepareStatement("INSERT INTO EMPLOYEEBEAN (EmpNo, EmpName, SAL)
       VALUES ( "+this.empNo.intValue()+", "+this.empName+","
       + this.salary.floatValue()+")");
  ps.executeUpdate();
  ps.close();

  // return the new primary key
  return (empNo);
}
```

The deployment descriptor defines only the primary key class in the `<prim-key-class>` element. Because the bean is saving the data, there is no definition of persistence data in the deployment descriptor. Note that the deployment descriptor does define the database the bean uses in the `<resource-ref>` element. For more information on database configuration, see "Using Deployment XML" on page 13-14.

```
<enterprise-beans>
   <entity>
      <display-name>EmployeeBean</display-name>
      <ejb-name>EmployeeBean</ejb-name>
      <local-home>employee.EmployeeLocalHome</local-home>
      <local>employee.EmployeeLocal</local>
      <ejb-class>employee.EmployeeBean</ejb-class>
      <persistence-type>Bean</persistence-type>
      <prim-key-class>java.lang.Integer</prim-key-class>
      <reentrant>False</reentrant>
      <resource-ref>
            <res-ref-name>jdbc/OracleDS</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Application</res-auth>
      </resource-ref>
   </entity>
</enterprise-beans>
```

Alternatively, you can create a complex primary key based on several data types. You define a complex primary key within its own class as follows:

```
package employee;

import java.io.*;
java.io.Serializable;

...

public class EmployeePK implements java.io.Serializable {
  public Integer empNo;
  public String empName;
  public Float salary;

  public EmployeePK(Integer empNo) {
    this.empNo = empNo;
```

```
    this.empName = null;
    this.salary = null;
  }

  public EmployeePK(Integer empNo, String empName, Float salary) {
    this.empNo = empNo;
    this.empName = empName;
    this.salary = salary;
  }

}
```

For a primary key class, you define the class in the `<prim-key-class>` element, which is the same for the simple primary key definition.

```
<enterprise-beans>
   <entity>
      <display-name>EmployeeBean</display-name>
      <ejb-name>EmployeeBean</ejb-name>
      <local-home>employee.EmployeeLocalHome</local-home>
      <local>employee.EmployeeLocal</local>
      <ejb-class>employee.EmployeeBean</ejb-class>
      <persistence-type>Bean</persistence-type>
      <prim-key-class>employee.EmployeePK</prim-key-class>
      <reentrant>False</reentrant>
      <resource-ref>
            <res-ref-name>jdbc/OracleDS</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Application</res-auth>
      </resource-ref>
   </entity>
</enterprise-beans>
```

The employee example requires that the employee number is given to the bean by the user. Another method would generate the employee number by computing the next available employee number, and use this in combination with the employee's name and office location.

After defining the complex primary key class, you would create your primary key within the `ejbCreate` method as follows:

```
public EmployeePK ejbCreate(Integer empNo, String empName, Float salary)
    throws CreateException {
  pk = new EmployeePK(empNo, empName, salary);
  ...
}
```

The other task that the `ejbCreate` (or `ejbPostCreate`) should handle is allocating any resources necessary for the life of the bean. For this example, because there is already the information for the employee, the `ejbCreate` performs the following:

1. Retrieves a connection to the database. This connection remains open for the life of the bean. It is used to update employee information within the database. It should be released in `ejbPassivate` and `ejbRemove`, and reallocated in `ejbActivate`.
2. Updates the database with the employee information.

This is executed as follows:

```
public EmployeePK ejbCreate(Integer empNo, String empName, Float salary)
    throws CreateException {
```

```
    pk = new EmployeePK(empNo, empName, salary);
    conn = getConnection(dsName);
    ps = conn.prepareStatement("INSERT INTO EMPLOYEEBEAN (EmpNo, EmpName, SAL)
        VALUES ( "+this.empNo.intValue()+", "+this.empName+","
        + this.salary.floatValue()+")");
    ps.executeUpdate();
    ps.close();
    return pk;
}
```

Implementing the EJB 2.1 Home Interfaces

The home interfaces are used to specify what methods a client uses to create or retrieve an entity bean instance.

The home interface must contain a `create` method, which the client invokes to create the bean instance. The entity bean can have zero or more `create` methods, each with its own defined parameters. For each `create` method, you define a corresponding `ejbCreate` method in the bean implementation.

All entity beans must define one or more finder methods in the home interface, where at least one is a `findByPrimaryKey` method. Optionally, you can define other finder methods, which are named `find<`*name*`>`, including predefined and default finders. For more information, see "Understanding Finder Methods" on page 1-53.

In addition to creation and retrieval methods, you can provide home interface business methods within the home interface. The functionality within these methods cannot access data of a particular entity object. Instead, the purpose of these methods is to provide a way to retrieve information that is not related to a single entity bean instance. When the client invokes any home interface business method, an entity bean is removed from the pool to service the request. Thus, this method can be used to perform operations on general information related to the bean.

For example, in an employee application, you might provide the local home interface with a `create`, `findByPrimaryKey`, `findAll`, and `calcSalary` methods. The `calcSalary` method is a home interface business method that calculates the sum of all employee salaries. It does not access the information of a particular employee, but performs a SQL query against the database for all employees.

There are the following two types of home interface:

- The remote home interface extends `javax.ejb.EJBHome` (see "Implementing the Remote Home Interface" on page 13-18)
- The local home interface extends `javax.ejb.EJBLocalHome` (see "Implementing the Local Home Interface" on page 13-19)

Implementing the Remote Home Interface

A remote client invokes the EJB through its remote interface. The client invokes the `create` method that is declared within the remote home interface. The container passes the client call to the `ejbCreate` method–with the appropriate parameter signature–within the bean implementation. The requirements for developing the remote home interface include the following:

- The remote home interface must extend the `javax.ejb.EJBHome` interface.
- All `create` methods may throw the following exceptions:
  - `javax.ejb.CreateException`

  - `javax.ejb.EJBException` or another `RuntimeException`

Example 13–2 shows the remote home interface corresponding to the EJB 2.1 entity bean with container-managed persistence in Example 13–1 and Example 13–6 shows the remote home interface corresponding to the EJB 2.1 entity bean with bean-managed persistence in Example 13–5.

Implementing the Local Home Interface

An EJB can be called locally from a client that exists in the same container. Thus, a collocated bean, JSP, or servlet invokes the `create` method that is declared within the local home interface. The container passes the client call to the `ejbCreate` method–with the appropriate parameter signature–within the bean implementation. The requirements for developing the local home interface include the following:

- The local home interface must extend the `javax.ejb.EJBLocalHome` interface.
- All `create` methods may throw the following exceptions:
  - `javax.ejb.CreateException`
  - `javax.ejb.EJBException` or another `RuntimeException`

Implementing the EJB 2.1 Component Interfaces

The component interfaces define the business methods of the bean that a client can invoke.

The entity bean component interface is the interface that the client can invoke its methods with. The component interface defines the business logic methods for the entity bean instance.

There are the following two types of component interface:

- The remote component interface extends `javax.ejb.EJBObject` (see "Implementing the Remote Component Interface" on page 13-19)
- The local component interface extends `javax.ejb.EJBLocalObject` (see "Implementing the Local Component Interface" on page 13-20)

Implementing the Remote Component Interface

The remote interface defines the business methods that a remote client can invoke. The requirements for developing the remote component interface include:

- The remote component interface of the bean must extend the `javax.ejb.EJBObject` interface, and its methods must throw the `java.rmi.RemoteException` exception.
- You must declare the remote interface and its methods as `public` for remote clients.
- The remote component interface, all its method parameters, and return types must be serializable. In general, any object that is passed between the client and the enterprise bean must be serializable, because RMI marshals and unmarshalls the object on both ends.
- Any exception can be thrown to the client. Run-time exceptions, including `EJBException` and `RemoteException`, are transferred back to the client as remote run-time exceptions.
- A remote component interface can throw a specified application exceptions.

Example 13–3 shows the remote component interface corresponding to the EJB 2.1 entity bean with container-managed persistence in Example 13–1 and Example 13–7 shows the remote component interface corresponding to the EJB 2.1 entity bean with bean-managed persistence in Example 13–5.

Implementing the Local Component Interface

The local component interface defines the business methods of the bean that a local (collocated) client can invoke. The requirements for developing the local component interface include the following:

- The local component interface of the bean must extend the `javax.ejb.EJBLocalObject` interface.
- You declare the local component interface and its methods as `public`.

Implementing the setEntityContext and unsetEntityContext Methods

An entity bean instance uses this method to retain a reference to its context. Entity beans have contexts that the container maintains and makes available to the beans. The bean may use the methods in the entity context to retrieve information about the bean, such as security and transactional role. Refer to the EJB specification from Sun Microsystems for the full range of information that you can retrieve about the bean from the context.

The container invokes the `setEntityContext` method after it first instantiates the bean to enable the bean to retrieve the context. The container will never call this method from within a transaction context. If the bean does not save the context at this point, the bean will never gain access to the context.

Note: You can also use the `setEntityContext` and `unsetEntityContext` methods to allocate and destroy any resources that will exist for the life time of the instance.

When the container calls this method, it passes the reference of the `EntityContext` object to the bean. The bean can then store the reference for later use. The following example shows the bean saving the context in the **`this.ctx`** variable.

You use this method to obtain a reference to the context of the bean. Entity beans have entity contexts that the container maintains and makes available to the beans. The bean may use the methods in the entity context to make callback requests to the container.

Example 13–10 shows an entity bean saving the session context in the `entityctx` variable.

Example 13–10 Implementing the setEntityContext and unsetEntityContext Methods

```
import javax.ejb.*;

public class MyBean implements EnityBean {
   EntityContext entityctx;

   public void setEntityContext(EntityContext ctx) {
      entityctx = ctx;   // entity context is stored in instance variable
   }
```

```
    public void unsetEntityContext() {
        entityctx = null;
    }

   // other methods in the bean
}
```

14

Using an EJB 2.1 Entity Bean With Container-Managed Persistence

This chapter describes the various options that you must configure in order to use an EJB 2.1 entity bean with container-managed persistence.

Table 14–1 lists these options and indicates which are basic (applicable to most applications) and which are advanced (applicable to more specialized applications).

For more information, see the following:

- "What is an EJB 2.1 Entity Bean With Container-Managed Persistence?" on page 1-42
- "Implementing an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 13-1

Table 14–1 Configurable Options for an EJB 2.1 CMP Entity Bean

Options	Type
"Configuring a Primary Key for an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 14-2	Basic
"Configuring Table and Column Information" on page 14-4	Advanced
"Configuring Automatic Database Table Creation" on page 14-5	Advanced
"Configuring Default Relationship Generation" on page 14-6	Advanced
"Configuring a Container-Managed Persistent Field for an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 14-7	Basic
"Configuring a Container-Managed Relationship Field for an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 14-9	Basic
"Configuring a One-to-One Relationship" on page 14-11	Basic
"Configuring a Many-to-One Relationship" on page 14-12	Basic
"Configuring a One-to-Many Relationship" on page 14-11	Basic
"Configuring a Many-to-Many Relationship" on page 14-13	Basic
"Configuring Lazy Loading on Finder Methods" on page 14-14	Advanced
"Configuring Bean Instance Pool Size" on page 31-4	Basic
"Configuring Bean Instance Pool Timeouts for Entity Beans" on page 31-7	Advanced
"Configuring a Life Cycle Callback Method for an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 14-15	Basic

Configuring a Primary Key for an EJB 2.1 Entity Bean With Container-Managed Persistence

Every EJB 2.1 entity bean with container-managed persistence must have a primary key field.

You can configure the primary key as a well-known Java type (see "Configuring a Primary Key Field for an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 14-2) or as a special type that you create (see "Configuring a Composite Primary Key Class for an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 14-3).

For more information, see "What is a Primary Key of an Entity Bean With Container-Managed Persistence?" on page 1-45

Typically, you rely on OC4J to assign primary key values automatically. To configure how OC4J assigns primary key values, you use TopLink persistence API. For more information, see the following:

- "Customizing the TopLink EJB 2.1 Persistence Manager" on page 3-13
- "Understanding Sequencing in Relational Projects" in the *Oracle TopLink Developer's Guide*

Configuring a Primary Key Field for an EJB 2.1 Entity Bean With Container-Managed Persistence

For a simple EJB 2.1 entity bean with container-managed persistence, you can define your primary key to be a well-known Java type as follows:

- Code your bean's `ejbCreate` method to return the primary key class type (see "Implementing an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 13-1)
- Configure your deployment XML to use it (see "Using Deployment XML" on page 14-2)

Once defined, the container may create a column or columns in the entity bean table for the primary key and maps the primary key defined in the deployment descriptor to this column. The container manages the instantiation of primary keys of this type and initializes your entity bean primary key field accordingly.

Using Deployment XML

Example 14–1 shows the `ejb-jar.xml` file `entity` element attributes `prim-key-class` and `primkey-field` configured to specify a primary key as well-known Java type `Integer`.

Example 14–1 ejb-jar.xml for Primary Key Field With Type Integer for EJB 2.1 Entity Bean With Container-Managed Persistence

```
<enterprise-beans>
  <entity>
    <display-name>Employee</display-name>
    <ejb-name>EmployeeBean</ejb-name>
    <local-home>employee.EmployeeLocalHome</local-home>
    <local>employee.EmployeeLocal</local>
    <ejb-class>employee.EmployeeBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.Integer</prim-key-class>
    <reentrant>False</reentrant>
```

```
    <cmp-version>2.x</cmp-version>
    <abstract-schema-name>Employee</abstract-schema-name>
    <cmp-field><field-name>empNo</field-name></cmp-field>
    <cmp-field><field-name>empName</field-name></cmp-field>
    <cmp-field><field-name>salary</field-name></cmp-field>
    <primkey-field>empNo</primkey-field>
    </entity>
...
</enterprise-beans>
```

Configuring a Composite Primary Key Class for an EJB 2.1 Entity Bean With Container-Managed Persistence

If your primary key is more complex than a well-known Java data type, then you can define your own primary key class.

Your primary key class must have the following characteristics:

- be named `<name>PK`
- be `public` and `serializable`
- provide a constructor for creating a primary key instance

Your class may contain any number of instance variables used to form the primary key. Instance variables must have the following characteristics:

- be `public`
- use data types that are either primitive or serializable, or types that can be mapped to SQL types

Once the primary key class is defined (see "Using Java" on page 14-3), to use it in an entity bean, you must do the following:

- Code your bean's `ejbCreate` method to return the primary key class type (see "Implementing an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 13-1)
- Configure your deployment XML to use it (see "Using Deployment XML" on page 14-4)

Once defined, the container may create a column or columns in the entity bean table for the primary key and maps the primary key defined in the deployment descriptor to this column. The container manages the instantiation of primary keys of this type and initializes your entity bean primary key field accordingly.

Using Java

Example 14–2 shows an example primary key class.

Example 14–2 Primary Key Class Implementation for an EJB 2.1 Entity Bean With Container-Managed Persistence

```
package employee;

import java.io.*;
import java.io.Serializable;
...

public class EmployeePK implements java.io.Serializable {
  public Integer empNo;
```

```
  public EmployeePK() {
    this.empNo = null;
  }

  public EmployeePK(Integer empNo) {
    this.empNo = empNo;
  }
}
```

Using Deployment XML

As Example 14–3 shows, you define the primary key class within the `ejb-jar.xml` file `<prim-key-class>` element. You define each primary key class instance variable in a `<cmp-field><field-name>` element using the same variable name as that used in the primary key class.

Example 14–3 ejb-jar.xml for Primary Key Class and Its Instance Variables for an EJB 2.1 Entity Bean With Container-Managed Persistence

```
<enterprise-beans>
  <entity>
    <description>no description</description>
    <display-name>EmployeeBean</display-name>
         <ejb-name>EmployeeBean</ejb-name>
         <local-home>employee.LocalEmployeeHome</home>
         <local>employee.LocalEmployee</remote>
         <ejb-class>employee.EmployeeBean</ejb-class>
         <persistence-type>Container</persistence-type>
        <prim-key-class>employee.EmployeePK</prim-key-class>
         <reentrant>False</reentrant>
         <cmp-version>2.x</cmp-version>
         <abstract-schema-name>Employee</abstract-schema-name>
        <cmp-field><field-name>empNo</field-name></cmp-field>
         <cmp-field><field-name>empName</field-name></cmp-field>
         <cmp-field><field-name>salary</field-name></cmp-field>
      </entity>
</enterprise-beans>
```

Once defined, the container may create a column or columns in the entity bean table for the primary key and maps the primary key class defined in the deployment descriptor to this column.

Configuring Table and Column Information

The EJB 2.1 specification does not prescribe how the abstract persistence schema of an entity bean should be mapped to a relational (or other) schema of a persistent store, or define how such a mapping is described.

However, using OC4J and the TopLink persistence API, you can do the following:

- Specify the table and column names of the database table associated with an entity bean with container-managed persistence.
- Specify how container-managed persistent fields and container-managed relationship fields should be mapped to your relational schema.
- Automatically create (and, optionally, delete) database tables for your persistent objects.

For more information, see the following:

- "Customizing the TopLink EJB 2.1 Persistence Manager" on page 3-13
- "Understanding Relational Mappings" in the *Oracle TopLink Developer's Guide*
- "Configuring Associated Tables" in the *Oracle TopLink Developer's Guide*
- "Configuring Automatic Database Table Creation" on page 14-5

> **Note:** In this release, `orion-ejb-jar.xml` file `<entity-deployment>` subelement `<cmp-field-mapping>` is not used. For more information, see "<entity-deployment>" on page A-10.

Configuring Automatic Database Table Creation

You can configure OC4J to automatically create (and, optionally, delete) database tables for your persistent objects (see "Using Deployment XML" on page 14-5).

You can use this feature in conjunction with default mappings (see "Configuring Default Relationship Generation" on page 14-6).

Using Deployment XML

You can configure automatic database table creation at one of three levels as Table 14–2 shows. You can override the system level configuration at the application level and you can override system and application configuration at the EJB module level.

Table 14–2 Configuring Automatic Table Generation

Level	Configuration File	Setting	Values
System (global)	`<OC4J_HOME>/config/application.xml`	`autocreate-tables` `autodelete-tables`	`True`[1] or `False` `True` or `False`[1]
Application (EAR)	`orion-application.xml`	`autocreate-tables` `autodelete-tables`	`True`[1] or `False` `True` or `False`[1]
EJB Module (JAR)	`orion-ejb-jar.xml`	`pm-properties` sub-element `default-mapping` attribute `db-table-gen`[2]	`Create`, `DropAndCreate`, or `UseExisting`[3]

1 Default.

2 For more information, see "Customizing the TopLink EJB 2.1 Persistence Manager" on page 3-13.

3 See Table 14–3.

If you configure automatic table generation at the EJB module level, the value you assign to the `db-table-gen` attribute corresponds to the `autocreate-tables` and `autodelete-tables` settings, as Table 14–3 shows.

Table 14–3 Equivalent Settings for db-table-gen

db-table-gen Setting	autocreate-tables Setting	autodelete-tables Setting
`Create`	`True`	`False`
`DropAndCreate`	`True`	`True`
`UseExisting`	`False`	NA

Configuring Default Relationship Generation

You can configure OC4J to automatically generate all required relationships at deployment time (see "Using Deployment XML" on page 14-6). To use this feature, you must do the following:

- Omit all container-managed relationship configuration (see "Configuring a Container-Managed Relationship Field for an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 14-9).
- Ensure that no `toplink-ejb-jar.xml` is present in the EJB module (see "What is the toplink-ejb-jar.xml File?" on page 2-6).

You can use this feature in conjunction with automatic database table creation (see "Configuring Automatic Database Table Creation" on page 14-5).

Using Deployment XML

To configure default relationship generation, configure the `orion-ejb-jar.xml` file element `pm-properties` subelement `default-mapping`, as Table 14–4 shows.

Table 14–4 orion-ejb-jar.xml File pm-properties Subentries for default-mapping

Entry	Description
`db-table-gen`	Optional element that determines what TopLink will do to prepare the database tables that are being mapped to. The following are valid values: ▪ `Create` (default): This value tells TopLink to create the mapped tables during the deployment. If the tables already exist, TopLink will log an appropriate warning messages (such as "*Table already existed...*") and keeps processing the deployment. ▪ `DropAndCreate`: This value tells TopLink to drop tables before creating them during deployment. If a table does not initially exist, the drop operation will cause an `SQLException` to be thrown through the driver. However, TopLink handles the exception (logs and ignores it) and moves on to process the table creation operation. The deployment fails only if both drop and create operations fail. ▪ `UseExisting`: This value tells TopLink to perform no table manipulation. If the tables do not exist, deployment still goes through without error. If no `orion-ejb-jar.xml` file is defined in your EAR file, the OC4J container generates one during deployment. In this case, to specify a value for `db-table-gen`, use the TopLink system property `toplink.defaultmapping.dbTableGenSetting`. For example: `-Dtoplink.defaultmapping.dbTableGenSetting="DropAndCreate"`. The `orion-ejb-jar.xml` property overrides the system property. If both the `orion-ejb-jar.xml` property and the system property are present, TopLink retrieves the setting from the `orion-ejb-jar.xml` file. This setting overrides `autocreate-tables` and `autodelete-tables` configuration at the application (EAR) or system level. For more information, see "Configuring Automatic Database Table Creation" on page 14-5.
`extended-table-names`	An element used if the generated table names are not long enough to be unique. Values are restricted to `true` or `false` (default). When set to `true`, the TopLink runtime will ensure that generated tables names are unique. In default mapping, each entity is mapped to one table. The only exception is in many-to-many mappings, where there is one extra relation table involved in the source and target entities. When `extended-table-names` is set to `false` (the default), a simple table naming algorithm is used as follows: table names are defined as `TL_<bean_name>`. For example, if the bean name is `Employee`, the associated table name would be `TL_EMPLOYEE`. However, if the same entity is defined in multiple JAR files in an application, or across multiple applications, table-naming collision is inevitable. To address this problem, set `extended-table-names` to `true`. When set to `true`, TopLink uses an alternative table-naming algorithm as follows: table names are defined as *<bean_name>_<jar_name>_<app_name>*. This algorithm uses the combination of bean, JAR, and EAR names to form a table name unique across the application. For example, given a bean named `Employee`, which is in `Test.jar`, which is in `Demo.ear` (and the application name is "`Demo`"), then the corresponding table name will be `EMPLOYEE_TEST_DEMO`. If there is no `orion-ejb-jar.xml` file defined in the EAR file, the OC4J container generates one during deployment. In this case, to specify a value for `extended-table-names`, use the TopLink system property `toplink.defaultmapping.useExtendedTableNames`. For example: `-Dtoplink.defaultmapping.useExtendedTableNames="true"`. The `orion-ejb-jar.xml` property overrides the system property. If both the `orion-ejb-jar.xml` property and the system property are present, TopLink retrieves the setting from the `orion-ejb-jar.xml` file.

Configuring a Container-Managed Persistent Field for an EJB 2.1 Entity Bean With Container-Managed Persistence

You do not define container-managed persistent fields in the entity bean class: container-managed persistent fields are virtual only. OC4J supplies the implementation of the container-managed persistent fields.

You must define `public`, `abstract` getter and setter methods for the container-managed persistent fields, using the EJB conventions (see "Using Java" on page 14-8). OC4J supplies the implementation of these methods. You must not expose these getter and setter methods in the remote interface of the entity bean.

You may assign only the following Java types to container-managed persistent fields: Java primitive types and Java serializable types. You may not assign an entity bean local interface type (or a collection of such) to a container-managed persistent field.

The container-managed persistent fields must be specified in the `ejb-jar.xml` deployment descriptor using the `cmp-field` element (see "Using Deployment XML" on page 14-8). The names of these fields must be valid Java identifiers and must begin with a lowercase letter, as determined by `java.lang.Character.isLowerCase`.

> **Note:** In this release, `orion-ejb-jar.xml` file `<entity-deployment>` subelement `<cmp-field-mapping>` is not used. For more information, see "<entity-deployment>" on page A-10.

The accessor methods must bear the name of the `cmp-field` that is specified in the deployment descriptor, and in which the first letter of the name of the `cmp-field` has been upper cased and prefixed by `get` or `set`.

For more information, see "What are Container-Managed Persistent Fields?" on page 1-42.

Using Java

Example 14–4 shows the abstract getter and setter methods for the container-managed persistent fields specified in the `ejb-jar.xml` file (see "Using Deployment XML" on page 14-8).

Example 14–4 EJB 2.1 Container-Managed Persistent Fields

```
package cmpapp;

import javax.ejb.*;
import java.rmi.*;

public abstract class EmployeeBean implements EntityBean {

    private EntityContext ctx;

    // container-managed persistent fields accessors
    public abstract Integer getEmpNo();
    public abstract void setEmpNo(Integer empNo);

    public abstract String getEmpName();
    public abstract void setEmpName(String empName);

    public abstract Float getSalary();
    public abstract void setSalary(Float salary);
...
}
```

Using Deployment XML

Example 14–5 shows the `cmp-field` elements for the getter and setter methods specified in the bean class (see "Using Java" on page 14-8).

Example 14–5 ejb-jar.xml for an EJB 2.1 Container-Managed Persistent Field

```
    <enterprise-beans>
        <entity>
            <ejb-name>Topic</ejb-name>
```

```
            <local-home>faqapp.TopicLocalHome</local-home>
            <local>faqapp.TopicLocal</local>
            <ejb-class>faqapp.TopicBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>java.lang.Integer</prim-key-class>
            <primkey-field>topicID</primkey-field>
            <reentrant>False</reentrant>
            <cmp-version>2.x</cmp-version>
            <abstract-schema-name>TopicBean</abstract-schema-name>
            <cmp-field>
                <field-name>topicID</field-name>
            </cmp-field>
            <cmp-field>
                <field-name>topicDesc</field-name>
            </cmp-field>
            ...
        </entity>
    </enterprise-beans>
```

Configuring a Container-Managed Relationship Field for an EJB 2.1 Entity Bean With Container-Managed Persistence

You do not define container-managed relationship fields in the entity bean class: container-managed relationship fields are virtual only. OC4J supplies the implementation of the container-managed relationship fields.

You must define `public`, `abstract` getter and setter methods for the container-managed relationship fields in the local interface of the related entity bean, using the EJB conventions (see "Using Java" on page 14-10). OC4J supplies the implementation of these methods. You must not expose these getter and setter methods in the remote interface of the entity bean.

You may assign only the following Java types to container-managed relationship fields: Java primitive types and Java serializable types. You may assign an entity bean local interface type (or a collection of such) to a container-managed relationship field.

You must specify container-managed relationship fields in the `ejb-jar.xml` deployment descriptor using the `cmr-field` element (see "Using Deployment XML" on page 14-10). The names of these fields must be valid Java identifiers and must begin with a lowercase letter, as determined by `java.lang.Character.isLowerCase`.

The accessor methods must bear the name of the container-managed relationship field (`cmr-field`) that is specified in the deployment descriptor, and in which the first letter of the name of the `cmr-field` has been upper cased and prefixed by `get` or `set`.

The accessor methods for container-managed relationship fields for one-to-many or many-to-many relationships must utilize one of the following collection interfaces: `java.util.Collection` or `java.util.Set`. The collection interfaces used in relationships are specified in the deployment descriptor. The implementation of the collection classes used for the container-managed relationship fields is supplied by the container. The collection classes that are used for container-managed relationships must not be exposed through the remote interface of the entity bean.

For more information, see the following:

- "What are Container-Managed Relationship Fields?" on page 1-42
- "Configuring Default Relationship Generation" on page 14-6
- "Configuring a One-to-One Relationship" on page 14-11

- "Configuring a One-to-Many Relationship" on page 14-11
- "Configuring a Many-to-One Relationship" on page 14-12
- "Configuring a Many-to-Many Relationship" on page 14-13

Using OC4J and the TopLink persistence API, you can configure how container-managed relationship fields are mapped to your relational schema. For more information, see the following:

- "Customizing the TopLink EJB 2.1 Persistence Manager" on page 3-13
- "Understanding Relational Mappings" in the *Oracle TopLink Developer's Guide*

Using Java

Example 14–6 shows the abstract getter and setter methods for the container-managed relationship fields specified in the `ejb-jar.xml` file (see "Using Deployment XML" on page 14-10).

Example 14–6 EJB 2.1 Container-Managed Relationship Fields

```
package cmpapp;

import javax.ejb.*;
import java.rmi.*;

public abstract class EmployeeBean implements EntityBean {

    private EntityContext ctx;

    // container-managed persistent fields accessors
    public abstract Integer getEmpNo();
    public abstract void setEmpNo(Integer empNo);

    public abstract String getEmpName();
    public abstract void setEmpName(String empName);

    public abstract Float getSalary();
    public abstract void setSalary(Float salary);

    public abstract void setProjects(Collection projects);
    public abstract Collection getProjects();
...
}
```

Using Deployment XML

Example 14–7 shows the `cmr-field` elements for the getter and setter methods specified in the bean class (see "Using Java" on page 14-10).

Example 14–7 ejb-jar.xml for an EJB 2.1 Container-Managed Relationship Field

```
...
    <relationships>
        <ejb-relation>
            <ejb-relation-name>Topic-Faqs</ejb-relation-name>
            <ejb-relationship-role>
                <ejb-relationship-role-name>Topic-has-Faqs</ejb-relationship-role-name>
                <multiplicity>Many</multiplicity>
                <relationship-role-source>
                    <ejb-name>TopicBean</ejb-name>
                </relationship-role-source>
```

```
                <cmr-field>
                    <cmr-field-name>faqs</cmr-field-name>
                    <cmr-field-type>java.util.Collection</cmr-field-type>
                </cmr-field>
            </ejb-relationship-role>
        <ejb-relation>
...
    <relationships>
```

Configuring a One-to-One Relationship

In a one-to-one relationship, an entity bean instance is related to a single instance of another entity bean.

You specify a container-managed one-to-one relationship in the `ejb-jar.xml` deployment descriptor (see "Using Deployment XML" on page 14-11).

For more information, see "Configuring a Container-Managed Relationship Field for an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 14-9.

Using Deployment XML

Example 14–8 shows the pair of `<ejb-relationship-role>` elements that define a one-to-one unidirectional relationship between `Order` and `ShippingAddress`. For a bidirectional relationship, you would add the appropriate `cmr-field` to the `<ejb-relationship-role>` for `ShippingAddress`.

Example 14–8 ejb-jar.xml for an EJB 2.1 Unidirectional One-to-One Relationship

```
...
<relationships>
    <ejb-relation>
        <ejb-relation-name>Order-ShippingAddress</ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>order-has-address</ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>OrderEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>shippingAddress</cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>address-for-order</ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>AddressEJB</ejb-name>
            </relationship-role-source>
        </ejb-relationship-role>
    </ejb-relation>
...
<relationships>
```

Configuring a One-to-Many Relationship

In a one-to-many relationship, an entity bean instance is related to multiple instances of another entity bean.

You specify a container-managed one-to-many relationship in the `ejb-jar.xml` deployment descriptor (see "Using Deployment XML" on page 14-12).

For more information, see "Configuring a Container-Managed Relationship Field for an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 14-9.

Using Deployment XML

Example 14–9 shows the pair of `<ejb-relationship-role>` elements that define a one-to-many bidirectional relationship between `Order` and `LineItem`. For a unidirectional relationship, you would omit the `cmr-field` from the appropriate `<ejb-relationship-role>` element.

Example 14–9 ejb-jar.xml for an EJB 2.1 Bidirectional One-to-Many Relationship

```
...
<relationships>
    <ejb-relation>
        <ejb-relation-name>Order-LineItem</ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>order-has-lineitems</ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>OrderEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>lineItems</cmr-field-name>
                <cmr-field-type>java.util.Collection</cmr-field-type>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>lineitem-belongsto-order</ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <cascade-delete/>
            <relationship-role-source>
                <ejb-name>LineItemEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>order</cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
    </ejb-relation>
...
<relationships>
```

Configuring a Many-to-One Relationship

In a many-to-one relationship, multiple instances of an entity bean may be related to a single instance of another entity bean. This multiplicity is the opposite of one-to-many.

You specify a container-managed many-to-one relationship in the `ejb-jar.xml` deployment descriptor (see "Using Deployment XML" on page 14-12).

For more information, see "Configuring a Container-Managed Relationship Field for an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 14-9.

Using Deployment XML

Example 14–10 shows the pair of `<ejb-relationship-role>` elements that define a many-to-one bidirectional relationship between `Employees` and `Department`. For a

unidirectional relationship, you would omit the `cmr-field` from the appropriate `<ejb-relationship-role>` element.

Example 14–10 ejb-jar.xml for an EJB 2.1 Bidirectional Many-to-One Relationship

```
...
<relationships>
    <ejb-relation>
        <ejb-relation-name>Employee-Department</ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>employees-belongto-dept</ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <relationship-role-source>
                <ejb-name>DepartmentEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>dept</cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>dept-has-employees</ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <cascade-delete/>
            <relationship-role-source>
                <ejb-name>LineItemEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>employees</cmr-field-name>
                <cmr-field-type>java.util.Collection</cmr-field-type>
            </cmr-field>
        </ejb-relationship-role>
    </ejb-relation>
...
<relationships>
```

Configuring a Many-to-Many Relationship

In a many-to-many relationship, entity bean instances may be related to multiple instances of each other.

You specify a container-managed many-to-many relationship in the `ejb-jar.xml` deployment descriptor (see "Using Deployment XML" on page 14-12).

For more information, see "Configuring a Container-Managed Relationship Field for an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 14-9.

Using Deployment XML

Example 14–11 shows the pair of `<ejb-relationship-role>` elements that define a many-to-many relationship between `Teams` and `Players`.

Example 14–11 ejb-jar.xml for an EJB 2.1 Many-to-Many Relationship

```
...
<relationships>
    <ejb-relation>
        <ejb-relationship-role>
            <ejb-relationship-role-name>team-has-players</ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <relationship-role-source>
                <ejb-name>TeamEJB</ejb-name>
            </relationship-role-source>
```

```
            <cmr-field>
                <cmr-field-name>players</cmr-field-name>
                <cmr-field-type>java.util.Collection</cmr-field-type>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>player-has-teams</ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <relationship-role-source>
                <ejb-name>PlayerEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>teams</cmr-field-name>
                <cmr-field-type>java.util.Collection</cmr-field-type>
            </cmr-field>
        </ejb-relationship-role>
    </ejb-relation>
...
<relationships>
```

Configuring Lazy Loading on Finder Methods

Each finder method retrieves one or more objects. In the default scenario (which is set to NO lazy loading), the finder method causes a single SQL select statement to be executed against the database. For an entity bean with container-managed persistence, one or more objects are retrieved with all of their container-managed persistent fields. So, for example, with the `findAllEmployees` method, this finder retrieves all employee objects with all of the container-managed persistent fields in each employee object.

If you turn on lazy loading, then only the primary keys of the objects retrieved within the finder are returned. Then, only when you access the object within your implementation, OC4J uploads the actual object based on the primary key. With the `findAllEmployees` finder method example, all of the employee primary keys are returned in a `Collection`. The first time you access one of the employees in the `Collection`, OC4J uses the primary key to retrieve the single employee object from the database. You may want to turn on the lazy loading feature if the number of objects that you are retrieving is so large that loading them all into your local cache would be a performance degradation.

You have a performance consideration with lazy loading. If you retrieve multiple objects, but you only use a few of them, then you should turn on lazy loading. In addition, if you only use objects through the `getPrimaryKey` method, then you should also turn on lazy loading.

Using Deployment XML

To turn on lazy loading in the `findByPrimaryKey` method, set the `findByPrimaryKey-lazy-loading` attribute to `true`, as follows:

```
<entity-deployment ... findByPrimaryKey-lazy-loading="true" ... >
```

To turn on lazy loading in any custom finder method, set the `lazy-loading` attribute to `true` in the `<finder-method>` element for that custom finder, as follows:

```
<finder-method ... lazy-loading="true" ...>
  ...
</finder-method>
```

Configuring a Life Cycle Callback Method for an EJB 2.1 Entity Bean With Container-Managed Persistence

The following are the EJB 2.1 life cycle methods, as specified in the `javax.ejb.EntityBean` interface, that an entity bean with container-managed persistence must implement (see "Using Java" on page 14-15):

- `ejbCreate`
- `ejbPostCreate`
- `ejbRemove`
- `ejbStore`
- `ejbLoad`
- `ejbActivate`
- `ejbPassivate`

> **Note:** Using EJB 2.1, you must implement all entity bean callback methods. If you do not need to take any action, implement an empty method.

For more information, see "What is the Life Cycle of an EJB 2.1 Entity Bean With Container-Managed Persistence?" on page 1-43.

Using Java

Example 14–12 shows how to implement an EBJ 2.1 entity bean life cycle callback method.

Example 14–12 EJB 2.1 Entity Bean Life Cycle Callback Method Implementation

```
public void ejbActivate() {
    // when bean is activated
}
```

15

Using an EJB 2.1 Entity Bean With Bean-Managed Persistence

This chapter describes the various options that you must configure in order to use an EJB 2.1 entity bean with bean-managed persistence.

Table 15–1 lists these options and indicates which are basic (applicable to most applications) and which are advanced (applicable to more specialized applications).

For more information, see the following:

- "What is an EJB 2.1 Entity Bean With Bean-Managed Persistence?" on page 1-46
- "Implementing an EJB 2.1 Entity Bean With Bean-Managed Persistence" on page 13-6

Table 15–1 Configurable Options for an EJB 2.1 Entity Bean With Bean-Managed Persistence

Options	Type
"Configuring a Primary Key for an EJB 2.1 Entity Bean With Bean-Managed Persistence" on page 15-1	Basic
"Configuring a Read-Only Entity Bean With Bean-Managed Persistence" on page 15-4	Advanced
"Configuring Commit Options for an Entity Bean With Bean-Managed Persistence" on page 15-5	Advanced
"Configuring a Query for an EJB 2.1 Entity Bean With Bean-Managed Persistence" on page 15-5	Basic
"Configuring a Life Cycle Callback Method for an EJB 2.1 Entity Bean With Bean-Managed Persistence" on page 15-7	Basic

Configuring a Primary Key for an EJB 2.1 Entity Bean With Bean-Managed Persistence

Every EJB 2.1 entity bean with bean-managed persistence must have a primary key field.

You can configure the primary key as a well-known Java type (see "Configuring a Primary Key Field for an EJB 2.1 Entity Bean With Bean-Managed Persistence" on page 15-2) or as a special type that you create (see "Configuring a Primary Key Class for an EJB 2.1 Entity Bean With Bean-Managed Persistence" on page 15-2).

For more information, see "What is a Primary Key of an Entity Bean With Bean-Managed Persistence?"

For EJB 2.1 entity beans with bean-managed persistence, you are responsible for assigning primary key values, typically in the `ejbCreate` method (see "Configuring a Life Cycle Callback Method for an EJB 2.1 Entity Bean With Bean-Managed Persistence" on page 15-7).

Configuring a Primary Key Field for an EJB 2.1 Entity Bean With Bean-Managed Persistence

For a simple EJB 2.1 entity bean with bean-managed persistence, you can define your primary key to be a well-known Java type as follows:

- Code your bean's `ejbCreate` method to return the primary key class type (see "Implementing an EJB 2.1 Entity Bean With Bean-Managed Persistence" on page 13-6)
- Configure your deployment XML to use it (see "Using Deployment XML" on page 15-2)

Using Deployment XML

Example 15–1 shows the `ejb-jar.xml` file `<entity>` element `<prim-key-class>` and `<primkey-field>` subelements configured to specify a primary key as well-known Java type `Integer`.

Example 15–1 ejb-jar.xml for Primary Key Field With Type Integer of EJB 2.1 Entity Bean With Bean-Managed Persistence

```
<enterprise-beans>
  <entity>
    <display-name>Employee</display-name>
    <ejb-name>EmployeeBean</ejb-name>
    <local-home>employee.EmployeeLocalHome</local-home>
    <local>employee.EmployeeLocal</local>
    <ejb-class>employee.EmployeeBean</ejb-class>
    <persistence-type>Bean</persistence-type>
    <prim-key-class>java.lang.Integer</prim-key-class>
    <reentrant>False</reentrant>
    <cmp-version>2.x</cmp-version>
    <abstract-schema-name>Employee</abstract-schema-name>
    <cmp-field><field-name>empNo</field-name></cmp-field>
    <cmp-field><field-name>empName</field-name></cmp-field>
    <cmp-field><field-name>salary</field-name></cmp-field>
    <primkey-field>empNo</primkey-field>
    </entity>
...
</enterprise-beans>
```

Configuring a Primary Key Class for an EJB 2.1 Entity Bean With Bean-Managed Persistence

If your primary key is more complex than a well-known Java data type, then you can define your own primary key class.

Your primary key class must have the following characteristics:

- be named `<`*name*`>PK`
- be `public` and `serializable`
- provide a constructor for creating a primary key instance

Your class may contain any number of instance variables used to form the primary key. Instance variables must have the following characteristics:

- primitive object types

- serializable types
- types that can be mapped to SQL types
- types that are a legal Value Type in RMI-IIOP
- types that provide a suitable implementation of the `hashCode()` and `equals(Object)` methods

Once the primary key class is defined (see "Using Java" on page 15-3), to use it in an EJB, you must do the following:

- Code your bean's `ejbCreate` method to return the primary key class type (see "Implementing an EJB 2.1 Entity Bean With Bean-Managed Persistence" on page 13-6)
- Configure your deployment XML to use it (see "Using Deployment XML" on page 15-3)

Using Java

Example 15–2 shows an example primary key class.

Example 15–2 Primary Key Class Implementation for a EJB 2.1 Entity Bean With Bean-Managed Persistence

```
package employee;

import java.io.*;
import java.io.Serializable;
...

public class EmployeePK implements java.io.Serializable {
  public Integer empNo;

  public EmployeePK() {
    this.empNo = null;
  }

  public EmployeePK(Integer empNo) {
    this.empNo = empNo;
  }
}
```

Using Deployment XML

As Example 15–3 shows, you define the primary key class within the `ejb-jar.xml` file `<prim-key-class>` element. You define each primary key class instance variable in a `<cmp-field><field-name>` element using the same variable name as that used in the primary key class.

Example 15–3 ejb-jar.xml for Primary Key Class and Its Instance Variables of EJB 2.1 Entity Bean With Bean-Managed Persistence

```
<enterprise-beans>
  <entity>
    <description>no description</description>
    <display-name>EmployeeBean</display-name>
         <ejb-name>EmployeeBean</ejb-name>
         <local-home>employee.LocalEmployeeHome</home>
         <local>employee.LocalEmployee</remote>
```

```
        <ejb-class>employee.EmployeeBean</ejb-class>
        <persistence-type>Bean</persistence-type>
        <prim-key-class>employee.EmployeePK</prim-key-class>
        <reentrant>False</reentrant>
        <cmp-version>2.x</cmp-version>
        <abstract-schema-name>Employee</abstract-schema-name>
        <cmp-field><field-name>empNo</field-name></cmp-field>
        <cmp-field><field-name>empName</field-name></cmp-field>
        <cmp-field><field-name>salary</field-name></cmp-field>
      </entity>
</enterprise-beans>
```

Configuring a Read-Only Entity Bean With Bean-Managed Persistence

You can configure an entity bean with container-managed persistence as read-only. By doing so, you enter into a contract with OC4J, by which you guarantee not to change the state of the entity bean with container-managed persistence after it is activated. Unlike read-only entity beans with container-managed persistence, no exception will be thrown if you do update a read-only bean with bean-managed persistence.

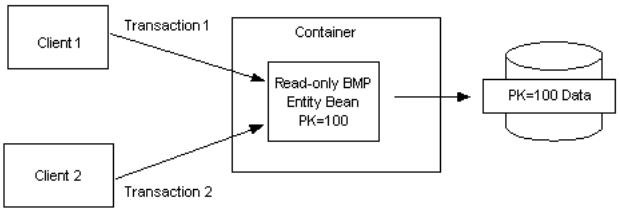
When you configure an entity bean with bean-managed persistence as read-only, OC4J uses a special case of commit option A (see "Configuring Commit Options for an Entity Bean With Bean-Managed Persistence" on page 15-5) to improve performance by the following:

- Caching the instance
- Not calling `ejbLoad` after activation
- Not updating the instance or calling `ejbStore` when the transaction commits

As Figure 15–1 shows, multiple clients accessing the same read-only entity bean with bean-managed persistence by primary key are allocated a single instance. Both Client 1 and Client 2 are satisfied by the same cached instance of a read-only entity bean with bean-managed persistence. Because the entity bean with bean-managed persistence is read-only, both transactions can proceed in parallel.

Without this optimization, each client is allocated a separate instance, and each instance requires the execution of all life cycle methods.

Figure 15–1 Read-Only Entity Beans With Bean-Managed Persistence and Commit Option A



Using Deployment XML

Example 15–4 shows the `orion-ejb-jar.xml` file `entity-deployment` element `locking-mode` attribute `mode` configured to specify an entity bean with bean-managed persistence as read-only.

Example 15–4 orion-ejb-jar.xml For Read-Only

```
<entity-deployment
    name=EmployeeBean"
    location="bmpapp/EmployeeBean"
    locking-mode="read-only"
>
...
</entity-deployment>
```

Configuring Commit Options for an Entity Bean With Bean-Managed Persistence

For an entity bean with bean-managed persistence, you can choose between commit options A and C.

Commit option A offers a performance improvement by postponing a call to `ejbLoad`.

If you configure a read-only entity bean with bean-managed persistence to use commit option A (see "Configuring a Read-Only Entity Bean With Bean-Managed Persistence" on page 15-4), you can further improve performance by taking advantage of the caching of the read-only entity bean with bean-managed persistence (see "Commit Options and BMP Applications" on page 1-50.

Commit option C is the default.

For more information, see "What are Entity Bean Commit Options?" on page 1-48.

Using Deployment XML

Example 15–5 shows the `orion-ejb-jar.xml` file `entity-deployment` element `commit-option` sub-element attribute `mode`. Valid settings are `A` and `C`. The `number-of-buckets` attribute is the maximum number of cached instances allowed and is applicable only for commit option `A`.

Example 15–5 orion-ejb-jar.xml For Commit Options

```
<entity-deployment name=EmployeeBean" location="bmpapp/EmployeeBean" >
   <resource-ref-mapping name="jdbc/OracleDS" />
   <commit-option mode="A" number-of-buckets="10" />
</entity-deployment>
```

Configuring a Query for an EJB 2.1 Entity Bean With Bean-Managed Persistence

You must implement an `ejbFindByPrimaryKey` method for an entity bean with bean-managed persistence (see "Implementing an ejbFindByPrimaryKey Method for an EJB 2.1 Entity Bean With Bean-Managed Persistence" on page 15-6). Optionally, you may configure other finders (see "Implementing Other Finder Methods for a EJB 2.1 Entity Bean With Bean-Managed Persistence" on page 15-6).

For more information, see "Implementing EJB 2.1 Queries" on page 16-1.

Implementing an ejbFindByPrimaryKey Method for an EJB 2.1 Entity Bean With Bean-Managed Persistence

The `ejbFindByPrimaryKey` implementation is a requirement for all entity beans with bean-managed persistence. Its primary responsibility is to ensure that the primary key corresponds to a valid bean. Once it is validated, it returns the primary key to the container, which uses the key to return the bean reference to the user.

This sample verifies that the employee number is valid and returns the primary key, which is the employee number, to the container. A more complex verification would be necessary if the primary key was a class.

```
public EmployeePK ejbFindByPrimaryKey(EmployeePK pk)
    throws FinderException {
    if (pk == null || pk.empNo == null) {
        throw new FinderException("Primary key cannot be null");
    }
    try {
        conn = getConnection(dsName);
        ps = conn.prepareStatement(findByPKStatement);
        ps.setInt(1, pk.empNo.intValue());
        ps.executeQuery();
        ResultSet rs = ps.getResultSet();
        if (rs.next()) {
            pk.empNo = new Integer(rs.getInt(1));
            pk.empName = new String(rs.getString(2));
            pk.salary = new Float(rs.getFloat(3));
        }
        else {
            throw new FinderException("Failed to select this PK");
        }
    }
    catch (SQLException e) {
        throw new FinderException(e.getMessage());
    }
    catch (NamingException e) {
        System.out.println("Caught an exception 1 " + e.getMessage() );
        throw new EJBException(e.getMessage());
    }
    finally {
        try {
            ps.close();
            conn.close();
        }
        catch (SQLException e) {
            throw new EJBException(e.getMessage());
        }
    }
    return pk;
}
```

Implementing Other Finder Methods for a EJB 2.1 Entity Bean With Bean-Managed Persistence

Optionally, you can create other finder methods in addition to the single `ejbFindByPrimaryKey` method.

To create other finder methods, do the following:

1. Add the finder method to the home interface.
2. Implement the finder method in the bean implementation of an entity bean with bean-managed persistence.

Finders can retrieve one or more beans according to the `WHERE` clause. If more than a single bean is returned, then a `Collection` of primary keys must be returned by the bean's finder method. These finder methods need only to gather the primary keys for all of the entity beans that should be returned to the user. The container maps the primary keys to references to each entity bean within either a `Collection` (if multiple references are returned) or to the single class type.

The following example shows the implementation of a finder method that returns all employee records.

```
public Collection ejbFindAll() throws FinderException {
  ArrayList recs = new ArrayList();

  ps = conn.prepareStatement("SELECT EMPNO FROM EMPLOYEEBEAN");
  ps.executeQuery();
  ResultSet rs = ps.getResultSet();

  int i = 0;

  while (rs.next()) {
   retEmpNo = new Integer(rs.getInt(1));
   recs.add(retEmpNo);
  }

  ps.close();
  return recs;
}
```

Configuring a Life Cycle Callback Method for an EJB 2.1 Entity Bean With Bean-Managed Persistence

The following are the EJB 2.1 life cycle methods, as specified in the `javax.ejb.EntityBean` interface, that an entity bean with bean-managed persistence must implement (see "Using Java" on page 15-7):

- `ejbCreate`
- `ejbPostCreate`
- `ejbRemove`
- `ejbStore`
- `ejbLoad`
- `ejbActivate`
- `ejbPassivate`

For an entity bean with bean-managed persistence, you are responsible for providing a complete implementation of all life cycle methods.

For more information, see "What is the Life Cycle of an EJB 2.1 Entity Bean With Bean-Managed Persistence?" on page 1-46.

Using Java

Example 15–6 shows how to implement an EBJ 2.1 entity bean life cycle callback method.

Example 15–6 EJB 2.1 Entity Bean Life Cycle Callback Method Implementation

```
public void ejbActivate() {
    // when bean is activated
}
```

16

Implementing EJB 2.1 Queries

This chapter describes the following:

- Implementing an EJB 2.1 EJB QL Finder Method
- Implementing an EJB 2.1 EJB QL Select Method
- OC4J EJB 2.1 EJB QL Extensions

For more information, see the following:

- "How do you Query for an EJB 2.1 Entity Bean?" on page 1-50
- "Implementing an EJB 2.1 Entity Bean" on page 13-1

> **Note:** For an example OC4J EJB QL application, see: `http://www.oracle.com/technology/sample_code/tech/java/ejb_corba/ejbql/Readme.html`.

Implementing an EJB 2.1 EJB QL Finder Method

The following procedure describes how to implement an EJB 2.1 EJB QL finder method.

Before implementing a finder method, consider the predefined and default finders that OC4J provides (see "Predefined TopLink Finders" on page 1-53 and "Default TopLink Finders" on page 1-54).

For more information, see "Understanding Finder Methods" on page 1-53.

1. Define the finder method in the home interface (see "Using Java" on page 16-2).

   If you are exposing only predefined or default finders (see "Predefined TopLink Finders" on page 1-53 and "Default TopLink Finders" on page 1-54), you are done.

   If you are exposing a custom finder, proceed to step 2.

2. Configure the `ejb-jar.xml` file (see "Using Deployment XML" on page 16-3).

   > **Note:** You can do this manually as described here or you can use the TopLink Workbench (see "Using TopLink Workbench" on page 16-4) to automate this step and to take advantage of advanced TopLink finder configuration.

   a. For each entity bean that you plan to reference in your EJB QL query, configure the `<entity>` element `<abstract-schema-name>` subelement.

The <abstract-schema-name> subelement defines the name that identifies the entity bean in the EJB QL statement. For example, given an entity bean class named EmpBean, if you define its <abstract-schema-name> as Employee, then in your EJB QL statement, when you use the name Employee, the container will map that name to the EmpBean entity bean (see Example 16–2).

b. Define a <query> element for each finder method that you exposed in the EJB home interface.

Note: Do not define a <query> element for predefined or default finders, including findByPrimaryKey.

The <query> element has the following subelements:

– <description>: optional explanatory text.

– <query-method>: describes the finder method and includes the following subelements:

  <method-name>: identifies the finder method. Configure this element with the same method name as defined in the home interface.

  <method-params>: if the finder takes arguments, define this element and for each argument, define a <method-param> subelement that gives the argument type. The type and order of arguments must match that specified by this finder's signature.

– <ejb-ql>: contains the EJB QL statement for this method.

  You can define a full query or just the conditional statement (the WHERE clause).

  If the finder method returns a Collection, to ensure that no duplicates are returned, specify the DISTINCT keyword in the EJB QL statement.

  To use parameters (as specified by <method-params>) in your EJB QL, use the <*integer*>? notation where <*integer*> begins with 1. For example,?1 corresponds to the first <method-param> element,?2 corresponds to the second <method-param> element, and so on (see the findAllByEmpName finder in Example 16–2).

  To define an EJB QL statement that relates this EJB with another, you must first define the appropriate container-managed relationship. The findByDeptNo finder in Example 16–2 requires the relationship with <ejb-relation-name> Employee-Departments. For more information, see "Configuring a Container-Managed Relationship Field for an EJB 2.1 Entity Bean With Container-Managed Persistence" on page 14-9.

Using Java

Example 16–1 shows a remote home interface called EmpBeanHome.

Example 16–1 Finder Methods in an EJB 2.1 Entity Bean With Container-Managed Persistence Remote Home Interface

```
package cmpapp;

import javax.ejb.*;
import java.rmi.*;
```

```
public interface EmpBeanHome extends EJBHome {
    public EmpBean create(Integer empNo, String empName) throws CreateException;

    /**
     * Finder methods. These are implemented by the container. You can
     * customize the functionality of these methods in the deployment
     * descriptor through EJB-QL.
     **/

// Predefined Finders: <query> element in ejb-jar.xml not required

    public Topic findByPrimaryKey(Integer key) throws FinderException;
    public Collection findManyBySQL(String sql, Vector args) throws FinderException

// Default Finder: <query> element in ejb-jar.xml not required

    public Topic findByEmpNo(Integer empNo) throws FinderException;

// Custom Finders: <query> element is required in ejb-jar.xml

    public Collection findAllRegionalEmployees(Integer empNo) throws FinderException;
    public Collection findAllByEmpName(String empName) throws FinderException;
    public Topic findByDeptNo(Integer deptNo) thorws FinderException
    public Collection findAllBetweenSalaries(Integer lowSalary, Integer highSalary);

}
```

Using Deployment XML

Example 16–2 shows the `ejb-jar.xml` for the finders declared in the home interface that Example 16–1 shows.

Example 16–2 ejb-jar.xml For EJB 2.1 EJB QL Finders

```
<enterprise-beans>
    <entity>
        <display-name>EmpBean</display-name>
        <ejb-name>EmpBean</ejb-name>
        ...
        <abstract-schema-name>Employee</abstract-schema-name>
        <cmp-field><field-name>empNo</field-name></cmp-field>
        <cmp-field><field-name>empName</field-name></cmp-field>
        <cmp-field><field-name>salary</field-name></cmp-field>
        <primkey-field>empNo</primkey-field>
        <prim-key-class>java.lang.Integer</prim-key-class>
        ...
        <query>
            <description>Regional employees have empNo greater than 10000</description>
            <query-method>
                    <method-name>findAllRegionalEmployees</method-name>
                    <method-params></method-params>
            </query-method>
            <ejb-ql>SELECT OBJECT(e) FROM Employee e WHERE e.empNo > 10000</ejb-ql>
        </query>
        <query>
            <description>Find all employees with the given name</description>
            <query-method>
                <method-name>findAllByEmpName</method-name>
                <method-params>
                    <method-param>java.lang.String</method-param>
                </method-params>
            </query-method>
            <ejb-ql>SELECT OBJECT(e) FROM Employee e WHERE e.empName = ?1</ejb-ql>
```

```
        </query>
        <query>
            <description>Relationship finder</description>
            <query-method>
                <method-name>findByDeptNo</method-name>
                <method-params>
                    <method-param>java.lang.Integer</method-param>
                </method-params>
            </query-method>
            <ejb-ql>
              SELECT DISTINCT OBJECT(e) From Employee e, IN (e.dept) AS d WHERE d.deptNo = ?1
           </ejb-ql>
        </query>
        <query>
            <description>Find all employees with salaries in the given range</description>
            <query-method>
                <method-name>findAllBetweenSalaries</method-name>
                <method-params>
                    <method-param>java.lang.Integer</method-param>
                    <method-param>java.lang.Integer</method-param>
                </method-params>
            </query-method>
            <ejb-ql>
                SELECT  OBJECT (e) FROM Employee e WHERE e.salary BETWEEN ?1 and ?2
            </ejb-ql>
        </query>
    ...
    </entity>
...
</enterprise-beans>
<relationships>
    <ejb-relation>
        <ejb-relation-name>Employee-Departments</ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>Employee-has-Departments</ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>Department</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>dept</cmr-field-name>
                <cmr-field-type>java.lang.Integer</cmr-field-type>
            </cmr-field>
        </ejb-relationship-role>
    <ejb-relation>
...
<relationships>
```

Using TopLink Workbench

Using the TopLink Workbench, you can configure your `toplink-ejb-jar.xml` file with a custom TopLink finder and update your `ejb-jar.xml` file.

For more information, see the following:

- "Creating a Finder" in the *Oracle TopLink Developer's Guide*
- "Configuring Named Queries at the Descriptor Level" in the *Oracle TopLink Developer's Guide*

Implementing an EJB 2.1 EJB QL Select Method

The following procedure describes how to implement an EJB 2.1 EJB QL select method.

For more information, see "Understanding Select Methods" on page 1-55.

1. Define the select method as a `public`, `abstract` method of your abstract entity bean class (see "Using Java" on page 16-5).
2. In the `ejb-jar.xml` file (see "Using Deployment XML" on page 16-7), do the following:
   a. For each entity bean that you plan to reference in your EJB QL query, configure the `<entity>` element `<abstract-schema-name>` subelement.

      The `<abstract-schema-name>` subelement defines the name that identifies the entity bean in the EJB QL statement. For example, given an entity bean class named `EmpBean`: if you define your `<abstract-schema-name>` as `Employee`, then in your EJB QL statement, when you use the name `Employee`, the container will map that name to the `EmpBean` entity bean.

   b. Define a `<query>` element for each select method that you exposed in the EJB home interface.

      You can define a full query or just the conditional statement (the `WHERE` clause).

      If the select method returns a `Collection`, to ensure that no duplicates are returned, specify the `DISTINCT` keyword in the EJB QL statement.

      The `<query>` element has the following two main elements:

      – The `<method-name>` element identifies the select method: configure this element with the same name as defined in the bean class.

      – The `<ejb-ql>` element contains the EJB QL statement for this method.

   c. If the query returns a `Collection` of CMR values, decide on the interface type you want returned:

      The `ejb-jar.xml` file `<result-type-mapping>` element determines the return type for select methods. Set the flag to `Remote` to return `EJBObjects`; set it to `Local` to return `EJBLocalObjects`.

Using Java

Example 16–3 shows an abstract entity bean class called `UserAccountBean` for an EJB 2.1 entity bean with container-managed persistence with select methods.

Example 16–3 Implementation of an EJB 2.1 Entity Bean With Container-Managed Persistence With Select Methods

```
package oracle.otnsamples.ejbql;

import javax.ejb.*;
import java.util.*;

public abstract class UserAccountBean implements EntityBean {

    // Non-Persistent State

    protected EntityContext ctx;
```

```
/**
 * Begin abstract get/set methods. Container-managed
 * persistent fields are specified in the ejb-jar.xml
 * deployment descriptor.
 */

public abstract Long getAccountnumber();
public abstract void setAccountnumber(Long newAccountnumber);

public abstract Long getCreditlimit();
public abstract void setCreditlimit(Long newCreditlimit);

/**
 * Select methods. These are implemented by the container. You can
 * customize the functionality of these methods in the deployment
 * descriptor through EJB-QL.
 *
 * These methods are NOT exposed in the bean's home interface.
 */

public abstract Long ejbSelectCreditLimit(Long accountnumber) throws FinderException;
public abstract Collection ejbSelectByTopAccounts() throws FinderException;

/**
 * Begin buisness logic methods that use select methods.
 *
 * These methods are exposed in the bean's home interfaces.
 */

/**
 * Method to perform post-processing operations on all the
 * UserAccounts retrieved by calling ejbSelectByTopAccounts. This
 * method further process the retrieved UserAccounts and checks
 * for the Accounts with TopCredits (credit limits) and returns the
 * collection of input number of UserAccounts.
 * Post-processing information within the EJB container itself
 * has the following two advantages:
 *     1) It improves performance as the application can now leverage
 *            the advantage of the vast resources available to the server.
 *     2) The data-processing code should go into the business logic
 *            and not the Web-tier. This helps in maintaining the code.
 * Consider these advantages when deciding between ejbFind and
 * ejbSelect methods.
 *
 * @return Collection of <input number of> Top (credited) UserAccounts
 */
public Collection ejbHomeTopAccounts(String accountNumbers) throws FinderException {
    // Invoke the ejbSelect method and get all the Account Information.
    Collection collection = this.ejbSelectByTopAccounts();
    ...
    return topAccounts;
}

/**
 * Method to call ejbSelectCreditLimit and return the credit limit value
 * for the input accountnumber without post-processing.
 * Please note that this method returns a Long instead of a collection
 * that is returned normally by the EJB container. This is a major
 * advantage of ejbSelect methods. Using these methods, You can return
 * an object from 'within' the CMP instead of 'the' CMP. This way, the
 * application uses the server and the EJB container resources more
 * effeciently.
 *
 * @return Credit Limit of the input UserAccount
 */
public Long ejbHomeCreditLimit(Long accountnumber) throws FinderException {
```

```
        // Return the Credit Limit of the specified Account
        return this.ejbSelectCreditLimit(accountnumber);
    }
    ...
}
```

Using Deployment XML

Example 16–4 shows the `ejb-jar.xml` file for the select methods defined in the abstract entity bean class that Example 16–3 shows.

Example 16–4 ejb-jar.xml For EJB 2.1 EJB QL Select Methods

```
<enterprise-beans>
  <entity>
    <description>Entity Bean ( CMP )</description>
    <display-name>UserAccount</display-name>
    <ejb-name>UserAccount</ejb-name>
    <local-home>oracle.otnsamples.ejbql.UserAccountLocalHome</local-home>
    <local>oracle.otnsamples.ejbql.UserAccount</local>
    <ejb-class>oracle.otnsamples.ejbql.UserAccountBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.Long</prim-key-class>
    <abstract-schema-name>UserAccount</abstract-schema-name>
    <cmp-field>
      <field-name>accountnumber</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>creditlimit</field-name>
    </cmp-field>
    <primkey-field>accountnumber</primkey-field>
    <query>
      <description>Selects all accounts and post-process to find top accounts</description>
      <query-method>
        <method-name>ejbSelectByTopAccounts</method-name>
      </query-method>
      <ejb-ql>select distinct object(ua) from UserAccount ua</ejb-ql>
    </query>
    <query>
      <description>Retrieves the Credit Limit for an Account</description>
      <query-method>
        <method-name>ejbSelectCreditLimit</method-name>
        <method-params>
          <method-param>java.lang.Long</method-param>
        </method-params>
      </query-method>
      <ejb-ql>
          select ua.creditlimit from UserAccount ua where ua.accountnumber = ?1
      </ejb-ql>
    </query>
  </entity>
</enterprise-beans>
```

Using TopLink Workbench

Using the TopLink Workbench, you can configure your `toplink-ejb-jar.xml` file with a custom TopLink `ejbSelect` method and update your `ejb-jar.xml` file.

For more information, see "Creating a Finder" in the *Oracle TopLink Developer's Guide*

OC4J EJB 2.1 EJB QL Extensions

Although EJB 2.1 does not support square root, date, time, and timestamp types, OC4J provides proprietary EJB QL extensions to support these types in EJB 2.1, as follows:

- `SQRT(v)`: Both the double primitive type and the `java.lang.Double` types are supported for arguments (see Example 16–5).
- You can use the following date, time, and timestamp types in an EJB QL binary expression, such as the following equality expressions:
  - `java.util.Date` (see Example 16–6
  - `java.sql.Date` (see Example 16–7)
  - `java.sql.Time` (see Example 16–8)
  - `java.sql.Timestamp` (see Example 16–9)

Note: These types are fully supported in EJB 3.0 EJB QL.

Example 16–5 Using the EJB 2.1 EJB QL Extension for SQRT

```
<query>
  <query-method>
    <method-name>ejbSelectDoubleTypeSqrt</method-name>
    <method-params>
      <method-param>double</method-param>
    </method-params>
  </query-method>
  <result-type-mapping>Remote</result-type-mapping>
  <ejb-ql>
    SELECT OBJECT(a) FROM Dept a WHERE a.deptDoubleType = SQRT(?1)
  </ejb-ql>
</query>
```

Example 16–6 Using the EJB 2.1 EJB QL Extension for java.util.Date

```
<query>
  <query-method>
   <method-name>ejbSelectDate</method-name>
   <method-params>
    <method-param>java.util.Date</method-param>
   </method-params>
  </query-method>
  <result-type-mapping>Remote</result-type-mapping>
  <ejb-ql>
    SELECT OBJECT(a) FROM Dept a WHERE a.deptDate = ?1
   </ejb-ql>
</query>
```

Example 16–7 Using the EJB 2.1 EJB QL Extension for java.sql.Date

```
<query>
  <query-method>
   <method-name>ejbSelectSqlDate</method-name>
   <method-params>
    <method-param>java.sql.Date</method-param>
   </method-params>
  </query-method>
  <result-type-mapping>Remote</result-type-mapping>
  <ejb-ql>
```

```
    SELECT OBJECT(a) FROM Dept a WHERE a.deptSqlDate = ?1
  </ejb-ql>
</query>
```

Example 16–8 Using the EJB 2.1 EJB QL Extension for java.sql.Time

```
<query>
  <query-method>
   <method-name>findByTimestamp</method-name>
   <method-params>
    <method-param>java.sql.Time</method-param>
   </method-params>
  </query-method>
  <result-type-mapping>Remote</result-type-mapping>
  <ejb-ql>
    SELECT OBJECT(a) FROM Dept a WHERE a.deptTime = ?1
  </ejb-ql>
</query>
```

Example 16–9 Using the EJB 2.1 EJB QL Extension for java.sql.Timestamp

```
<query>
  <query-method>
   <method-name>findByTimestamp</method-name>
   <method-params>
    <method-param>java.sql.Timestamp</method-param>
   </method-params>
  </query-method>
  <result-type-mapping>Remote</result-type-mapping>
  <ejb-ql>
    SELECT OBJECT(a) FROM Dept a WHERE a.deptTimestamp = ?1
  </ejb-ql>
</query>
```

Part VII

EJB 2.1 Message-Driven Beans

This part provides procedural information on implementing and configuring EJB 2.1 message-driven beans. For conceptual information, see Part I, "EJB Overview".

This part contains the following chapters:

- Chapter 17, "Implementing an EJB 2.1 Message-Driven Bean"
- Chapter 18, "Using an EJB 2.1 Message-Driven Bean"

17

Implementing an EJB 2.1 Message-Driven Bean

This chapter explains how to implement an EJB 2.1 message-driven bean (MDB).

For more information, see the following:

- "What is a Message-Driven Bean?" on page 1-56
- "Using an EJB 2.1 Message-Driven Bean" on page 18-1

Implementing an EJB 2.1 MDB

Table 17–1 summarizes the important parts of an EJB 2.1 message-driven bean and the following procedure describes how to implement these parts. For a typical implementation, see "Using Java" on page 17-3.

Table 17–1 Parts of an EJB 2.1 MDB Entity Bean

Part	Description
Bean implementation	This class must be declared as `public`, contain a `public`, empty, default constructor, one `public`, `void ejbCreate` method with no arguments, and no `finalize`() method. Implements `javax.ejb.MessageDrivenBean` to provide an empty implementation for life cycle method `ejbRemove` and an implementation of the `setMessageDrivenContext` method. Implements `javax.jms.MessageListener` to provide an implementation of the `onMessage` method.

For more information, see "What is a Message-Driven Bean?" on page 1-56.

> **Note:** You can download EJB code examples from: `http://www.oracle.com/technology/tech/java/oc4j/demos`.

To implement an EJB 2.1 message-driven bean, do the following:

1. Implement the MDB entity bean:
   a. Implement a `public`, zero-argument constructor.
   b. Implement any methods that are private to the bean or package used for facilitating the business logic. This includes private methods that your public methods use for completing the tasks requested of them.

c. Implement the `ejbCreate` method. The container invokes this method when it instantiates the MDB.

   The return type of the `ebjCreate` methods is `void`.

d. Provide an empty implementation for each of the `javax.ejb.MessageDrivenBean` interface container callback methods.

   For more information, see "Configuring a Life Cycle Callback Method for an EJB 2.1 MDB" on page 18-10.

e. Implement a `setMessageDrivenContext` method that takes an instance of `MessageDrivenContext` (see "Implementing the setMessageDrivenContext Method" on page 17-6).

f. Implement the appropriate message listener interface:

   For a JMS message-driven bean, implement the `javax.jms.MessageListener` interface to provide the `onMessages` method with signature:

   ```
   public void onMessage(javax.jms.Message message)
   ```

   For a non-JMS message service provider, implement the message listener interface (or interfaces) it specifies.

   This method processes the incoming message. Most MDBs receive messages from a queue or a topic, then invoke an entity bean to process the request contained within the message.

2. Configure message service provider information (see "Using Deployment XML" on page 17-4:

   a. Define the message connection factory and `Destination` used in the EJB deployment descriptor (`ejb-jar.xml`). Define if any durable subscriptions or message selectors are used.

      For more information, see the following:

      – "Configuring an EJB 2.1 MDB to Access a Message Service Provider Directly" on page 18-3

      – "Configuring an EJB 2.1 MDB to Access a Message Service Provider Using J2CA" on page 18-1

   b. If using resource references, define these in the `ejb-jar.xml` file and map them to their actual JNDI names in the OC4J-specific deployment descriptor (`orion-ejb-jar.xml`).

   c. If the MDB uses container-managed transaction demarcation, specify the `onMessage` method in the `<container-transaction>` element in the `ejb-jar.xml` file.

      All of the steps for an MDB should be in the `onMessage` method. Since the MDB is stateless, the `onMessage` method should perform all duties.

      In general, do not create the message service connection and session in the `ejbCreate` method.

      Note: If you are using OEMS JMS (see "OEMS JMS: In-Memory or File-Based Provider" on page 2-23), then you can optimize your MDB by creating the JMS connection and session in the `ejbCreate` method and destroying them in the `ejbRemove` method.

Using Java

Example 17–1 shows a typical implementation of an EJB 2.1 MDB.

Example 17–1 EJB 2.1 MDB Implementation

```
import java.util.*;
import javax.ejb.*;
import javax.jms.*;
import javax.naming.*;

public class rpTestMdb implements MessageDrivenBean, MessageListener {

    private QueueConnection         m_qc    = null;
    private QueueSession            m_qs    = null;
    private QueueSender             m_snd   = null;
    private MessageDrivenContext    m_ctx   = null;

    // Constructor, which is public and takes no arguments
    public rpTestMdb() {
    }

    /**
     * Begin private methods. The following methods
     * are used internally.
     */

...

    /**
     * Begin EJB-required methods. The following methods are called
     * by the container, and never called by client code.
     */

    /**
     * ejbCreate method, declared as public (but not final or
     * static), with a return type of void, and with no arguments.
     */
    public void ejbCreate() {
    }

    public void setMessageDrivenContext(MessageDrivenContext ctx) {
      // As with all enterprise beans, you must set the context in order to be
      // able to use it at another time within the MDB methods
      m_ctx = ctx;
    }

    // life cycle Methods

    public void ejbRemove() {
    }


    /**
     * JMS MessageListener-required methods. The following
     * methods are called by the container, and never called by
     * client code.
     */

    // Receives the incoming Message and displays the text.
    public void onMessage(Message msg) {
       // MDB does not carry state for an individual client
       try {
            Context ctx = new InitialContext();
            // 1. Retrieve the QueueConnectionFactory using a
            // resource reference defined in the ejb-jar.xml file.
```

```
            QueueConnectionFactory qcf = (QueueConnectionFactory)
                ctx.lookup("java:comp/env/jms/myQueueConnectionFactory");
            ctx.close();

            // 2. Create the queue connection
            m_qc = qcf.createQueueConnection();
            // 3. Create the session over the queue connection.
            m_qs = m_qc.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
            // 4. Create the sender to send messages over the session.
            m_snd = m_qs.createSender(null);

            // When the onMessage method is called, a message has been sent.
            // You can retrieve attributes of the message using the Message object.
            String txt = ("mdb rcv: " + msg.getJMSMessageID());
            System.out.println(txt + " redel="
                + msg.getJMSRedelivered() + " cnt="
                + msg.getIntProperty("JMSXDeliveryCount"));

            // Create a new message using the createMessage method.
            // To send it back to the originator of the other message,
            // set the String property of "RECIPIENT" to "CLIENT."
            // The client only looks for messages with string property CLIENT.
            // Copy the original message ID into new msg's Correlation ID for
            // tracking purposes using the setJMSCorrelationID method. Finally,
            // set the destination for the message using the getJMSReplyTo method
            // on the previously received message. Send the message using the
            // send method on the queue sender.

            // 5. Create a message using the createMessage method
            Message rmsg = m_qs.createMessage();
            // 6. Set properties of the message.
            rmsg.setStringProperty("RECIPIENT", "CLIENT");
            rmsg.setIntProperty("count", msg.getIntProperty("JMSXDeliveryCount"));
            rmsg.setJMSCorrelationID(msg.getJMSMessageID());
            // 7. Retrieve the reply destination.
            Destination d = msg.getJMSReplyTo();
            // 8. Send the message using the send method of the sender.
            m_snd.send((Queue) d, rmsg);
            System.out.println(txt + " snd: " + rmsg.getJMSMessageID());
            // close the connection
            m_qc.close();
        }
        catch (Throwable ex) {
            ex.printStackTrace();
        }
    }
}
```

Using Deployment XML

Using the `ejb-jar.xml` file, define the MDB name, class, JNDI reference, and JMS `Destination` type (queue or topic) in the `message-driven` element. If a topic is specified, you define whether it is durable. If you have used resource references, define the resource reference for both the connection factory and the `Destination` object.

Example 17–2 shows the `ejb-jar.xml` file `message-driven` element corresponding to the MDB shown in Example 17–1.

Note the following:

- MDB name specified in the `<ejb-name>` element.
- MDB class defined in the `<ejb-class>` element, which ties the `<message-driven>` element to the specific MDB implementation.

- JMS `Destination` type is a `Queue` that is specified in the `<message-driven-destination><destination-type>` element.
- Message selector specifies that this MDB only receives messages where the `RECIPIENT` is `MDB`.

> **Note:** You could also specify a topic in this type definition. If you did specify a `Topic` in the type, then you could also define the durability of the topic, which is specified in the `<message-driven-destination><subscription-durability>` element as "`Durable`" or "`nonDurable`."

- The type of transaction to use is defined in the `<transaction-type>` element. The value can be `Container` or `Bean`. If `Container` is specified, define the `onMessage` method within the `<container-transaction>` element with the type of CMT support.
- The resource reference for the connection factory is defined in the `<resource-ref>` element; the resource reference for the `Destination` object is defined in the `<resource-env-ref>` element.

Example 17–2 ejb-jar.xml For an EJB 2.1 MDB

```
...
    <enterprise-beans>
        <message-driven>
            <display-name>testMdb</display-name>
            <ejb-name>testMdb</ejb-name>
            <ejb-class>rpTestMdb</ejb-class>
            <transaction-type>Container</transaction-type>
            <message-selector>RECIPIENT='MDB'</message-selector>
            <message-driven-destination>
                <destination-type>javax.jms.Queue</destination-type>
            </message-driven-destination>
            <resource-ref>
                <description>description</description>
                <res-ref-name>jms/myQueueConnectionFactory</res-ref-name>
                <res-type>javax.jms.QueueConnectionFactory</res-type>
                <res-auth>Application</res-auth>
                <res-sharing-scope>Shareable</res-sharing-scope>
            </resource-ref>
            <resource-env-ref>
               <resource-env-ref-name>jms/persistentQueue
             </resource-env-ref-name>
             <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
            </resource-env-ref>
        </message-driven>
    </enterprise-beans>

    <assembly-descriptor>
        <container-transaction>
            <method>
                <ejb-name>testMdb</ejb-name>
                <method-name>onMessage</method-name>
                <method-params>
                    <method-param>javax.jms.Message</method-param>
                </method-params>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>
    </assembly-descriptor>
```

```
...
```

If you were going to configure a durable `Topic` instead, then the `<message-driven-destination>` element would be configured Example 17–3.

Example 17–3 ejb-jar.xml For an EJB 2.1 MDB for a Durable Topic

```
<message-driven-destination>
  <destination-type>javax.jms.Topic</destination-type>
  <subscription-durability>Durable</subscription-durability>
</message-driven-destination>
```

For more information, see "Configuring an EJB 2.1 MDB to Access a Message Service Provider Directly" on page 18-3.

Implementing the setMessageDrivenContext Method

An MDB instance uses this method to retain a reference to its context. Message-driven beans have contexts that the container maintains and makes available to the beans. The bean may use the methods in the message-driven context to retrieve information about the bean, such as security, and transactional role. Refer to the EJB specification from Sun Microsystems for the full range of information that you can retrieve about the bean from the context.

The container invokes the `setMessageDrivenContext` method, after it first instantiates the bean, to enable the bean to retrieve the context. The container will never call this method from within a transaction context. If the bean does not save the context at this point, the bean will never gain access to the context.

Example 17–4 shows an MDB saving the message-driven context in the `ctx` variable.

Example 17–4 Implementing the setMessageDrivenContext Methods

```
import javax.ejb.*;

public class myBean implements MessageDrivenBean, MessageListener {

   MessageDrivenContext m_ctx;

    // setMessageDrivenContext method
    public void setMessageDrivenContext(MessageDrivenContext ctx) {
      // As with all enterprise beans, you must set the context in order to be
      // able to use it at another time within the MDB methods
      m_ctx = ctx;
    }

   // other methods in the bean
}
```

18

Using an EJB 2.1 Message-Driven Bean

This chapter describes the various options that you must configure in order to use an EJB 2.1 message-driven bean.

Table 18–1 lists these options and indicates which are basic (applicable to most applications) and which are advanced (applicable to more specialized applications).

For more information, see the following:

- "What is a Message-Driven Bean?" on page 1-56
- "Implementing an EJB 2.1 Message-Driven Bean" on page 17-1

Table 18–1 Configurable Options for an EJB 2.1 Message-Driven Bean

Options	Type
"Configuring an EJB 2.1 MDB to Access a Message Service Provider Using J2CA" on page 18-1	Basic
"Configuring an EJB 2.1 MDB to Access a Message Service Provider Directly" on page 18-3	Basic
"Configuring an MDB for Fast Undeploy on Windows Operating System" on page 18-5	Advanced
"Configuring an MDB for Oracle RAC Failover" on page 18-6	Advanced
"Configuring Bean Instance Pool Size" on page 31-4	Basic
"Configuring a Transaction Timeout for a Message-Driven Bean" on page 21-7	Advanced
"Configuring Parallel Message Processing" on page 18-7	Advanced
"Configuring Connection Failure Recovery for an EJB 2.1 MDB" on page 18-9	Advanced
"Configuring a Life Cycle Callback Method for an EJB 2.1 MDB" on page 18-10	Basic

Configuring an EJB 2.1 MDB to Access a Message Service Provider Using J2CA

You can configure an EJB 2.1 MDB to access a message service provider using a J2CA resource adapter such as the Oracle JMS Connector.

You can do this using deployment XML (see "Using Deployment XML" on page 18-2).

> **Note:** Oracle recommends that you access a message service provider using a J2CA resource adapter such as the Oracle JMS Connector. For more information, see "Restrictions When Accessing a Message Service Provider Without a J2CA Resource Adapter" on page 2-25.

OC4J supports both XA factories for two-phase commit (2PC) transactions and non-XA factories for transactions that do not require 2PC.

For more information, see:

- "Oracle JMS Connector: J2EE Connector Architecture (J2CA)-Based Provider" on page 2-21
- "Message Service Configuration Options: Annotations or XML? Attributes or Activation Configuration Properties?" on page 2-26
- "How do You Participate in a Global or Two-Phase Commit (2PC) Transaction?" on page 2-20

Using Deployment XML

To configure an EJB 2.1 MDB to access a JMS message service provider using a J2CA resource adapter by using deployment XML, you must use both `ejb-jar.xml` and `orion-ejb.jar.xml` files. You use the `orion-ejb-jar.xml` file configuration to override settings in `ejb-jar.xml` and to add the OC4J-specific setting for resource adapter. For example, the connection factory and destination name that you define in `ejb-jar.xml` may be logical names that may not exist in your local JNDI environment. The deployer can override these settings in the `orion-ejb-jar.xml` file and map them to the actual names. For more information on mapping logical names, see "Configuring an Environment Reference to a JMS Destination or Connection Resource Manager Connection Factory (JMS 1.0)" on page 19-14.

To configure an EJB 2.1 MDB to use a J2CA message service provider:

1. Specify the name of the resource adapter.

   You do this using the `orion-ejb-jar.xml` file `<message-driven-deployment>` element `resource-adapter` attribute as Example 18–1 shows.

2. Specify the required activation configuration properties.

   You may specify activation configuration properties using any combination of `<config-property>` elements in the `orion-ejb-jar.xml` file `<message-driven-deployment>` element (as Example 18–1 shows) and `<activation-config-property>` elements in the `ejb-jar.xml` file `<message-driven>` element (as Example 18–2 shows). The `orion-ejb-jar.xml` file configuration overrides that in the `ejb-jar.xml` file.

   For more information, see:

   - "J2CA Activation Configuration Properties" on page B-1
   - "Message Service Configuration Options: Annotations or XML? Attributes or Activation Configuration Properties?" on page 2-26

Example 18–1 shows how to configure the `orion-ejb-jar.xml` file to configure this message-driven bean to use the Oracle JMS resource adapter named `OracleASjms`. You must set the `resource-adapter` attribute. Optionally, you can override or configure additional activation configuration properties using one or more `config-property` elements.

Example 18–1 orion-ejb-jar.xml for a J2CA Message Service Provider

```
<message-driven-deployment
    name="JCA_QueueMDB"
    resource-adapter="OracleASjms">
    ...
    <config-property>
        <config-property-name>DestinationName</config-property-name>
        <config-property-value>OracleASJMSRASubcontext/MyQ</config-property-value>
```

```
    </config-property>
    ...
</message-driven-deployment>
```

Example 18–2 shows how to configure `ejb-jar.xml` to configure a message-driven bean to use the Oracle JMS resource adapter named `OracleASjms`. It assumes that you defined connection factory `OracleASjms/MyQCF` in the `oc4j-ra.xml` file and destination name `OracleASjms/MyQueue` in the `oc4j-connectors.xml` when you configured your message service provider. You can define either XA-enabled factories for two-phase commit (2PC) support, or non-XA factories if 2PC support is not required. For more information, see "Configuring Message Services" on page 23-1.

Example 18–2 ejb-jar.xml for a J2CA Message Service Provider

```
<message-driven>
    <ejb-name>JCA_QueueMDB</ejb-name>
    <ejb-class>test.JCA_MDB</ejb-class>
    <messaging-type>javax.jms.MessageListener</messaging-type>
    <transaction-type>Container</transaction-type>

    <activation-config>
        <activation-config-property>
            <activation-config-property-name>
                DestinationType
            </activation-config-property-name>
            <activation-config-property-value>
                javax.jms.Queue
            </activation-config-property-value>
        </activation-config-property>
        <activation-config-property>
            <activation-config-property-name>
                DestinationName
            </activation-config-property-name>
            <activation-config-property-value>
                OracleASjms/MyQueue
            </activation-config-property-value>
        </activation-config-property>
        <activation-config-property>
            <activation-config-property-name>
                ConnectionFactoryJndiName
            </activation-config-property-name>
            <activation-config-property-value>
                OracleASjms/MyQCF
            </activation-config-property-value>
        </activation-config-property>
    </activation-config>
</message-driven>
```

You may also set the optional attributes that Table A–3 lists.

The actual names you use depend on your message service provider installation. For more information, see "J2CA Message Service Provider Connection Factory Names" on page 23-2.

Configuring an EJB 2.1 MDB to Access a Message Service Provider Directly

You can configure an EJB 2.1 MDB to access a message service provider directly (without a J2CA resource adapter).

Note: Oracle recommends that you access a message service provider using a J2CA resource adapter such as the Oracle JMS Connector. For more information, see:

- "Restrictions When Accessing a Message Service Provider Without a J2CA Resource Adapter" on page 2-25.
- "Configuring an EJB 2.1 MDB to Access a Message Service Provider Using J2CA" on page 18-1

You can do this by using deployment XML (see "Using Deployment XML" on page 18-4).

OC4J supports both XA factories for two-phase commit (2PC) transactions and non-XA factories for transactions that do not require 2PC. For more information on 2PC support, see "How do You Participate in a Global or Two-Phase Commit (2PC) Transaction?" on page 2-20.

Using Deployment XML

To configure an EJB 2.1 MDB to access a JMS message service provider directly (without a J2CA resource adapter) by using deployment XML, you can use either the `ejb-jar.xml` or `orion-ejb.jar.xml` file. You use the `orion-ejb-jar.xml` file configuration to override settings in `ejb-jar.xml` or to add OC4J-specific settings. For example, the connection factory and destination name that you define in `ejb-jar.xml` may be logical names that may not exist in your local JNDI environment. The deployer can override these settings in the `orion-ejb-jar.xml` file and map them to the actual names. For more information on mapping logical names, see "Configuring an Environment Reference to a JMS Destination or Connection Resource Manager Connection Factory (JMS 1.0)" on page 19-14.

To configure

1. Specify the required activation configuration properties.

   You may specify activation configuration properties using any combination of `<config-property>` elements in the `orion-ejb-jar.xml` file `<message-driven-deployment>` element and `<activation-config-property>` elements in the `ejb-jar.xml` file `<message-driven>` element (as Example 18–3 shows). The `orion-ejb-jar.xml` file configuration overrides that in the `ejb-jar.xml` file.

   For more information, see:

   - "J2CA Activation Configuration Properties" on page B-1
   - "Message Service Configuration Options: Annotations or XML? Attributes or Activation Configuration Properties?" on page 2-26

Example 18–3 shows how to configure `ejb-jar.xml` to configure a message-driven bean to use a non-J2CA JMS message service provider. It assumes that you defined connection factory `jms/MyQCF` and queue `jms/MyQueue` when you configured your message service provider. You can define either XA-enabled factories for two-phase commit (2PC) support or non-XA factories if 2PC support is not required. For more information, see "Configuring Message Services" on page 23-1.

Example 18–3 ejb-jar.xml for a Non-J2CA Message Service Provider

```
<message-driven>
```

```
    <ejb-name>QueueMDB</ejb-name>
    <ejb-class>test.QueueMDB</ejb-class>
    <message-destination-type>javax.jms.Queue</message-destination-type>
    <transaction-type>Container</transaction-type>

    <activation-config>
        <activation-config-property>
            <activation-config-property-name>
                ConnectionFactoryJndiName
            </activation-config-property-name>
            <activation-config-property-value>
                jms/MyQCF
            </activation-config-property-value>
        </activation-config-property>
        <activation-config-property>
            <activation-config-property-name>
                DestinationName
            </activation-config-property-name>
            <activation-config-property-value>
                jms/MyQueue
            </activation-config-property-value>
        </activation-config-property>
    </activation-config>
</message-driven>
```

The actual names you use depend on your message service provider installation. For more information, see the following:

- "OEMS JMS Destination and Connection Factory Names" on page 23-3
- "OEMS JMS Database Destination and Connection Factory Names" on page 23-6

Configuring an MDB for Fast Undeploy on Windows Operating System

When you use an MDB, it is blocked in a receive state waiting for incoming messages. In a non-Windows environment, if you shut down OC4J while the MDB is in a wait state, OC4J shuts down in a timely fashion.

If you are using message-driven beans with the OEMS JMS Database provider (see "OEMS JMS Database: Advanced Queueing (AQ)-Based Provider" on page 2-24) and OC4J is running in a Windows environment, or when the back-end database is running in a Windows environment and you shutdown OC4J while an MDB is in a wait state, then the OC4J instance cannot be stopped and the MDB cannot be undeployed in a timely manner: in this case, the OC4J process will hang for at least 2.5 hours

Using the `oracle.mdb.fastUndeploy` system property (see "Using System Properties" on page 18-5), you can modify the behavior of the MDB in the Windows environment to ensure that your message-driven beans can be undeployed, and OC4J can be shut down, in a timely manner, when necessary.

Using System Properties

The `oracle.mdb.fastUndeploy` system property is set to the frequency, as an integer number of seconds, at which OC4J polls the database (requiring a database round-trip) to determine whether or not the session is shut down when an MDB is not processing incoming messages and in a wait state.

For optimal performance, a reasonable value should be 120 seconds or more.

If you set this property to 120 (seconds), then every 120 seconds, OC4J will poll the database.

Configuring an MDB for Oracle RAC Failover

If your MDB application uses OEMS JMS Database with an Oracle RAC database, you must configure your application to handle a database failover scenario, as follows:

- Configure message-driven beans to retry if message dequeing fails (see "Using Deployment XML" on page 18-6)
- Configure the MDB client to retry if connection acquisition fails (see "Using Java" on page 18-6)

> **Note:** The Oracle RAC-enabled attribute of a data source is discussed in Data Sources chapter in the Oracle Containers for J2EE Services Guide.

Using Deployment XML

To support Oracle RAC failover, you must configure `orion-ejb-jar.xml` file element `message-driven-deployment` attributes `dequeue-retry-count` and `dequeue-retry-interval`, as Example 18–4 shows.

The `dequeue-retry-count` attribute tells the container how many times to retry the database connection in case a failure happens; the default is 0 seconds.

The `dequeue-retry-interval` attribute tells the container how long to wait between retry attempts to accommodate for the time it takes for Oracle RAC database failover to complete; the default value is 60 seconds.

Example 18–4 orion-ejb-jar.xml For Oracle RAC Failover with an MDB

```
<message-driven-deployment name="MessageBeanTpc"
   connection-factory-location="java:comp/resource/cartojms1/TopicConnectionFactories/aqTcf"
   destination-location="java:comp/resource/cartojms1/Topics/topic1"
   subscription-name="MDBSUB"
   dequeue-retry-count=3
   dequeue-retry-interval=90/>
...
```

Using Java

To support Oracle RAC failover, you must configure a standalone OEMS JMS Database client running against an Oracle RAC database to retry if connection acquisition fails.

Oracle recommends that you use `com.evermind.sql.DbUtil` method `oracleFatalError` to determine if the connection object is invalid (see Example 18–5). If so, then reestablish the database connection, if necessary.

Example 18–5 Client Retrying After Connection Acquisition Failure

```
import com.evermind.sql.DbUtil;
...
getMessage(QueueSesssion session) {
    try {
        QueueReceiver rcvr = session.createReceiver(rcvrQueue);
        Message msgRec = rcvr.receive();
    }
    catch(Exception e ) {
        if (exc instanceof JMSException) {
            JMSException  jmsexc = (JMSException) exc;
```

```
            sql_ex = (SQLException)(jmsexc.getLinkedException());
            db_conn = oracle.jms.AQjmsSession)session.getDBConnection();
            if ((DbUtil.oracleFatalError(sql_ex, db_conn)) {
               // failover logic
            }
        }
    }
}
```

Configuring Parallel Message Processing

By default, OC4J uses one receiver thread to poll for messages from the message location.

Having more than one receiver thread allows messages to be received in parallel which can improve performance.

If your message location is a Topic, the number of receiver threads is fixed to one.

If your message location is a Queue, you can configure the number of receiver threads (see "Using Deployment XML" on page 18-7).

Note that the minimum number of bean instances in the MDB pool should be at least the same as the number of receiver threads to avoid blocking receiver threads from acquiring a bean instance from the pool to process messages.

For more information, see:

- "Message Service Configuration Options: Annotations or XML? Attributes or Activation Configuration Properties?" on page 2-26
- "Configuring Bean Instance Pool Size" on page 31-4

Using Deployment XML

You configure parallel message processing in the `orion-ejb-jar.xml` file. How you configure this option depends on the type of message-service provider you are using:

- J2CA Adapter Message Service Provider
- Non-J2CA Adapter Message Service Provider

In either case, you must restart OC4J to apply your changes.

J2CA Adapter Message Service Provider

If you are using a J2CA adapter message service provider, use the `<config-property>` element to set the `ReceiverThreads` configuration property.

For example, if you are using a J2CA adapter message service provider, and you want three message-driven bean instances receiving from the message location in parallel, set the `ReceiverThreads` configuration property to `3`, as follows:

```
<message-driven-deployment ... >
...
    <config-property>
        <config-property-name>RecieverThreads</config-property-name>
        <config-property-value>3</config-property-value>
    </config-property>
...
</message-driven-deployment>
```

For more information on `ReceiverThreads`, see Table B–2.

Non-J2CA Adapter Message Service Provider

If you are using a non-J2CA adapter message service provider like OEMS JMS or OEMS JMS Database, use the `<message-driven-deployment>` element `listener-threads` attribute.

For example, if you are using OEMS JMS or OEMS JMS Database, and you want three message-driven bean instances receiving from the message location in parallel, set the `listener-threads` attribute to `3`, as follows:

```
<message-driven-deployment ...  listener-threads="3"
  ...
</message-driven-deployment>
```

For more information on `listener-threads`, see Table A–3.

Note: Oracle recommends that you access a message service provider using a J2CA resource adapter such as the Oracle JMS Connector. For more information, see:

- "Restrictions When Accessing a Message Service Provider Without a J2CA Resource Adapter" on page 2-25.
- "Configuring an EJB 2.1 MDB to Access a Message Service Provider Using J2CA" on page 18-1

Configuring Maximum Delivery Count

You can configure the maximum number of times OC4J will attempt the immediate re delivery of a message to a message-driven bean's `onMessage` method if that method returns failure: fails to invoke an acknowledgment operation, throws an exception, or both (see "Using Deployment XML" on page 18-8).

After this number of redeliveries, the message is deemed undeliverable and is handled according to the policies of your message service provider. For example, OEMS JMS will put the message on its exception queue (`jms/Oc4jJmsExceptionQueue`).

Using Deployment XML

You set the maximum delivery count in the `orion-ejb-jar.xml` file. How you configure this value depends on the type of message-service provider you are using:

- J2CA Adapter Message Service Provider
- Non-J2CA Adapter Message Service Provider

J2CA Adapter Message Service Provider

If you are using a J2CA adapter message service provider, use the `<config-property>` element to set the `MaxDeliveryCnt` configuration property.

For example, if you are using a J2CA adapter message service provider, and you wanted to set the maximum delivery count to 3, you would do as follows:

```
<message-driven-deployment ... >
...
    <config-property>
        <config-property-name>MaxDeliveryCnt</config-property-name>
        <config-property-value>3</config-property-value>
    </config-property>
...
</message-driven-deployment>
```

For more information on `MaxDeliveryCnt`, see Table B–2.

Non-J2CA Adapter Message Service Provider

If you are using a non-J2CA adapter message service provider like OEMS JMS (in-memory/file based), use the `max-delivery-count` attribute of the `<message-driven-deployment>` element.

For example, if you wanted to set the maximum delivery count to 3, you would do as follows:

```
<message-driven-deployment ...  max-delivery-count="3"
  ...
</message-driven-deployment>
```

For more information on `max-delivery-count`, see Table A–3.

Note: Oracle recommends that you access a message service provider using a J2CA resource adapter such as the Oracle JMS Connector. For more information, see:

- "Restrictions When Accessing a Message Service Provider Without a J2CA Resource Adapter" on page 2-25.
- "Configuring an EJB 2.1 MDB to Access a Message Service Provider Using J2CA" on page 18-1

Configuring Connection Failure Recovery for an EJB 2.1 MDB

You can configure how a message-driven bean's listener thread responds to connection failures due to such events as network and JMS server outages.

These options are applicable to only container-managed transactions in a message-driven bean.

You can configure connection failure recovery options using the `orion-ejb-jar.xml` file (see "Using Deployment XML" on page 18-9).

For more information about failover, see "Understanding OC4J EJB Application Clustering Services" on page 2-29.

Using Deployment XML

You set the dequeue retry count and interval in the `orion-ejb-jar.xml` file. How you configure this value depends on the type of message-service provider you are using:

- J2CA Adapter Message Service Provider
- Non-J2CA Adapter Message Service Provider

In either case, you must restart OC4J to apply your changes.

J2CA Adapter Message Service Provider

If you access your message-service provider using a J2CA resource adapter, the Oracle JMS Connector does an infinite retry of polling for the JMS resource and this retry interval can be configured in the activation configuration property, `EndpointFailureRetryInterval` as shown in Example 18–6.

Note that the recovery of message after retry does not guarantee message ordering, and messages can be lost or duplicated when MDB subscription to the JMS topic is non-durable.

For more information, see `EndpointFailureRetryInterval` in Table B–2.

Example 18–6 Configuring EndpointFailureRetryInterval in orion-ejb-jar.xml

```
<message-driven-deployment ... >
...
    <config-property>
        <config-property-name>EndpointFailureRetryInterval</config-property-name>
        <config-property-value>20000</config-property-value>
    </config-property>
...
</message-driven-deployment>
```

Non-J2CA Adapter Message Service Provider

If you are using a non-J2CA adapter message service provider like OEMS JMS or OEMS JMS Database, use the `dequeue-retry-count` and `dequeue-retry-interval` attribute of the `<message-driven-deployment>` element. The default dequeue retry count is zero and the default dequeue retry interval is 60 seconds.

For example, if you are using OEMS JMS or OEMS JMS Database, and you wanted to set the dequeue retry count to 3 and the dequeue retry interval to 90 seconds, you would do as follows:

```
<message-driven-deployment ... dequeue-retry-count="3" dequeue-retry-interval="90"
  ...
</message-driven-deployment>
```

For more information on `dequeue-retry-count` and `dequeue-retry-interval`, see Table A–3.

Note: Oracle recommends that you access a message service provider using a J2CA resource adapter such as the Oracle JMS Connector. For more information, see:

- "Restrictions When Accessing a Message Service Provider Without a J2CA Resource Adapter" on page 2-25.
- "Configuring an EJB 2.1 MDB to Access a Message Service Provider Using J2CA" on page 18-1

Configuring a Life Cycle Callback Method for an EJB 2.1 MDB

The following are the EJB 2.1 life cycle methods, as specified in the `javax.ejb.MessageDrivenBean` interface, that a message-driven bean must implement (see "Using Java" on page 18-11):

- `ejbCreate`
- `ejbRemove`

Note: Using EJB 2.1, you must implement all message-driven bean callback methods. If you do not need to take any action, implement an empty method.

For more information, see "What is the Life Cycle of a Message-Driven Bean?" on page 1-57.

Using Java

Example 18–7 shows how to implement an EBJ 2.1 message-driven bean life cycle callback method.

Example 18–7 EJB 2.1 MDB Life Cycle Callback Method Implementation

```
public void ejbRemove() {
    // when bean is removed
}
```

Part VIII

Configuring OC4J EJB Services

This part provides procedural information on configuring OC4J EJB services for EJB 3.0 and EJB 2.1 enterprise JavaBeans. For conceptual information, see Part I, "EJB Overview".

This part contains the following chapters:

19

Configuring JNDI Services

This chapter describes the following:

- Configuring Environment References
- Configuring the Initial Context Factory
- Setting JNDI Properties in an Enterprise Bean
- Looking Up an EJB 3.0 Resource Manager Connection Factory
- Looking Up an EJB 3.0 Environment Variable
- Looking Up an EJB 2.1 Resource Manager Connection Factory
- Looking Up an EJB 2.1 Enviornment Variable

For more information, see the following:

- "Understanding EJB JNDI Services" on page 2-14
- "Accessing an Enterprise Bean From a Client" on page 29-1
- "Oracle JNDI" in the *Oracle Containers for J2EE Services Guide*

Configuring Environment References

Before you can access essential resources from your EJB at run time using JNDI, you must define environment references to them. Environment references are static and cannot be changed by the bean.

This section describes configuring the following:

- EJB Environment References
- Resource Manager Connection Factory Environment References
- Environment Variable Environment References
- Web Service Environment References
- Persistence Context References

In EJB 3.0, instead of defining environment references, you can use annotations, resource injection, and default JNDI names (based on class and interface names). Alternatively, you can define environment references using either OC4J-specific deployment descriptors or OC4J-proprietary annotations.

In EJB 2.1, you must define `<ejb-ref>` or `<ejb-local-ref>` elements in the appropriate deployment descriptor.

In either case, when you define an environment reference, you can use the actual JNDI name or use a logical name associated with it to increase deployment flexibility.

For more information, see the following:

- "Where do you Configure an EJB Environment Reference?" on page 19-3
- "Should you use Logical Names?" on page 19-3

EJB Environment References

Before one enterprise bean, acting in the role of a client (call it the source enterprise bean), can access another enterprise bean (call it the target enterprise bean), you must define an EJB reference to the target enterprise bean in the deployment descriptor of the source enterprise bean.

> **Note:** In EJB 3.0, an environment reference to a target enterprise bean is not needed. You can access a target enterprise bean directly using resource injection (see "Accessing an EJB 3.0 Enterprise Bean" on page 29-5).

For more information, see the following:

- Configuring an Environment Reference to a Remote EJB: Clustered or Combined Web Tier and EJB Tier
- Configuring an Environment Reference to a Remote EJB: Unclustered Separate Web Tier and EJB Tier
- Configuring an Environment Reference to a Local EJB

Resource Manager Connection Factory Environment References

You can define an environment reference to resource manager connection factories that provide connections to such services as a JDBC data source, JMS topic or queue, Java mail, or an HTTP URL. These references are logical names that OC4J binds at deployment time to the actual resource manager connection factories that it provides.

> **Note:** In EJB 3.0, an environment reference to a resource manager connection factory is not needed. You can access a resource manager connection factory directly using resource injection (see "Looking Up an EJB 3.0 Resource Manager Connection Factory" on page 19-23).

For each client in which you want to access a resource manager connection factory, you must either inject it in the client source code or define an environment reference to it in the client's deployment descriptor.

For more information, see the following:

- "Configuring an Environment Reference to a JDBC Data Source Resource Manager Connection Factory" on page 19-11
- "Configuring an Environment Reference to a JMS Destination Resource Manager Connection Factory (JMS 1.1)"
- "Configuring an Environment Reference to a JMS Destination or Connection Resource Manager Connection Factory (JMS 1.0)" on page 19-14

Environment Variable Environment References

You can define an enviornment variable with an enviornment reference to make the environment variable value accessible using JNDI.

For more information, see "Configuring an Environment Reference to an Environment Variable" on page 19-16

Web Service Environment References

You can define a Web service with an enviornment reference to make the Web service accessible using JNDI

For more information, see "Configuring an Environment Reference to a Web Service" on page 19-17.

Persistence Context References

The preferred way to access an entity manager is using annotations and dependency injection (see "Acquiring the OC4J Default Entity Manager" on page 29-9 and "Acquiring an Entity Manager Using JNDI" on page 29-9).

To acquire an entity manager in a class that does not support annotations and injection, namely helper classes and Web clients, you must first define a persistence context reference and then lookup the entity manager using JNDI.

For more information, see the following:

- "Configuring an Environment Reference to a Persistence Context" on page 19-18
- "Acquiring an Entity Manager in a Helper Class" on page 29-11

Where do you Configure an EJB Environment Reference?

If you choose to use environment references, where you configure the EJB reference depends on the type of client, as Table 19–1 shows.

Table 19–1 Deployment Descriptor by Client Type

Client Type	Description	Deployment Descriptor	OC4J-Specific Deployment Descriptor
EJB	Another enterprise bean invoking an enterprise bean from within the container.	`ejb-jar.xml`	`orion-ejb-jar.xml`
Standalone client	A pure-Java client invoking an enterprise bean from outside of the container.	`application-client.xml`	`orion-application-client.xml`
Servlet or JSP	A servlet or JSP invoking an enterprise bean from outside of the container.	`web.xml`	`orion-web.xml`

In EJB 3.0, if you wish to define an EJB environment reference, you can use OC4J-proprietary annotations as an alternative to OC4J-specific deployment descriptors.

Should you use Logical Names?

When you define an environment reference, you can identify the resource by a logical name or by its JNDI name. To maximize application assembly and deployment flexibility, you typically develop an EJB application by referring to resources by a

logical name that you define in your application environment. This indirection enables the bean developer to refer to enterprise beans, other resources (such as a JDBC `DataSource`), and environment variables without specifying the actual name, which may change depending on how an application is assembled and deployed. The procedures in this chapter explain how to configure either logical or JNDI names.

Configuring an Environment Reference to a Remote EJB: Clustered or Combined Web Tier and EJB Tier

In a clustered OC4J architecture or a single-instance OC4J architecture with both Web tier and EJB tier on the same OC4J instance, you can define an EJB reference to the remote interface of a target enterprise bean using one of the following approaches (in increasing order of assembly and deployment flexibility):

- Configure an `<ejb-ref>` element in the appropriate client EJB deployment descriptor that specifies the actual name of the target bean (see "Configuring ejb-ref in the Client: No Indirection" on page 19-4).
- Configure an `<ejb-ref>` element in the appropriate client EJB deployment descriptor that specifies a logical name and an `<ejb-link>` element that associates this logical name with the actual bean (see "Configuring ejb-ref in the Client: Using ejb-link to Resolve Indirection" on page 19-5).
- Configure an `<ejb-ref>` element in the appropriate client EJB deployment descriptor that specifies a logical name and an `<ejb-ref-mapping>` in the appropriate OC4J-specific deployment descriptor that associates this logical name with the actual bean (see "Configuring ejb-ref in the Client: Using orion-ejb-jar.xml ejb-ref-mapping to Resolve Indirection" on page 19-5).

Note: In EJB 3.0, an environment reference to a target enterprise bean is not needed. You can access a target enterprise bean directly using resource injection (see "Accessing an EJB 3.0 Enterprise Bean" on page 29-5).

For an unclustered architecture, in which the Web tier and EJB tier are deployed to separate OC4J instances on different hosts, see "Configuring an Environment Reference to a Remote EJB: Unclustered Separate Web Tier and EJB Tier" on page 19-6.

For information on looking up a target enterprise bean, see "Accessing an Enterprise Bean From a Client" on page 29-1.

Configuring ejb-ref in the Client: No Indirection

Choose this option if the bean interfaces are unique (for example, only one session bean uses the interface `Cart.class`), or you do not want to use indirection that offers some assembly and deployment flexibility:

1. Define an `<ejb-ref>` element in the appropriate client deployment descriptor (see "Where do you Configure an EJB Environment Reference?" on page 19-3) and configure the following subelements, as Example 19–1 shows:
   - `<ejb-ref-name>`: the actual name of the target enterprise bean.
   - `<ejb-ref-type>`: the type of the target enterprise bean, one of `Session` or `Entity`.

- `<home>`: the package and class name of the target enterprise bean's remote home interface.
- `<remote>`: package and class name of the target enterprise bean's remote component interface.

Example 19–1 Configuring ejb-ref-name

```
<ejb-ref>
 <ejb-ref-name>myBeans/BeanA</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <home>myBeans.BeanAHome</home>
 <remote>myBeans.BeanA</remote>
</ejb-ref>
```

Configuring ejb-ref in the Client: Using ejb-link to Resolve Indirection

Choose this option if the bean interfaces are not unique or, if you want to use indirection that offers some assembly and deployment flexibility:

1. Define an `<ejb-ref>` element in the appropriate client deployment descriptor (see "Where do you Configure an EJB Environment Reference?" on page 19-3) and configure the following subelements, as Example 19–2 shows:
   - `<ejb-ref-name>`: the logical name of the target enterprise bean.
   - `<ejb-ref-type>`: the type of the target enterprise bean, one of `Session` or `Entity`.
   - `<home>`: the package and class name of the target enterprise bean's remote home interface.
   - `<remote>`: package and class name of the target enterprise bean's remote component interface.
   - `<ejb-link>`: the actual name of the target bean.

Example 19–2 Configuring ejb-ref-name with a Logical Name Resolved by ejb-link

```
<ejb-ref>
 <ejb-ref-name>ejb/nextVal</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <home>myBeans.BeanAHome</home>
 <remote>myBeans.BeanA</remote>
 <ejb-link>myBeans/BeanA</ejb-link>
</ejb-ref>
```

Configuring ejb-ref in the Client: Using orion-ejb-jar.xml ejb-ref-mapping to Resolve Indirection

Choose this option, if the following is true:

- The bean interfaces are not unique.
- You want to use indirection that offers the most assembly and deployment flexibility.

1. Define an `<ejb-ref>` element in the appropriate client deployment descriptor (see "Where do you Configure an EJB Environment Reference?" on page 19-3) and configure the following subelements, as Example 19–3 shows:

- `<ejb-ref-name>`: the logical name of the target enterprise bean.
- `<ejb-ref-type>`: the type of the target enterprise bean, one of `Session` or `Entity`.
- `<home>`: the package and class name of the target enterprise bean's remote home interface.
- `<remote>`: package and class name of the target enterprise bean's remote component interface.

Example 19–3 Configuring ejb-ref-name With a Logical Name Resolved by ejb-ref-mapping

```
<ejb-ref>
 <ejb-ref-name>ejb/nextVal</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <home>myBeans.BeanAHome</home>
 <remote>myBeans.BeanA</remote>
</ejb-ref>
```

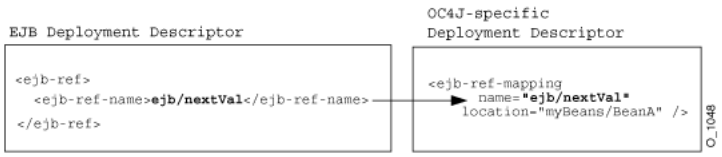
2. Within the `orion-ejb-jar.xml` deployment descriptor, define an `<ejb-ref-mapping>` element that maps the logical name to the actual name of the target bean, as Example 19–4 shows.

Example 19–4 Mapping Logical Name to Actual Name With ejb-ref-mapping

```
<ejb-ref-mapping name="ejb/nextVal" location="myBeans/BeanA"/>
```

As Figure 19–1 shows, in the `<ejb-ref-mapping>` element, configure the `name` attribute to match the `<ejb-ref-name>` and configure the `location` attribute with the actual name of the target bean. In Example 19–4, the logical name `ejb/nextVal` is mapped to the actual name of the target bean `myBeans/BeanA`.

Figure 19–1 Associating ejb-ref-name and ejb-ref-mapping



OC4J maps the logical name to the actual JNDI name on the client side. The server side receives the JNDI name and resolves it within its JNDI tree.

Configuring an Environment Reference to a Remote EJB: Unclustered Separate Web Tier and EJB Tier

A common Java EE application architecture is one in which you deploy the Web tier to one OC4J instance, and the EJB tier–to another OC4J instance on a separate host in a nonclustered environment.

In this architecture, to access a remote enterprise bean, you must populate the required JNDI properties in your Web-tier code when you create the context (for example, see "Setting JNDI Properties in the Initial Context" on page 19-23). These hard-coded properties can cause portability problems, when, for example, migrating from a test environment to a production environment.

Using OC4J-proprietary deployment XML (see "Using Deployment XML" on page 19-7), you can associate a reference to a remote enterprise bean with a JNDI properties file that contains the required JNDI context variables. This simplifies assembly and deployment.

Figure 19–2 shows this architecture for a JSP/Servlet client, and Figure 19–3 shows this architecture for an EJB client.

Figure 19–2 Web-tier and EJB-tier Remote EJB Access: JSP/Servlet Client

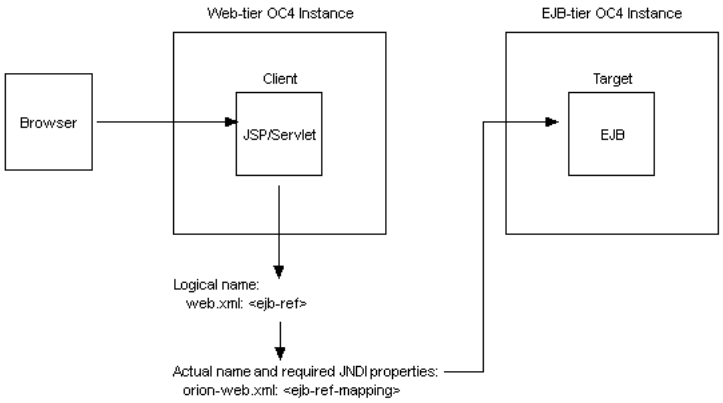
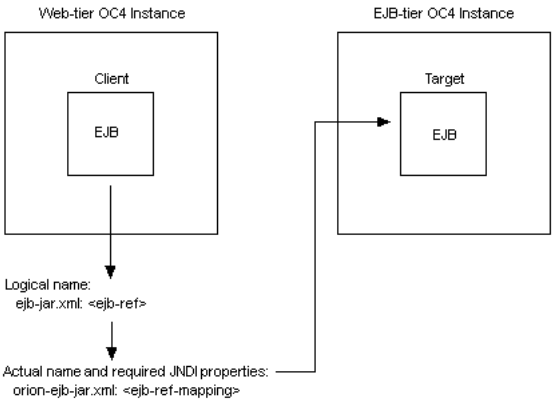


Figure 19–3 Web-tier and EJB-tier Remote EJB Access: EJB Client



For more information about the JNDI properties file, see "Setting JNDI Properties With the JNDI Properties File" on page 19-22).

Using Deployment XML

To associate a reference to a remote enterprise bean with a JNDI properties file that contains the required JNDI context variables using OC4J-proprietary element `<ejb-ref-mapping>`, perform the following configuration on the Web-tier OC4J instance:

1. Define an `<ejb-ref>` element in the appropriate client deployment descriptor (see "Where do you Configure an EJB Environment Reference?" on page 19-3) and configure the following subelements, as Example 19–5 shows:
   - `<ejb-ref-name>`: the logical name of the target enterprise bean.

- <ejb-ref-type>: the type of the target enterprise bean, one of Session or Entity.
- <home>: the package and class name of the target enterprise bean's remote home interface.
- <remote>: package and class name of the target enterprise bean's remote component interface.

Example 19–5 Configuring ejb-ref-name With a Logical Name Resolved by ejb-ref-mapping

```
<ejb-ref>
 <ejb-ref-name>ejb/emp</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <home>myBeans.EmployeeBeanHome</home>
 <remote>myBeans.EmployeeBean</remote>
</ejb-ref>
```

In this architecture, the client deployment descriptor is on the Web-tier OC4J instance. The client of the remote enterprise bean is one of the following:

- a JSP or servlet deployed on the Web tier: use the web.xml file.
- an enterprise bean deployed on the Web tier: use the ejb-jar.xml file.

2. Within the orion-web.xml or orion-ejb-jar.xml deployment descriptor (depending on your type of client), define an <ejb-ref-mapping> element that does the following, as Example 19–6 shows:
   - maps the logical name (ejb/emp) to the actual name (myBeans/EmployeeBean) of the target bean;
   - specifies that target EJB instances are located on a remote host (remote-server-ref="true");
   - associates the reference with a JNDI properties file (jndi-properties-file="empjndi.properties") that contains the JNDI context variables that a client needs to access the remote host, on which target EJB instances are deployed.

Example 19–6 Mapping Logical Name to Actual Name With ejb-ref-mapping for a Remote Target EJB

```
<ejb-ref-mapping
    name="ejb/emp"
    location="myBeans/EmployeeBean"
    remote-server-ref="true"
    jndi-properties-file="empjndi.properties"
/>
```

As Figure 19–1 shows, in the <ejb-ref-mapping> element, you configure the name attribute to match the <ejb-ref-name> and configure the location attribute with the actual name of the target bean. In Example 19–4, the logical name ejb/emp is mapped to the actual name of the target bean myBeans/EmployeeBean.

Figure 19–4 Associating ejb-ref-name and ejb-ref-mapping for a Remote Target EJB

When the Web-tier client (JSP/Servlet or enterprise bean deployed to the Web tier) accesses the remote target enterprise bean (using injection or JNDI lookup), the Web-tier OC4J instance maps the logical name (specified in the Web-tier OC4J instance's `web.xml` or `ejb-jar.xml` file) to the actual name (specified in the Web-tier OC4J instance's `orion-web.xml` or `orion-ejb-jar.xml` file). The Web-tier OC4J instance uses the JNDI properties file specified in the `<ejb-ref-mapping>` element to access the EJB-tier OC4J instance and resolve the actual name to the target enterprise bean on the EJB-tier OC4J instance.

Configuring an Environment Reference to a Local EJB

You can define an EJB reference to the local interface of a target enterprise bean using one of the following approaches (in increasing order of assembly and deployment flexibility):

- Configure an `<ejb-local-ref>` element in the appropriate client EJB deployment descriptor that specifies the actual name of the target bean (see "Configuring ejb-local-ref in the Client: No Indirection" on page 19-9).
- Configure an `<ejb-local-ref>` element in the appropriate client EJB deployment descriptor that specifies a logical name and an `<ejb-link>` element that associates this logical name with the actual bean (see "Configuring ejb-local-ref in the Client: Using ejb-link to Resolve Indirection" on page 19-10).
- Configure an `<ejb-local-ref>` element in the appropriate client EJB deployment descriptor that specifies a logical name and an `<ejb-ref-mapping>` in the appropriate OC4J-specific deployment descriptor that associates this logical name with the actual bean (see "Configuring ejb-local-ref in the Client: Using orion-ejb-jar.xml ejb-ref-mapping to Resolve Indirection" on page 19-10).

> **Note:** In EJB 3.0, an environment reference to a target enterprise bean is not needed. You can access a target enterprise bean directly using resource injection (see "Accessing an EJB 3.0 Enterprise Bean" on page 29-5).

For information on looking up a target enterprise bean, see "Accessing an Enterprise Bean From a Client" on page 29-1.

Configuring ejb-local-ref in the Client: No Indirection

Choose this option if the bean interfaces are unique (for example, only one session bean uses the interface `Cart.class`) or you do not want to use indirection that offers some assembly and deployment flexibility:

1. Define an `<ejb-local-ref>` element in the appropriate client deployment descriptor (see "Where do you Configure an EJB Environment Reference?" on page 19-3) and configure the following subelements, as Example 19–1 shows:
   - `<ejb-ref-name>`: the actual name of the target enterprise bean.
   - `<ejb-ref-type>`: the target bean's type: `Session` or `Entity`.
   - `<local-home>`: the package and class name of the target bean's local home interface.
   - `<local>`: the package and class name of the target bean's local component interface.

Example 19–7 Configuring ejb-local-ref-name

```
<ejb-local-ref>
 <ejb-ref-name>myBeans/BeanA</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <local-home>myBeans.BeanAHome</home>
 <local>myBeans.BeanA</remote>
</ejb-ref>
```

Configuring ejb-local-ref in the Client: Using ejb-link to Resolve Indirection

Choose this option if the bean interfaces are not unique, or if you want to use indirection that offers some assembly and deployment flexibility:

1. Define an `<ejb-local-ref>` element in the appropriate client deployment descriptor (see "Where do you Configure an EJB Environment Reference?" on page 19-3) and configure the following subelements, as Example 19–8 shows:
   - `<ejb-ref-name>`: the logical name of the target enterprise bean.
   - `<ejb-ref-type>`: the target bean's type: `Session` or `Entity`.
   - `<local-home>`: the package and class name of the target bean's local home interface.
   - `<local>`: the package and class name of the target bean's local component interface.
   - `<ejb-link>`: actual name of the target bean

Example 19–8 Configuring ejb-ref-name with a Logical Name Resolved by ejb-link

```
<ejb-local-ref>
 <ejb-ref-name>ejb/nextVal</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <local-home>myBeans.BeanAHome</home>
 <local>myBeans.BeanA</remote>
 <ejb-link>myBeans/BeanA</ejb-link>
</ejb-ref>
```

Configuring ejb-local-ref in the Client: Using orion-ejb-jar.xml ejb-ref-mapping to Resolve Indirection

Choose this option if the following is true:

- The bean interfaces are not unique.

- You want to use indirection that offers the most assembly and deployment flexibility.

1. Define an `<ejb-ref>` element in the appropriate client deployment descriptor (see "Where do you Configure an EJB Environment Reference?" on page 19-3) and configure the following subelements, as Example 19–9 shows:
   - `<ejb-ref-name>`: the logical name of the target enterprise bean.
   - `<ejb-ref-type>`: the target bean's type: `Session` or `Entity`.
   - `<local-home>`: the package and class name of the target bean's local home interface.
   - `<local>`: the package and class name of the target bean's local component interface.

Example 19–9 Configuring ejb-ref-name With a Logical Name Resolved by ejb-ref-mapping

```
<ejb-local-ref>
 <ejb-ref-name>ejb/nextVal</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <local-home>myBeans.BeanAHome</home>
 <local>myBeans.BeanA</remote>
</ejb-ref>
```

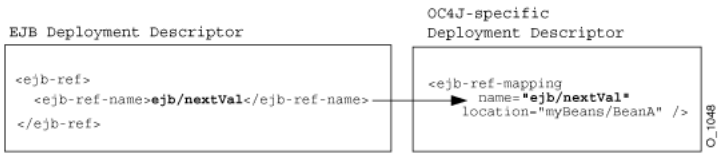
2. Within the `orion-ejb-jar.xml` deployment descriptor, define an `<ejb-ref-mapping>` element that maps the logical name to the actual name of the target bean, as Example 19–10 shows.

Example 19–10 Mapping Logical Name to Actual Name With ejb-ref-mapping

```
<ejb-ref-mapping name="ejb/nextVal" location="myBeans/BeanA"/>
```

As Figure 19–5 shows, in the `<ejb-ref-mapping>` element, configure the `name` attribute to match the `<ejb-ref-name>` and configure the `location` attribute with the actual name of the target bean. In Example 19–10, the logical name `ejb/nextVal` is mapped to the actual name of the target bean `myBeans/BeanA`.

Figure 19–5 Associating ejb-ref-name and ejb-ref-mapping



OC4J maps the logical name to the actual JNDI name on the client side. The server side receives the JNDI name and resolves it within its JNDI tree.

Configuring an Environment Reference to a JDBC Data Source Resource Manager Connection Factory

You can access a database through JDBC by creating an environment element for a JDBC `DataSource` using deployment XML (see "Using Deployment XML" on page 19-12).

> **Note:** In EJB 3.0, an environment reference to a resource manager connection factory is not needed. You can access a resource manager connection factory directly using resource injection (see "Looking Up an EJB 3.0 Resource Manager Connection Factory" on page 19-23).

For information on looking up a resource manager connection factory, see the following:

- "Looking Up an EJB 3.0 Resource Manager Connection Factory" on page 19-23
- "Looking Up an EJB 2.1 Resource Manager Connection Factory" on page 19-25

Using Deployment XML

To define a reference to a JDBC `DataSource` using deployment XML, do the following:

1. In the `data-sources.xml` file, define the desired `DataSource` and specify its actual JNDI name (see "Configuring Data Sources" on page 20-1).

   In this example, assume a `DataSource` is specified in the `data-sources.xml` file with the JNDI name of `/test/OrderDataSource`.

2. Define a `<resource-ref>` element in the appropriate client deployment descriptor (see "Where do you Configure an EJB Environment Reference?" on page 19-3) and configure the following subelements, as Example 19–11 shows:

   - `<res-ref-name>`: the logical name for the JDBC data source.

     It is a best practice to prefix the reference name with `jdbc`, but it is not required. If you use the initial context to look up this reference in your bean source code (see Example 19–30 on page 19-25), always prefix the logical name with `java:comp/env/` (for example, `java:comp/env/jdbc/OrderDB`).

   - `<res-type>`: the Java type of the resource. For the JDBC `DataSource` object, this is `javax.sq.DataSource`.

   - `<res-auth>`: the source of authentication information, either `Application` or `Container`.

Example 19–11 Configuring <resource-ref> in ejb-jar.xml

```
<enterprise-beans>
...
    <resource-ref>
        <res-ref-name>jdbc/OrderDB</res-ref-name>
        <res-type>javax.sq.DataSource</res-type>
        <res-auth>Application</res-auth>
    </resource-ref>

</enterprise-beans>
```

3. In the `orion-ejb-jar.xml` deployment descriptor, define a `<resource-ref-mapping>` and configure the following attributes, as Example 19–12 shows:

   - `name`: the logical name of the data source (defined in `ejb-jar.xml`).

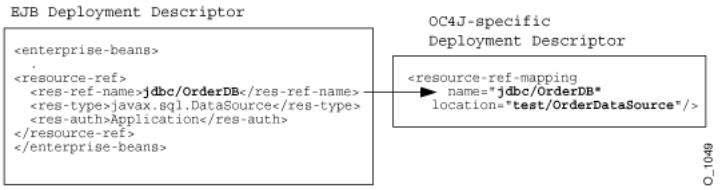
   - `location`: the actual name of the data source (defined in `data-sources.xml`).

Example 19–12 Mapping Logical to Actual JDBC Data Source Resource Manager Connection Factory Using <resource-ref-mapping>

```
<resource-ref-mapping
    name="jdbc/OrderDB"
    location="test/OrderDataSource"
/>
```

Figure 19–6 shows a `<resource-ref-mapping>` element with the `name` attribute set to `jdbc/OrderDB` (the logical name defined in `ejb-jar.xml`) and the `location` attribute set to `test/OrderDataSource` (the JNDI name defined in `data-sources.xml`).

Figure 19–6 Mapping Logical to Actual JDBC Data Source Resource Manager Connection Factory



Within the bean's implementation, you can look up the JDBC data source resource manager connection factory for this data source using the logical name `java:comp/env/jdbc/OrderDB` (see Example 19–30 on page 19-25).

Configuring an Environment Reference to a JMS Destination Resource Manager Connection Factory (JMS 1.1)

Using JMS 1.1, you define an environment reference to a JMS connection resource manager connection factory the same as you do in JMS 1.0 (see "Configuring an Environment Reference to a JMS Destination or Connection Resource Manager Connection Factory (JMS 1.0)" on page 19-14). However, you can define an environment reference to a JMS destination using a `<message-destination-ref>` element in the client deployment descriptor and a `<message-destination-ref-mapping>` element in the corresponding OC4J-specific deployment descriptor (see "Where do you Configure an EJB Environment Reference?" on page 19-3).

You use the `<message-destination-ref-mapping>` to map the client `<message-destination-ref-name>` to another location that is available in the OC4J environment. This provides the means of linking message consumers and producers to one or more common logical destinations.

You can use `<message-destination-ref>` in all EJB types, therefore `<message-destination-ref-mapping>` is not restricted to message-driven deployment.

For more information, see "Oracle Enterprise Messaging Service (OEMS)" in the *Oracle Containers for J2EE Services Guide*.

Note: In EJB 3.0, an environment reference to a resource manager connection factory is not needed. You can access a resource manager connection factory directly using resource injection (see "Looking Up an EJB 3.0 Resource Manager Connection Factory" on page 19-23).

For information on looking up a resource manager connection factory, see the following:

- "Looking Up an EJB 3.0 Resource Manager Connection Factory" on page 19-23
- "Looking Up an EJB 2.1 Resource Manager Connection Factory" on page 19-25

Configuring an Environment Reference to a JMS Destination or Connection Resource Manager Connection Factory (JMS 1.0)

You can access a JMS destination (queue or topic) and JMS connection resource manager connection factory by creating an environment reference to them using deployment XML (see "Using Deployment XML" on page 19-14).

Note: In EJB 3.0, an environment reference to a resource manager connection factory is not needed. You can access a resource manager connection factory directly using annotations and resource injection (see "Looking Up an EJB 3.0 Resource Manager Connection Factory" on page 19-23).

For information on looking up a resource manager connection factory, see the following:

- "Looking Up an EJB 3.0 Resource Manager Connection Factory" on page 19-23
- "Looking Up an EJB 2.1 Resource Manager Connection Factory" on page 19-25

Using Deployment XML

To define a reference to a JMS destination and JMS connection resource manager connection factory, do the following:

1. Configure your JMS service provider.

   For more information, see the following:

   - "Configuring a J2CA Resource Adapter for use With Your Message Service Provider" on page 23-1
   - "Configuring an OEMS JMS Message Service Provider" on page 23-3
   - "Configuring an OEMS JMS Database Message Service Provider" on page 23-5

2. Define the JNDI name for the JMS destination and connection factory.

   For more information, see the following:

   - "J2CA Message Service Provider Connection Factory Names" on page 23-2
   - "OEMS JMS Destination and Connection Factory Names" on page 23-3
   - "OEMS JMS Database Destination and Connection Factory Names" on page 23-6

3. Define a logical name for the JMS destination and JMS connection factory:

   How you define the logical names is the same regardless of what type of JMS provider you use.

   a. Define a `<resource-env-ref>` element in the appropriate client deployment descriptor (see "Where do you Configure an EJB Environment Reference?" on page 19-3) and configure the following subelements:

- `<resource-env-ref-name>`: a logical name for the JMS destination resource manager connection factory.
- `<resource-env-ref-type>`: The destination class type; either `javax.jms.Queue` or `javax.jms.Topic`.

Example 19–13 shows a `<resource-env-ref>` element for a JMS topic resource manager connection factory.

Example 19–13 <resource-env-ref> for a JMS Topic Destination

```
<resource-env-ref>
  <resource-env-ref-name>rpTestTopic</resource-env-ref-name>
  <resource-env-ref-type>javax.jms.Topic</resource-env-ref-type>
</resource-env-ref>
```

b. Define a `<resource-ref>` element in the same client deployment descriptor and configure the following subelements:

- `<res-ref-name>`: a logical name for the JMS connection resource manager connection factory.
- `<res-type>`: the connection factory class type; either `javax.jms.QueueConnectionFactory` or `javax.jms.TopicConnectionFactory`.
- `<res-auth>`: the authentication responsibility; either `Container` or `Bean`.
- `<res-sharing-scope>`: the sharing scope; either `Shareable` or `Unshareable`.

Example 19–14 shows a `<resource-ref>` element for a JMS topic connection resource manager connection factory.

Example 19–14 <resource-ref> for a JMS Topic Connection Factory

```
<resource-ref>
  <res-ref-name>myTCF</res-ref-name>
  <res-type>javax.jms.TopicConnectionFactory</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

4. Map the logical names to the actual JNDI names.

a. Define a `<resource-env-ref-mapping>` element in the corresponding OC4J-specific deployment descriptor (see "Where do you Configure an EJB Environment Reference?" on page 19-3) and configure its `name` attribute to the JMS destination logical name (defined in the `<resource-env-ref>`), and its `location` attribute–to the JNDI name defined when you configured your JMS provider (see step 2).

Example 19–15 shows a `<resource-env-ref-mapping>` element for OEMS JMS.

Example 19–15 OEMS JMS <resource-env-ref-mapping>

```
<resource-env-ref-mapping
    name="rpTestTopic"
    location="jms/Topic/rpTestTopic">
</resource-env-ref-mapping>
```

b. Define a `<resource-ref-mapping>` element in the same OC4J-specific deployment descriptor (see "Where do you Configure an EJB Environment Reference?" on page 19-3) and configure its `name` attribute to the JMS connection factory logical name (defined in the `<resource-ref>`), and its `location` attribute–to the JNDI name defined when you configured your JMS provider (see step 2).

Example 19–16 shows a `<resource-ref-mapping>` element for OEMS JMS.

Example 19–16 OEMS JMS <resource-ref-mapping>

```
<resource-ref-mapping
    name="myTCF"
    location="jms/Topic/myTCF">
</resource-ref-mapping>
```

Configuring an Environment Reference to an Environment Variable

You can create environment variables that your bean accesses through a JNDI lookup on the `InitialContext`. These variables are defined within an `ejb-jar.xml` file `<env-entry>` element and can be of the following types: `String`, `Integer`, `Boolean`, `Double`, `Byte`, `Short`, `Long`, and `Float`. The environment variable name is defined in the `<env-entry-name>` subelement, the type is defined in the `<env-entry-type>` subelement, and the value is defined in the `<env-entry-value>` subelement. The `<env-entry-name>` is relative to the `"java:comp/env"` context.

Example 19–17 shows how to define environment variables for `java:comp/env/minBalance` and `java:comp/env/maxCreditBalance` in the `ejb-jar.xml` file.

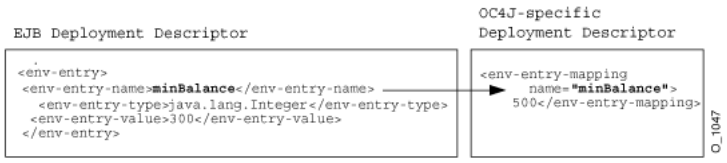
Example 19–17 ejb-jar.xml For Environment Variables

```
<env-entry>
     <env-entry-name>minBalance</env-entry-name>
     <env-entry-type>java.lang.Integer</env-entry-type>
     <env-entry-value>500</env-entry-value>
</env-entry>
<env-entry>
     <env-entry-name>maxCreditBalance</env-entry-name>
     <env-entry-type>java.lang.Integer</env-entry-type>
     <env-entry-value>10000</env-entry-value>
</env-entry>
```

You can override an environment variable value defined in the `ejb-jar.xml` file by defining an `env-entry-mapping` element in your `orion-ejb-jar.xml` file, whose `name` attribute matches the `env-entry-name` defined in the `ejb-jar.xml` file. The type specified in the `ejb-jar.xml` file stays the same.

Figure 19–7 shows how the `minBalance` environment variable value is overridden by the `orion-ejb-jar.xml` file and set to 500.

Figure 19–7 Overriding Environment Variables in ejb-jar.xml with orion-ejb-jar.xml



For more information on looking up environment variables, see the following:

"Looking Up an EJB 3.0 Environment Variable" on page 19-23

"Looking Up an EJB 2.1 Enviornment Variable" on page 19-25

Configuring an Environment Reference to a Web Service

You can access a Web service from a stateless session bean by creating a resource manager connection factory reference to the Web service.

> **Note:** In EJB 3.0, an environment reference to a Web service is not needed. You can access a Web service directly using annotations and resource injection.

For each client, in which you want to access a resource manager connection factory, you must either inject it in the client source code, or define an environment reference to it in the client's deployment descriptor.

To create an environment reference to a Web service, do the following:

1. Define a logical name for the Web service.

   Define a `<service-ref>` element in the appropriate client deployment descriptor (see "Where do you Configure an EJB Environment Reference?" on page 19-3) and configure the following subelements:

   - `<service-ref-name>`: a logical name for the Web service.
   - `<service-interface>`: the Web service interface.

   Example 19–18 shows a `<service-ref>` element for a Web service.

   It is a best practice to start the reference name with `service`, but it is not required. In the bean code, the lookup of this reference (see Example 30–5 on page 30-3) is always prefaced by `java:comp/env` (for example, `java:comp/env/service/myService`).

Example 19–18 ejb-jar.xml For a Web Service Logical Name

```
<service-ref>
    <service-ref-name>service/StockQuoteService</service-ref-name>
    <service-interface>com.example.StockQuoteService</service-interface>
</service-ref>
```

2. Map the logical name to the actual JNDI name.

   Define a `<service-ref-mapping>` element in the corresponding OC4J-specific deployment descriptor (see "Where do you Configure an EJB Environment Reference?" on page 19-3) and configure its `name` attribute to the Web service logical name (defined in the `<service-ref>`) and the `<service-qname>` subelement.

Example 19–15 shows a <service-ref-mapping> element for a Web service.

Example 19–19 orion-ejb-jar.xml For a Web Service Logical to JNDI Mapping

```
<service-ref-mapping name="service/WebServiceBroker">
    <service-qname namespaceURI="urn:WebServiceBroker" localpart="WebServiceBroker"/>
</service-ref-mapping>
```

For information on looking up and using a Web service, see "Using EJB and Web Services" on page 30-1.

Configuring an Environment Reference to a Persistence Context

The simplest way to acquire an entity manager is by using the `@PersistenceContext` annotation (see "Acquiring an EntityManager" on page 29-8).

However, to acquire an entity manager in a class that does not support annotations and injection, namely helper classes, you must first define a `persistence-context-ref` in the appropriate deployment descriptor file.

To create an environment reference to a persistence context, do the following:

1. Define a logical name for the persistence context.

   Define a <persistence-context-ref> element in the appropriate client deployment descriptor (see "Where do you Configure an EJB Environment Reference?" on page 19-3) and configure the following subelements:

   - <persistence-context-ref-name>: a logical name for the persistence context.
   - <persistence-unit-name>: the name of the persistence unit associated with this persistence context.

     You must define a persistence unit of this name in a `persistence.xml` file.

     For more information, see the following:

     - "What is the persistence.xml File?" on page 2-8
     - "Configuring the persistence.xml File" on page 26-3

   Example 19–18 shows a <persistence-context-ref> element for a persistence context in a `web.xml` file.

   It is a best practice to start the reference name with `persistence`, but it is not required. In the bean code, the lookup of this reference (see "Acquiring an Entity Manager in a Helper Class" on page 29-11) is always prefaced by `java:comp/env` (for example, `java:comp/env/persistence/InventoryAppMgr`).

Example 19–20 web.xml For a Persistence Context

```
...
    <servlet>
        <servlet-name>webTierEntryPoint</servlet-name>
        <servlet-class>com.sun.j2ee.blueprints.waf.controller.web.MainServlet</servlet-class>
        <init-param>
            <param-name>default_locale</param-name>
            <param-value>en_US</param-value>
        </init-param>
        <persistence-context-ref>
            <description>
                Persistence context for the inventory management application.
```

```
            </description>
            <persistence-context-ref-name>
                persistence/InventoryAppMgr
            </persistence-context-ref-name>
            <persistence-unit-name>
                InventoryManagement <!-- Defined in persistenc.xml -->
            </persistence-unit-name>
        </persistence-context-ref>
    </servlet>
...
```

For information on looking up and using an entity manager, see "Acquiring an Entity Manager in a Helper Class" on page 29-11.

Configuring the Initial Context Factory

You use an initial context factory to obtain an initial context–a reference to a JNDI namespace. Using the initial context, you can use the JNDI API to look up an enterprise bean, resource manager connection factory, environment variable, or other JNDI-accessible object.

The type of initial context factory you use depends on the type of client in which you are using it, as Table 19–2 shows.

Table 19–2 Client Initial Context Requirements

Client Type	Relationship to Target EJB	Initial Context Factory
Any Client	Client and target enterprise bean are collocated	Default (see "Configuring the Default Initial Context Factory" on page 19-19)
Any Client	Client and target enterprise bean are deployed in the same application	Default (see "Configuring the Default Initial Context Factory" on page 19-19)
Any Client	Target enterprise bean deployed in an application that is designated as the client's parent[1]	Default (see "Configuring the Default Initial Context Factory" on page 19-19)
EJB Client Servlet or JSP Client	Client and target enterprise bean are not collocated, not deployed in the same application, and target EJB application is not client's parent[1].	`oracle.j2ee.rmi.RMIInitialContextFactory` (see "Configuring an Oracle Initial Context Factory" on page 19-20)
Standalone Java Client	Client and target enterprise bean are not collocated, not deployed in the same application, and target EJB application is not client's parent[1].	`oracle.j2ee.naming.ApplicationClientInitialContextFactory` see "Configuring an Oracle Initial Context Factory" on page 19-20)

[1] See the *Oracle Containers for J2EE Developer's Guide* for more information on how to set the parent of an application.

> **Note:** In this release, note the new package names for the RMI and application client initial context factories.

For more information, see the following:

- *Oracle Containers for J2EE Security Guide*
- *Oracle Containers for J2EE Services Guide*.

Configuring the Default Initial Context Factory

A client that is collocated with the target bean (see Table 19–2) automatically accesses the JNDI properties for the node. Thus, accessing the enterprise bean is simple: no JNDI properties are required.

Example 19–21 Configuring the Default Initial Context

```
//Get the Initial Context for the JNDI lookup for a local EJB
InitialContext ic = new InitialContext();
//Retrieve the Home interface using JNDI lookup
Object helloObject = ic.lookup("java:comp/env/ejb/HelloBean");
```

Configuring an Oracle Initial Context Factory

If your client requires an Oracle initial context factory (see Table 19–2), you must set the following JNDI properties:

For more information about setting JNDI properties, see "Setting JNDI Properties in an Enterprise Bean" on page 19-22.

1. Define the `java.naming.factory.initial` property with the Oracle initial context factory appropriate for your client (see Table 19–2).
2. Define the `java.naming.provider.url` property with the naming provider URL appropriate for your OC4J installation:
   - "Configuring the Naming Provider URL for OC4J and Oracle Application Server" on page 19-20
   - "Configuring the Naming Provider URL for OC4J Standalone" on page 19-21
3. Create a `HashTable` and populate it with the required properties using `javax.naming.Context` fields as keys and `String` objects as values, as Example 19–22 shows.

Example 19–22 Specifying Initial Context Factory Properties

```
Hashtable env = new Hashtable();
env.put("java.naming.factory.initial",
        "oracle.j2ee.server.ApplicationClientInitialContextFactory");
env.put("java.naming.provider.url",
        "opmn:ormi://opmnhost:6004:oc4j_inst1/ejbsamples");
```

4. When you instantiate the initial context, pass the `HashTable` into the initial context constructor, as Example 19–23 shows.

Example 19–23 Instantiate the Initial Context Looking Up a JNDI-Accessible Resource

```
Context ic = new InitialContext (env);
```

5. Use the initial context to look up a JNDI-accessible resource:
   - Looking Up an EJB 3.0 Resource Manager Connection Factory on page 19-23
   - Looking Up an EJB 3.0 Environment Variable on page 19-23
   - Looking Up an EJB 2.1 Resource Manager Connection Factory on page 19-25
   - Looking Up an EJB 2.1 Enviornment Variable on page 19-25
   - "Accessing an Enterprise Bean From a Client" on page 29-1

Configuring the Naming Provider URL for OC4J and Oracle Application Server

In an Oracle Application Server install, OPMN manages one or more OC4J instances. In this case the value for `java.naming.provider.url` should be of the format:

```
opmn:ormi://<hostname>:<opmn-request-port>:<oc4j-instance-name>/<application-name>
```

The fields in this provider URL are defined as follows:

- <hostname>: The name of the host, on which the Oracle Application Server is running.
- <opmn-request-port>: In this configuration, you have to use the OPMN request port instead of using the ORMI port. You can find the OPMN request port in the opmn.xml file, as follows:

```
<notification-server>
    <port local="6100" remote="6200" request="6003"/>
    ...
</notification-server>
```

  The default OPMN request port is 6003.

- <oc4j-instance-name>: In this configuration, you may have more than one OC4J process that OPMN uses for load balancing/failover. You use the name of the instance to which you deployed your application.

  The default instance name is home.

For example, if the host name is dpanda-us, request port is 6003, and instances name is home1, then the provider URL would be:

```
opmn:ormi://dpanda-us:6003:home1/ejbsamples
```

For more information, see the following:

- "Setting JNDI Properties for RMI" in the *Oracle Containers for J2EE Services Guide*
- "Configuring Static Retrieval Load Balancing" on page 24-3
- "Configuring DNS Load Balancing" on page 24-3

Configuring the Naming Provider URL for OC4J Standalone

In a standalone OC4J install, the value for java.naming.provider.url should be of the format:

```
ormi://<hostname>:<ormi-port>/<application-name>
```

The fields in this provider URL are defined as follows:

- <hostname>: The name of the host on which OC4J is running
- <ormi-port>: The ORMI port as configured in the rmi.xml file, as follows:

```
<rmi-server
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/rmi-
server-10_0.xsd"
    port="23791"
    schema-major-version="10"
    schema-minor-version="0"
>
...
</rmi-server>
```

  The default port is 23791.

- <application-name>: The application name as configured in the server.xml file.

For example, if the host name is dpanda-us, ORMI port is 23793, and the application name is ejb30slsb, then the provider URL would be:

```
ormi://dpanda-us:23793/ejb30slsb
```

For more information, see the following:

- "Setting JNDI Properties for RMI" in the *Oracle Containers for J2EE Services Guide*
- "Configuring Static Retrieval Load Balancing" on page 24-3
- "Configuring DNS Load Balancing" on page 24-3

Setting JNDI Properties in an Enterprise Bean

If the client is collocated with the target, the client exists within the same application as the target, or the target exists within its parent, then you do not need to initialize JNDI properties. Otherwise, you must initialize JNDI properties in one of the following ways:

This section describes the following:

- Setting JNDI Properties With the JNDI Properties File
- Setting JNDI Properties With System Properties
- Setting JNDI Properties in the Initial Context

For more information, see the following:

- "Specifying Credentials in EJB Clients" on page 22-10
- *Oracle Containers for J2EE Services Guide*

Setting JNDI Properties With the JNDI Properties File

You can set JNDI properties in a file named `jndi.properties` that conforms to the requirements specified in the `java.util.Properties` method `load`.

Set JNDI properties as follows:

```
<PropertyName>=<PropertyValue>
```

For example:

```
java.naming.factory.initial= oracle.j2ee.server.ApplicationClientInitialContextFactory
```

For property names, see the field definitions in `javax.naming.Context`.

For an example, see "Specifying Credentials in JNDI Properties" on page 22-11.

If setting the JNDI properties within the `jndi.properties` file, make sure that this file is accessible from the client `CLASSPATH`, or specified in `ejb-ref-mapping` attribute `jndi-properties-file` in the appropriate the OC4J-proprietary deployment XML file (see "Configuring an Environment Reference to a Remote EJB: Unclustered Separate Web Tier and EJB Tier" on page 19-6).

Setting JNDI Properties With System Properties

You can set JNDI properties as system properties specified either on the command line as a `-D` argument or as an environment reference (see "Configuring an Environment Reference to an Environment Variable" on page 19-16).

Setting JNDI Properties in the Initial Context

You can set JNDI properties by creating a `HashTable` and populating it with the required properties using `javax.naming.Context` fields as keys and `String` objects as values. When you instantiate the initial context, pass the `HashTable` into the the initial context constructor.

For an example, see "Specifying Credentials in the Initial Context" on page 22-11.

Looking Up an EJB 3.0 Resource Manager Connection Factory

Using EJB 3.0, you can look up a resource manage connection using resource injection (see "Using Annotations" on page 19-23) or the `InitialContext` (see "Using Initial Context" on page 19-23).

Using Annotations

Example 19–24 shows how to use annotations and dependency injection to access an EJB 3.0 resource manager connection factory.

Example 19–24 Injecting an EJB 3.0 Resource Manager Connection Factory

```
@Stateless public class EmployeeServiceBean implements EmployeeService {
    ...
    public void sendEmail(String emailAddress) {
        @Resource Session testMailSession;
        ...
    }
}
```

Using Initial Context

Example 19–25 shows how to use the initial context to look up an EJB 3.0 resource manager connection factory.

Example 19–25 Looking Up an EJB 3.0 Resource Manager Connection Factory

```
@Stateless public class EmployeeServiceBean implements EmployeeService {
    ...
    public void sendEmail(String emailAddress) {
        InitialContext ic = new InitialContext();
        Session session = (Session) ic.lookup("java:comp/env/mail/testMailSession");
        ...
    }
}
```

For more information, see "Configuring the Initial Context Factory" on page 19-19.

Looking Up an EJB 3.0 Environment Variable

Using EJB 3.0, you can look up an environment variable using resource injection (see "Using Resource Injection" on page 19-23) or the `InitialContext` (see "Using Initial Context" on page 19-25).

Using Resource Injection

Using resource injection, you can rely on the container to initialize a field or a setter method (property) using either of the following:

- default JNDI name (of the form `java:comp/env/<FieldOrPropertyName>`)
- explicit JNDI name that you specify (do not prefix the name with "`java:comp/env`")

You cannot inject both field and setter using the same JNDI name.

The following examples show how to initialize the `maxExemptions` field with the value specified for the environment variable with the default JNDI name `java:comp/env/maxExemptions`.

You can use resource injection at the field level (see Example 19–26) or the setter method (property) level, as Example 19–27 shows.

Example 19–26 Resource Injection at Field Level with Default Environment Variable Name

```
@Stateless public class EmployeeServiceBean implements EmployeeService {
    ...
    // The maximum number of tax exemptions, configured by Deployer
    // Assumes JNDI name java:comp/env/maxExemptions.
    @Resource int maxExemptions;
    ...
    public void setMaxExemptions(int maxEx) {
        maxExemptions = maxEx;
    }
    ...
}
```

Example 19–27 Resource Injection at the Property Level with a Default Environment Variable Name

```
@Stateless public class EmployeeServiceBean implements EmployeeService {
    ...
    int maxExemptions;
    ...
    // Assumes JNDI name java:comp/env/maxExemptions.
    @Resource
    public void setMaxExemptions(int maxEx) {
        maxExemptions = maxEx;
    }
    ...
}
```

You can specify an explicit JNDI name, as Example 19–28 shows.

Example 19–28 Resource Injection with a Specific Environment Variable Name

```
@Stateless public class EmployeeServiceBean implements EmployeeService {
    ...
    int maxExemptions;
    ...
    @Resource(name="ApplicationDefaults/maxExemptions")
    public void setMaxExemptions(int maxEx) {
        maxExemptions = maxEx;
    }
    ...
}
```

Using Initial Context

Example 19–29 shows how you look up these environment variables within the bean's code using the `InitialContext`.

Example 19–29 Looking Up Environment Variables

```
InitialContext ic = new InitialContext();
Integer min = (Integer) ic.lookup("java:comp/env/minBalance");
Integer max = (Integer) ic.lookup("java:comp/env/maxCreditBalance"));
```

Notice that to retrieve the values of the environment variables, you prefix each environment element with "`java:comp/env/`", which is the location that the container stored the environment variable.

For more information, see "Configuring the Initial Context Factory" on page 19-19.

Looking Up an EJB 2.1 Resource Manager Connection Factory

Using EJB 2.1, you can look up a resource manager connection factory using the `InitialContext` (see "Using Initial Context" on page 19-25).

For more information on configuring resources, see "Resource Manager Connection Factory Environment References" on page 19-2.

Using Initial Context

Example 19–30 shows how to look up a JDBC data source resource manager connection factory within the bean's code using the `InitialContext` with the logical name defined in the EJB deployment descriptor (see "Configuring an Environment Reference to a JDBC Data Source Resource Manager Connection Factory" on page 19-11) prefixed with `java:comp/env/jdbc`.

Example 19–30 Looking Up a JDBC Data Source Resource Manager Connection Factory

```
javax.sql.DataSource db;
java.sql.Connection conn;
...
InitialContext ic = new InitialContext();
db = (javax.sql.DataSource) initCtx.lookup("java:comp/env/jdbc/OrderDB");
conn = db.getConnection();
```

For more information, see "Configuring the Initial Context Factory" on page 19-19.

Looking Up an EJB 2.1 Enviornment Variable

Using EJB 2.1, you can look up an environment variable using the `InitialContext` (see "Using Initial Context" on page 19-25).

For more information on configuring enviornment variables, see "Configuring an Environment Reference to an Environment Variable" on page 19-16.

Using Initial Context

Example 19–29 shows how you look up these environment variables within the bean's code using the `InitialContext`.

Example 19–31 Looking Up Environment Variables

```
InitialContext ic = new InitialContext();
Integer min = (Integer) ic.lookup("java:comp/env/minBalance");
Integer max = (Integer) ic.lookup("java:comp/env/maxCreditBalance"));
```

Notice that to retrieve the values of the environment variables, you prefix each environment element with `java:comp/env/`, which is the location that the container stored the environment variable.

For more information, see "Configuring the Initial Context Factory" on page 19-19.

20

Configuring Data Sources

This chapter describes the following:

- Configuring a Data Source for an Oracle Database
- Configuring a Data Source for a Third-Party Database
- Configuring a Default Data Source for an EJB 3.0 Application
- Configuring a Default Data Source for an EJB 2.1 Application
- Associating TopLink With an Oracle JDBC Driver

For more information, see the following:

- "Understanding EJB Data Source Services" on page 2-14
- "Specifying a Data Source in a Persistence Unit" on page 26-5
- "Data Sources" in the *Oracle Containers for J2EE Services Guide*

> **Note:** You can download a data source code example from `http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`.

Configuring a Data Source for an Oracle Database

To create a data source for an Oracle database, you create a managed datasource. You can create a managed data source using the Application Server Control Console (see "Using Application Server Control Console" on page 20-1) or deployment XML (see "Using Deployment XML" on page 20-2).

For more information, see the following:

- "What Types of Data Source Does OC4J Support?" on page 2-14
- "Data Sources" in the *Oracle Containers for J2EE Services Guide*

Using Application Server Control Console

You can use Application Server Control Console to create a managed data source dynamically without restarting OC4J.

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

Using Deployment XML

You can configure a managed data source for an Oracle database by configuring a `connection-pool` element and `managed-data-source` element in the `data-sources.xml` file, as Example 20–1 shows.

Example 20–1 data-sources.xml For an Oracle JDBC Data Source

```
<connection-pool name="ScottConnectionPool">
    <connection-factory
        factory-class="oracle.jdbc.pool.OracleDataSource"
        user="scott"
        password="tiger"
        url="jdbc:oracle:thin:@//localhost:1521/ORCL" >
    </connection-factory>
</connection-pool>

<managed-data-source
    name="OracleManagedDS"
    jndi-name="jdbc/OracleDS"
    connection-pool-name="ScottConnectionPool"
    tx-level="global"
/>
```

Be sure to specify a service-based connection URL in the `connection-factory` element (see "How do you Define a Connection URL in OC4J?" on page 2-15).

By default, a managed data source supports global (two-phase commit) transactions. To configure a managed data source to support only local transactions, set the `managed-data-source` attribute `tx-level` to `local`. For more information, see "What Transaction Types do Data Sources Support?" on page 2-16).

For more information, see the following:

- `http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`
- `http://www.oracle.com/technology/tech/java/newsletter/articles/oc4j_datasource_config.html`

If you configure a managed data source using this method, you must restart OC4J to apply your changes. Alternatively, you can use Application Server Control Console to create a data source dynamically without restarting OC4J (see Using Application Server Control Console on page 20-1)

Configuring a Data Source for a Third-Party Database

To create a data source for a third-party (non-Oracle) database, you create a native datasource. You can create a native data source using the Application Server Control Console (see "Using Application Server Control Console" on page 20-1) or deployment XML (see "Using Deployment XML" on page 20-2).

For more information, see the following:

- "What Types of Data Source Does OC4J Support?" on page 2-14
- "Data Sources" in the *Oracle Containers for J2EE Services Guide*

Using Application Server Control Console

You can use Application Server Control Console to create a native data source dynamically without restarting OC4J.

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

Using Deployment XML

Example 20–2 shows how to define a native data source element for a third-party database (in this example, SQLServer).

Example 20–2 data-sources.xml for a Third-Party Database

```
<native-data-source
  name="nativeDataSource"
  jndi-name="jdbc/nativeDS"
  description="Native DataSource"
  data-source-class="com.ddtek.jdbcx.sqlserver.SQLServerDataSource"
  user="frank"
  password="frankpw"
  url="jdbc:datadirect:sqlserver://server_name:1433;User=usr;Password=pwd">
</native-data-source>
```

By default, a native data source supports only local transactions. For global (two-phase commit) transactions, configure a managed data source. For more information, see "What Transaction Types do Data Sources Support?" on page 2-16).

For more information, see the following:

- http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html
- http://www.oracle.com/technology/tech/java/newsletter/articles/oc4j_datasource_config.html

If you configure a native data source using this method, you must restart OC4J to apply your changes. Alternatively, you can use Application Server Control Console to create a native data source dynamically without restarting OC4J (see "Using Application Server Control Console" on page 20-2)

Configuring a Default Data Source for an EJB 3.0 Application

You can configure a default data source for an EJB 3.0 application using deployment XML (see "Using Deployment XML" on page 20-3).

For more information, see the following:

- "What is a Default Data Source?" on page 2-16
- "Data Sources" in the *Oracle Containers for J2EE Services Guide*

Using Deployment XML

To configure a default data source for an EJB 3.0 application, do the following:

1. Set the name of the default data source in the `default-data-source` attribute of your `orion-application.xml` file.
2. Customize your EJB 3.0 application to define a data source of this name in your `ejb3-toplink-session.xml` file.

   For more information, see the following:

   - "What is the ejb3-toplink-sessions.xml File?" on page 2-7

- "Customizing the JPA Persistence Provider" on page 3-3

Configuring a Default Data Source for an EJB 2.1 Application

You can configure a default data source for an EJB 2.1 application using deployment XML (see "Using Deployment XML" on page 20-4).

For more information, see the following:

- "What is a Default Data Source?" on page 2-16
- "Data Sources" in the *Oracle Containers for J2EE Services Guide*

Using Deployment XML

To configure a default data source for an EJB 2.1 application, do the following:

1. Set the name of the default data source in the `default-data-source` attribute of the `orion-application` element in your `orion-application.xml` file.
2. Set the name of the default data source in the `data-source` attribute of the `entity-deployment` element in your `orion-ejb-jar.xml` file.
3. Define the default data source in the `<`*`OC4J_HOME`*`>/j2ee/home/config/data-sources.xml` file.

Associating TopLink With an Oracle JDBC Driver

In this release, by default, TopLink is associated with Oracle JDBC driver version 10.2 (`ojdbc14_102.jar`).

If this Oracle JDBC driver version is not appropriate for the Oracle Database version you need to use, you can associate TopLink with another Oracle JDBC driver version.

How you associate TopLink with another version of Oracle JDBC driver depends on the type of application you are building:

- EJB 3.0 and EJB 2.1 non-CMP Applications
- EJB 2.1 CMP Applications
- EIS AQ Connector Applications

For more information, see "Utilizing the OC4J Class Loading Framework" in the *Oracle Containers for J2EE Developer's Guide*.

EJB 3.0 and EJB 2.1 non-CMP Applications

For EJB 3.0 and EJB 2.1 non-CMP applications, note the following restrictions:

- You may use multiple versions of Oracle JDBC driver simultaneously on the server, but each application may use only a single version.
- For each `oracle.jdbc` shared library version, you must define a corresponding version of `oracle.toplink` shared library in `server.xml`.
- You may use only `Oc4jPlatform`; you cannot use `Oc4jPlatform_10_1_3`.
- If the version of imported shared library is not provided, then the one with the highest version is imported. For example, if `oracle.jdbc` version 10.2 defined in `server.xml` and `system-application.xml` imports `oracle.jdbc` without version, then `oracle.toplink` will use `oracle.jdbc` version 10.2.

To associate an EJB 3.0 or EJB 2.1 non-CMP application with a specific version of Oracle JDBC driver other than the default, do the following:

1. Create a folder in `<ORACLE_HOME>/j2ee/home/shared-lib/oracle.jdbc` for the new Oracle JDBC driver shared library.

   In this example, you would create folder `<ORACLE_HOME>/j2ee/home/shared-lib/oracle.jdbc/10.3`.

   When you reference the actual Oracle JDBC driver JAR file, you do so relative to this directory. You can either put the Oracle JDBC driver JAR file in this directory and simply reference the JAR file by name, or put it in some other directory and reference the JAR file with a partial path relative to this directory.

2. Define the new Oracle JDBC driver shared library in `server.xml`, as Example 20–3 shows.

Example 20–3 Defining a Shared Library for Oracle JDBC Driver Version 10.3 in server.xml

```
...
<shared-library name="oracle.jdbc" version="10.3">
    <code-source path="ojdbc14_103.jar"/>
</shared-library>
...
```

   Use the `oracle.jdbc` shared library name with a different version number that corresponds to the version of Oracle JDBC driver you want to use: in this example, 10.3.

   In this example, the `code-source` attribute `path` is just `ojdbc14_103.jar`: this assumes that you put the JAR file in `<ORACLE_HOME>/j2ee/home/shared-lib/oracle.jdbc/10.3`. Alternatively, you could set `path` to a partial path relative to the `<ORACLE_HOME>/j2ee/home/shared-lib/oracle.jdbc/10.3` directory.

3. Define a corresponding TopLink shared library in `server.xml`, as Example 20–4 shows.

Example 20–4 Defining a Corresponding oracle.toplink Shared Library for Oracle JDBC Driver Version 10.3 in server.xml

```
...
<shared-library name="oracle.jdbc" version="10.3">
    <code-source path="ojdbc14_103.jar"/>
</shared-library>
<shared-library name="oracle.toplink" version="10.3" library-compatible="true">
    <code-source path="../../../../../toplink/jlib/toplink.jar"/>
    <code-source path="../../../../../toplink/jlib/antlr.jar"/>
    <code-source path="../../../../../toplink/jlib/cciblackbox-tx.jar"/>
    <import-shared-library name="oc4j.internal"/>
    <import-shared-library name="oracle.xml"/>
    <import-shared-library name="oracle.jdbc" max-version="10.3"/>
    <import-shared-library name="oracle.dms"/>
</shared-library>
...
```

   Use the `oracle.toplink` shared library name with a different version number that corresponds to the version of Oracle JDBC driver you want to use: in this example, 10.3. In this `oracle.toplink` shared library, be sure to import the desired version of `oracle.jdbc` shared library: in this example, `max-version="10.3"`.

> **Note:** If your new `oracle.toplink` library uses the same JAR files as the original and you created it simply to specify another version of `oracle.jdbc`, then to avoid having to create a corresponding folder under `<ORACLE_HOME>/j2ee/home/shared-lib` for this library, set its `shared-library` attribute `library-compatible` to `true`, as Example 20–4 shows.

4. Import your new shared libraries for your application by doing the following:

   a. To make the new `oracle.jdbc` and `oracle.toplink` shared libraries the default for all applications in your OC4J instance, update the `system-applications.xml`, as Example 20–5 shows.

Example 20–5 Importing the Shared Libraries for all Applications in system-applications.xml

```
...
    <imported-shared-libraries>
...
        <import-shared-library name="oracle.jdbc" min-version="10.3" max-version="10.3"/>
        <import-shared-library name="oracle.toplink" min-version="10.3" max-version="10.3"/>
...
    </imported-shared-libraries>
...
```

   b. To make the new `oracle.jdbc` and `oracle.toplink` shared libraries applicable to only a certain application, update that application's `orion-applications.xml`, as Example 20–6 shows.

      In this case, you should define datasources in a `data-sources.xml` file located in the same folder as the `orion-applications.xml` file and referenced in the `orion-applications.xml` file as Example 20–6 shows.

Example 20–6 Importing the Shared Libraries for a Specific Application in orion-applications.xml

```
...
<orion-application>
    <ejb-module remote="true" path="simpleobject_ejb.jar" />
    <client-module path="simpleobject_ejb.jar" auto-start="false" />
    <persistence path="persistence" />
    <imported-shared-libraries>
        <import-shared-library name="oracle.jdbc" max-version="10.3"/>
        <import-shared-library name="oracle.toplink" max-version="10.3"/>
    </imported-shared-libraries>
    <log>
        <file path="application.log" />
    </log>
    <data-sources path="data-sources.xml" />
    <namespace-access>
        ..........
    </namespace-access>
</orion-application>
```

EJB 2.1 CMP Applications

For EJB 2.1 CMP applications, note the following restrictions:

- You cannot use more than one Oracle JDBC driver version with TopLink.

- If the version of imported shared library is not provided, then the original version is used. For example, if `oracle.jdbc` version 10.2 is defined in `server.xml` and `system-application.xml` imports `oracle.jdbc` without version, then `oracle.toplink` uses original `oracle.jdbc` version 10.1.
- In this case, the TopLink run-time may use any `oracle.toplink.platform.server.oc4j` platform instance. For more information, see "Customizing the TopLink EJB 2.1 Persistence Manager" on page 3-13.

EIS AQ Connector Applications

To associate an application that uses the EIS AQ connector with a specific version of Oracle JDBC driver other than the default, follow the procedure for "EJB 3.0 and EJB 2.1 non-CMP Applications" on page 20-4.

In this case, in your new `oracle.jdbc` shared library, you must also reload the `aqapi.jar` file, as Example 20–7 shows.

Example 20–7 Defining a Shared Library for Oracle JDBC Driver Version 10.3 in server.xml

```
...
<shared-library name="oracle.jdbc" version="10.3">
    <code-source path="ojdbc14_103.jar"/>
    <code-source path="../../../../../rdbms/jlib/aqapi.jar"/>
</shared-library>
...
```

21

Configuring Transaction Services

This chapter describes the following:

- Configuring EJB 3.0 Transaction Management
- Configuring an EJB 3.0 Transaction Attribute
- Configuring EJB 2.1 Transaction Management
- Configuring an EJB 2.1 Transaction Attribute
- Configuring Transaction Timeouts
- Transaction Best Practices

For more information, see the following:

- "Understanding EJB Transaction Services" on page 2-17
- "Java Transaction API (JTA)" in the *Oracle Containers for J2EE Services Guide*

Configuring EJB 3.0 Transaction Management

To configure EJB 3.0 EJB transaction management, you can use annotations (see "Using Annotations" on page 21-1) or deployment XML (see "Using Deployment XML" on page 21-2).

> **Note:** EJB 3.0 entities cannot be configured with a transaction management type. EJB 3.0 entities execute within the transactional context of the caller.

For more information, see the following:

- "Who Manages a Transaction?" on page 2-17
- "What are Container-Managed Transactions?" on page 2-18
- "What are Bean-Managed Transactions?" on page 2-18

Using Annotations

You can configure transaction management using the `@TransactionManagement` annotation attribute `value`, as Example 21–1 shows. You can specify one of the following values:

- `TransactionManagementType.CONTAINER`: container-managed transactions (default).

- `TransactionManagementType.BEAN`: bean-managed transactions.

You apply the `@TransactionManagement` annotation at the class level.

Example 21–1 Configuring Transaction Management for an EJB 3.0 Session Bean

```
import javax.ejb.Stateful
import javax.annotation.PostConstruct;
import javax.ejb.Remove;
import javax.ejb.TransactionManagement;
import javax.ejb.TransactionManagementType;

@Stateful
@TransactionManagement(value=TransactionManagementType.CONTAINER)
public class CartBean implements Cart {
    private ArrayList items;

    @PostConstruct
    public void initialize() {
        items = new ArrayList();
    }

    @Remove
    public void finishedShipping() {
        // Release any resources.
    }

    public void addItem(String item) {
        items.add(item);
    }

    public void removeItem(String item) {
        items.remove(item);
    }
}
```

Using Deployment XML

For an EJB 3.0 EJB, you configure transaction management in the `ejb-jar.xml` file as you would for an EJB 2.1 enterprise bean (see "Using Deployment XML" on page 21-4).

Configuring an EJB 3.0 Transaction Attribute

To configure how the container manages transactions when a client invokes a method of an EJB 3.0 enterprise bean configured for container-managed transactions, you can use annotations (see Using Annotations on page 21-2) or deployment XML (see "Using Deployment XML" on page 21-4).

For more information, see the following:

- "Who Manages a Transaction?" on page 2-17
- "What are Container-Managed Transactions?" on page 2-18
- "How are Transactions Handled When a Client Invokes a Business Method?" on page 2-19.

Using Annotations

You can configure transaction management using the `@TransactionAttribute` annotation attribute `value`, as Example 21–2 shows. Table 21–1 lists the `TransactionAttributeType` values you can specify and shows how the container

will respond depending on whether or not a client-controlled transaction exists at the time the method is invoked.

Table 21–1 TransactionAttributeType Values for @TransactionAttribute

Transaction Attribute	Client-Controlled Transaction Exists	Client-Controlled Transaction Does Not Exist
NOT_SUPPORTED	Container suspends client transaction	Use no transaction
SUPPORTS	Use client-controlled transaction	Use no transaction
REQUIRED[1]	Use client-controlled transaction	Container starts a new transaction
REQUIRES_NEW	Use client-controlled transaction	Container starts a new transaction
MANDATORY	Use client-controlled transaction	Exception raised
NEVER	Exception raised	Use no transaction

[1] Default.

You can apply the `@TransactionAttribute` annotation at the class-level to specify the default transaction attribute for all business methods of the enterprise bean. You can apply this annotation at the method-level to specify the transaction attribute for that method. Applying the annotation at the method-level overrides the class-level annotation (if any) for that method.

Example 21–2 Configuring Transaction Attribute for an EJB 3.0 Session Bean

```
import javax.ejb.Stateful;
import javax.annotation.PostConstruct;
import javax.ejb.Remove;
import javax.ejb.TransactionManagement;
import javax.ejb.TransactionManagementType;
import javax.ejb.TransactionAttribute;
import static javax.ejb.TransactionAttributeType.REQUIRED;
import static javax.ejb.TransactionAttributeType.REQUIRES_NEW;
import com.acme.Cart;

@Stateful
@TransactionManagement(value=TransactionManagementType.CONTAINER)
@TransactionAttribute(value=REQUIRED)
public class CartBean implements Cart {
    private ArrayList items;

    @PostConstruct
    public void initialize() {
        items = new ArrayList();
    }

    @Remove
    @TransactionAttribute(value=REQUIRES_NEW)
    public void finishedShipping() {
        // Release any resources.
    }

    public void addItem(String item) {
        items.add(item);
    }

    public void removeItem(String item) {
        items.remove(item);
    }
}
```

Using Deployment XML

For an EJB 3.0 enterprise bean, you configure transaction attributes in the `orion-ejb-jar.xml` file as you would for an EJB 2.1 enterprise bean (see "Using Deployment XML" on page 21-5).

Configuring EJB 2.1 Transaction Management

You can configure who is responsible for managing transactions that involve a given EJB 2.1 enterprise bean (see "Using Deployment XML" on page 21-4).

> **Note:** EJB 2.1 entity beans must always use container-managed transaction demarcation. An EJB 2.1 entity bean must not be designated with bean-managed transaction demarcation.

For more information, see the following:

- "Who Manages a Transaction?" on page 2-17
- "What are Container-Managed Transactions?" on page 2-18
- "What are Bean-Managed Transactions?" on page 2-18

Using Deployment XML

To configure transaction management, use the `ejb-jar.xml` file `<transaction-type>` subelement, as Example 21–3 shows.

Valid values are `Container` or `Bean`. The default is `Container`.

Example 21–3 Configuring Transaction Management for an EJB 2.1 Session Bean

```
<enterprise-beans>
    <session>
        <display-name>A Credit-Service Bean</display-name>
        <ejb-name>CreditService</ejb-name>
        <home>creditService.ejb.CreditServiceHome</home>
        <remote>creditService.ejb.CreditServiceRemote</remote>
        <ejb-class>creditService.ejb.CreditServiceBean</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Container</transaction-type>
        ...
    </session>
...
</enterprise-beans>
```

You can configure `<transaction-type>` for all of session, entity, and message-driven beans. However, for an EJB 2.1 entity bean, you can only configure `<transaction-type>` as `Container`.

Configuring an EJB 2.1 Transaction Attribute

For an enterprise bean that uses container-managed transactions, you can configure how the container manages transactions when a client invokes a bean method (see "Using Deployment XML" on page 21-5).

For more information, see the following:

- "Who Manages a Transaction?" on page 2-17

- "What are Container-Managed Transactions?" on page 2-18
- "How are Transactions Handled When a Client Invokes a Business Method?" on page 2-19.

Using Deployment XML

To configure how the container manages transactions when a client invokes a bean method, use the `ejb-jar.xml` file `<assembly-descriptor>` subelement `<container-transaction>`, as Example 21–4 shows.

Example 21–4 Configuring Transaction Attribute for an EJB 2.1 Session Bean

```
<assembly-descriptor>
    <container-transaction>
        <method>
            <ejb-name>CreditService</ejb-name>
            <method-name>setLimit</method-name>
            <method-params>
                <method-param>int</method-param>
            </method-params>
        </method>
        <trans-attribute>Required</trans-attribute>
    </container-transaction>
...
</assembly-descriptor>
```

The `<container-transaction>` element contains one or more `<method>` elements and one `<trans-attribute>` element. The `<trans-attribute>` element applies to all the `<method>` elements. You can specify methods by name, by name and parameters (signature), or you can specify all methods of the specified enterprise bean using a wildcard `<method-name>*</method-name>`.

Table 21–2 lists the values for the `<trans-attribute>` element that you can specify and shows how the container will respond depending on whether or not a client-controlled transaction exists at the time the method is invoked

Table 21–2 Valid Values for the <trans-attribute> Element

Transaction Attribute	Client-Controlled Transaction Exists	Client-Controlled Transaction Does Not Exist
`NotSupported`[1]	Container suspends client transaction	Use no transaction
`Supports`[2]	Use client-controlled transaction	Use no transaction
`Required`[3]	Use client-controlled transaction	Container starts a new transaction
`RequiresNew`	Container suspends client transaction and creates a new transaction	Container starts a new transaction
`Mandatory`	Use client-controlled transaction	Exception raised
`Never`	Exception raised	Use no transaction

[1] Default for EJB 2.1 message-driven beans.
[2] Default for EJB 2.1 session beans and BMP entity beans.
[3] Default for EJB 2.1 CMP entity beans.

Configuring Transaction Timeouts

To improve application performance, you can configure a transaction timeout that determines how long OC4J will wait for a transaction to commit or rollback.

This section describes the following:

- Configuring a Global Transaction Timeout

- Configuring a Transaction Timeout for a Session Bean
- Configuring a Transaction Timeout for a Message-Driven Bean

Configuring a Global Transaction Timeout

You can set a transaction timeout that applies globally to all transactions that OC4J manages for session and entity beans.

You can configure the global transaction timeout as follows:

- Using Application Server Control Console
- Using Deployment XML

Using Application Server Control Console

Using the Application Server Control Console (see "Using Oracle Enterprise Manager 10g Application Server Control" on page 31-1), you can set the `JTAResource` MBean attribute `transactionTimeout`.

For more information, see "How to configure the OC4J Transaction Manager" in the *Oracle Containers for J2EE Services Guide*.

Using Deployment XML

In the *<OC4J_HOME>*`\j2ee\home\config\transaction-manager.xml` file you set the global transaction timeout with the `transaction-timeout` attribute of the `<transaction-manager>` element.

For example, if you wanted to set the global transaction timeout to 180 seconds, you would do as follows:

```
<transaction-manager ...  transaction-timeout="180"
  ...
</transaction-manager>
```

If you change this property using this method, you must restart OC4J to apply your changes. Alternatively, you can use Application Server Control Console to modify this parameter dynamically without restarting OC4J (see "Using Application Server Control Console" on page 21-6).

Configuring a Transaction Timeout for a Session Bean

You can specify a transaction timeout for each session bean using OC4J-proprietary annotations (see "Using Annotations" on page 21-6), or using the `orion-ejb-jar.xml` file (see "Using Deployment XML" on page 21-7).

The session bean transaction timeout overrides the global transaction timeout (see "Configuring a Global Transaction Timeout" on page 21-6).

Configuration in the deployment XML overrides the corresponding configuration made using annotations.

Using Annotations

You can specify a transaction timeout for an EJB 3.0 session bean using the following OC4J-proprietary annotations and their attributes:

- `@StatelessDeployment` attribute `transactionTimeout`
- `@StatefulDeployment` attribute `transactionTimeout`

For more information on these attributes, see Table A–1.

Example 21–5 shows how to configure these attributes for an EJB 3.0 stateless session bean using the `@StatelessDeployment` annotation.

Example 21–5 @StatelessDeployment transactionTimeout Attribute

```
import javax.ejb.Stateless;
import oracle.j2ee.ejb.StatelessDeployment;

@Stateless
@StatelessDeployment(transactionTimeout=10)
public class HelloWorldBean implements HelloWorld {
    public void sayHello(String name) {
        System.out.println("Hello " + name + " from first EJB3.0");
    }
}
```

Using Deployment XML

In the `orion-ejb-jar.xml` file you set a session bean transaction timeout with the `transaction-timeout` attribute of the `<session-deployment>` element.

For example, if you wanted to set the global transaction timeout to 180 seconds, you would do as follows:

```
<session-deployment ...  transaction-timeout="180"
  ...
</session-deployment>
```

If you change this property using this method, you must restart OC4J to apply your changes.

Configuring a Transaction Timeout for a Message-Driven Bean

You can configure a transaction timeout for a message-driven bean using OC4J-proprietary annotations (see "Using Annotations" on page 21-8) or using the `orion-ejb-jar.xml` file (see "Using Deployment XML" on page 21-8).

Because the global transaction timeout (see "Configuring a Global Transaction Timeout" on page 21-6) does not apply to message-driven beans, you must configure transaction timeout for each message-driven bean if you want to change the default transaction timeout for a message-driven bean.

The type of message service provider you use (see "What Message Service Providers Can you use With Your MDB?" on page 2-21) affects your transaction timeout options in the following way:

- J2EE Connector Architecture (J2CA) adapter message provider: you can change the transaction timeout (see "J2CA Adapter Message Service Provider" on page 21-9).
- OEMS JMS: you cannot change the transaction timeout from the default of 86,400 seconds (1 day).
- OEMS JMS Database: you can change the transaction timeout (see "Non-J2CA Adapter Message Service Provider" on page 21-8).

Configuration in the deployment XML overrides the corresponding configuration made using annotations.

Using Annotations

You can specify a transaction timeout for an EJB 3.0 session bean using the OC4J-proprietary annotation `@MessageDrivenDeployment` attribute transactionTimeout.

For more information on this attribute, see Table A–3.

Example 21–5 shows how to configure this attribute for an EJB 3.0 message-driven bean using the `@MessageDrivenDeployment` annotation.

Example 21–6 @MessageDrivenDeployment

```
import javax.ejb.MessageDriven;
import oracle.j2ee.ejb.MessageDrivenDeployment;
import javax.ejb.ActivationConfigProperty;
import javax.annotation.Resource;

@MessageDriven(
    activationConfig = {
        @ActivationConfigProperty(
            propertyName="messageListenerInterface",
            propertyValue="javax.jms.MessageListener"),
        @ActivationConfigProperty(
            propertyName="connectionFactoryJndiName",
            propertyValue="jms/TopicConnectionFactory"),
         @ActivationConfigProperty(
            propertyName="destinationName",
            propertyValue="jms/demoTopic"),
         @ActivationConfigProperty(
            propertyName="destinationType",
            propertyValue="javax.jms.Topic"),
         @ActivationConfigProperty(
            propertyName="messageSelector",
            propertyValue="RECIPIENT = 'MDB'")
    }
)
@MessageDrivenDeployment(transactionTimeout=10)
public class MessageLogger implements MessageListener, TimedObject {
    @Resource javax.ejb.MessageDrivenContext mc;

    public void onMessage(Message message) {
    ...
    }

    public void ejbTimeout(Timer timer) {
    ...
    }
}
```

Using Deployment XML

You set the transaction timeout in the `orion-ejb-jar.xml` file. How you configure this value depends on the type of message-service provider you are using:

- Non-J2CA Adapter Message Service Provider
- J2CA Adapter Message Service Provider

Non-J2CA Adapter Message Service Provider

If you are using a non-J2CA adapter message service provider like OEMS JMS or OEMS JMS Database, use the `transaction-timeout` attribute of the `<message-driven-deployment>` element.

For example, if you are using OEMS JMS or OEMS JMS Database, and you wanted to set the transaction timeout to 180 seconds, you would do as follows:

```
<message-driven-deployment ...  transaction-timeout="180"
  ...
</message-driven-deployment>
```

J2CA Adapter Message Service Provider

If you are using a J2CA adapter message service provider, use the `<config-property>` element to set the `transactionTimeout` configuration property.

For example, if you are using a J2CA adapter message service provider, and you wanted to set the transaction timeout to 180 seconds, you would do as follows:

```
<message-driven-deployment ... >
...
    <config-property>
        <config-property-name>transactionTimeout</config-property-name>
        <config-property-value>180</config-property-value>
    </config-property>
...
</message-driven-deployment>
```

In either case, if you change this property using this method, you must restart OC4J to apply your changes.

Transaction Best Practices

This section describes the preferred approach to using transactions in an EJB application, including the following:

- Using Container Managed Transactions With Datasource Connections
- Using a Rollback Strategy

Using Container Managed Transactions With Datasource Connections

If you are using container-managed transactions and you use a data source connection, bear in mind that the connection is not released until the transaction commits. This is particularly important if you are using the data source connection in a loop: in this case, you should acquire and release the connection outside of the loop to avoid inadvertently exhausting your connection pool.

Consider a session bean that you configure for container-managed transactions. This session bean has method `runQueryConnectionEveryTime`, as Example 21–7 shows. When this method is called, a container-managed transaction is opened. In each iteration of the `for` loop, a connection is acquired and closed. However, the closed connection is not released until the method returns and the container-managed transaction commits. Depending on the number of iterations, this design can exhaust your connection pool.

To avoid this problem, you should acquire and close the connection outside of the loop, as Example 21–8 shows. By doing so, you guarantee that only one connection will be held until the container-managed transaction commits.

Example 21–7 Incorrect: count Number of Connections Held Until Commit

```
public static long runQueryConnectionEveryTime (int count) {
    InitialContext ic = new InitialContext();
```

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

    for (int i = 0; i < count; i++) {
        Connection con = ds.getConnection();  //connection created inside loop

        PreparedStatement ps = con.prepareStatement(
            "select AAA_ID, AAA_A FROM AAA_TABLE where AAA_ID = ? ");

        OracleStatement os = (OracleStatement)ps;
        os.defineColumnType(1, Types.BIGINT);
        ps.setLong(1, i);
        ResultSet rs = ps.executeQuery();
        rs.close();
        ps.close();

        con.close(); //connection closed inside loop
    }
}
```

Example 21–8 Correct: Only One Connection Held Until Commit

```
public static long runQueryConnectionEveryTime (int count) {
    InitialContext ic = new InitialContext();
    DataSource ds = (DataSource) ic.lookup("jdbc/OracleDS");

    Connection con = ds.getConnection();  //connection created outside loop

    for (int i = 0; i < count; i++) {
        PreparedStatement ps = con.prepareStatement(
            "select AAA_ID, AAA_A FROM AAA_TABLE where AAA_ID = ? ");

        OracleStatement os = (OracleStatement)ps;
        os.defineColumnType(1, Types.BIGINT);
        ps.setLong(1, i);
        ResultSet rs = ps.executeQuery();
        rs.close();
        ps.close();
    }

    con.close(); //connection closed outside loop
}
```

Using a Rollback Strategy

An enterprise bean with container-managed transaction demarcation can use the `setRollbackOnly` method of its `javax.ejb.EJBContext` object to mark the transaction such that the transaction can never commit.

Typically, you would do this to protect data integrity before throwing an application exception when the application exception does not automatically cause the container to rollback the transaction.

For example, an `AccountTransfer` bean which debits one account and credits another account could mark a transaction for rollback, if it successfully performs the debit, but fails during the credit operation.

For more information, see the following:

- "What is EJB Context?" on page 1-6
- "Accessing an EJB 3.0 EJBContext" on page 29-20
- "Accessing an EJB 2.1 EJBContext" on page 29-27

22

Configuring Security Services

This chapter explains security service configuration as it applies specifically to Java EE applications, including the following:

- Granting Permissions in Browser
- Defining Users, Groups, and Roles in an EJB Application
- Specifying Credentials in EJB Clients
- Using EJB 3.0 Security Annotations
- Retrieving Credentials From an Enterprise Bean Using the JAAS API
- Defining a Custom JAAS Login Module for an EJB Application

For more information, see the following:

- "Understanding EJB Security Services" on page 2-20
- *Oracle Containers for J2EE Security Guide*

Granting Permissions in Browser

If you download the EJB application as a client where the security manager is active, you must grant the following permissions before you can execute:

```
permission java.net.SocketPermission "*:*", "connect,resolve";
permission java.lang.RuntimePermission "createClassLoader";
permission java.lang.RuntimePermission "getClassLoader";
permission java.util.PropertyPermission "*", "read";
permission java.util.PropertyPermission "LoadBalanceOnLookup", "read,write";
```

Defining Users, Groups, and Roles in an EJB Application

For EJB authentication and authorization, you define the principals, under which each method executes, by configuring the EJB deployment descriptor. The container enforces that the user, who is trying to execute the method, is the same as defined within the deployment descriptor.

The EJB deployment descriptor enables you to define security roles under which each method is allowed to execute. These methods are mapped to users or groups in the OC4J-specific deployment descriptor. The users and groups are defined within your designated security user managers, which uses either the Oracle Application Server Java Authentication and Authorization Service (JAAS) Provider (OracleAS JAAS Provider) or XML user manager. For a full description of security user managers, see the *Oracle Containers for J2EE Services Guide.*

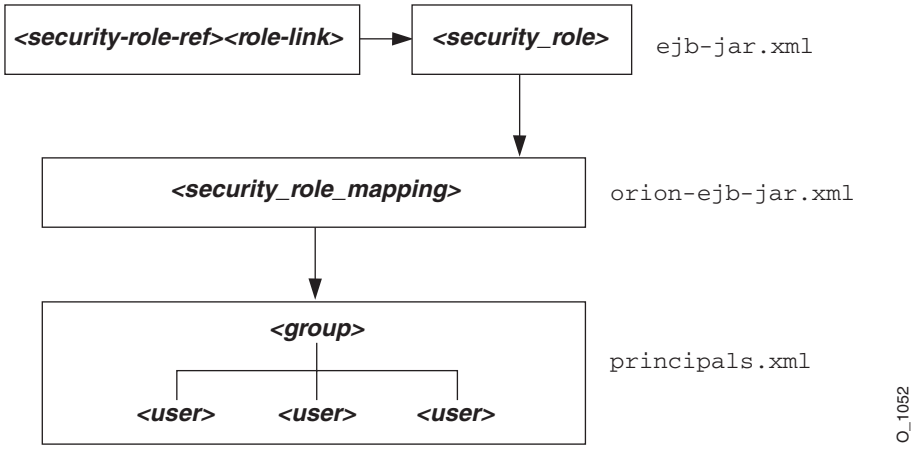
For authentication and authorization, this section focuses on XML configuration within the EJB deployment descriptors. EJB authorization is specified within the EJB and OC4J-specific deployment descriptors. You can manage the authorization piece of your security within the deployment descriptors as follows:

- The EJB deployment descriptor describes access rules using logical roles.
- The OC4J-specific deployment descriptor maps the logical roles to concrete users and groups, which are defined either the OracleAS JAAS Provider or XML user managers.

Users and groups are identities known by the container. Roles are the *logical* identities each application uses to indicate access rights to its different objects. The username/passwords can be digital certificates and, in the case of SSL, private key pairs.

Thus, the definition and mapping of roles is demonstrated in Figure 22–1.

Figure 22–1 Role Mapping



Defining users, groups, and roles are discussed in the following sections:

- Specifying Users and Groups
- Specifying Logical Roles in the EJB Deployment Descriptor
- Specifying a Role for an EJB Method
- Specifying Unchecked Security for EJB Methods
- Specifying the runAs Security Identity
- Mapping Logical Roles to Users and Groups
- Specifying a Default Role Mapping for Undefined Methods
- Specifying Users and Groups by the Client

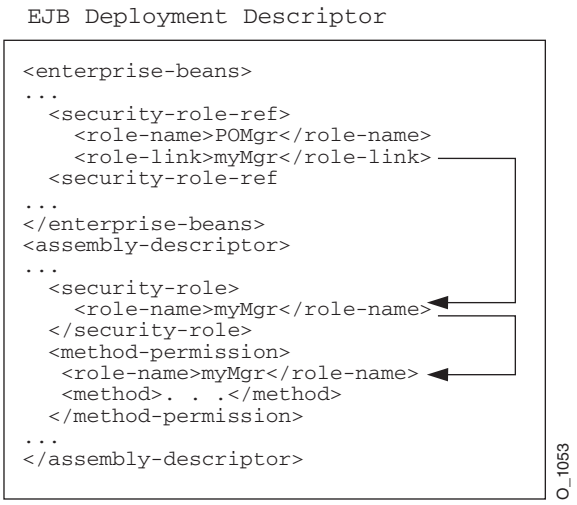
Specifying Users and Groups

OC4J supports the definition of users and groups: either shared by all deployed applications, or specific to given applications. You define shared or application-specific users and groups within either the OracleAS JAAS Provider or XML user managers. See the *Oracle Containers for J2EE Services Guide*.for directions.

Specifying Logical Roles in the EJB Deployment Descriptor

As Figure 22–2 shows, you can use a logical name for a role within your bean implementation, and map this logical name to the correct database role or user. The mapping of the logical name to a database role is specified in the OC4J-specific deployment descriptor. See "Mapping Logical Roles to Users and Groups" on page 22-8 for more information.

Figure 22–2 Security Mapping



If you use a logical name for a database role within your bean implementation for methods such as `isCallerInRole`, you can map the logical name to an actual database role by doing the following:

1. Declare the logical name within the `<enterprise-beans>` section `<security-role-ref>` element. For example, to define a role used within the purchase order example, you may have checked, within the bean's implementation, to see if the caller had authorization to sign a purchase order. Thus, the caller would have to be signed in under a correct role. In order for the bean to not need to be aware of database roles, you can check `isCallerInRole` on a logical name, such as `POMgr`, since only purchase order managers can sign off on the order. Thus, you would define the logical security role, `POMgr` within the `<security-role-ref><role-name>` element within the `<enterprise-beans>` section, as follows:

   ```
   <enterprise-beans>
   ...
     <security-role-ref>
      <role-name>POMgr</role-name>
      <role-link>myMgr</role-link>
     </security-role-ref>
   </enterprise-beans>
   ```

   The `<role-link>` element within the `<security-role-ref>` element can be the actual database role, which is defined further within the `<assembly-descriptor>` section. Alternatively, it can be another logical name, which is still defined more in the `<assembly-descriptor>` section and is mapped to an actual database role within the Oracle-specific deployment descriptor.

> **Note:** The `<security-role-ref>` element is not required. You only specify it when using security context methods within your bean.

2. Define the role and the methods to which it applies. In the purchase order example, any method executed within the `PurchaseOrder` bean must have authorized itself as `myMgr`. Note that `PurchaseOrder` is the name declared in the `<entity | session><ejb-name>` element.

   Thus, the following defines the role as `myMgr`, the enterprise bean as `PurchaseOrder`, and all methods by denoting the '*' symbol.

   > **Note:** The `myMgr` role in the `<security-role>` element is the same as the `<role-link>` element within the `<enterprise-beans>` section. This ties the logical name of `POMgr` to the `myMgr` definition.

```
<assembly-descriptor>
 <security-role>
  <description>Role needed purchase order authorization</description>
  <role-name>myMgr</role-name>
 </security-role>
 <method-permission>
  <role-name>myMgr</role-name>
  <method>
   <ejb-name>PurchaseOrder</ejb-name>
   <method-name>*</method-name>
  </method>
 </method-permission>
...
</assembly-descriptor>
```

After performing both steps, you can refer to `POMgr` within the bean's implementation and the container translates `POMgr` to `myMgr`.

> **Note:** If you define different roles within the `<method-permission>` element for methods in the same bean, the resulting permission is a union of all the method permissions defined for the methods of this bean.

Specifying a Role for an EJB Method

You can specify which security roles are allowed to invoke an enterprise bean method.

In an EJB 3.0 application, you can use annotations (see "Using Annotations" on page 22-4).

In an EJB 3.0 or EJB 2.1 application, you can use the `ejb-jar.xml` deployment descriptor (see "Using Deployment XML" on page 22-5).

Using Annotations

In an EJB 3.0 application, you can use the `@RolesAllowed` annotation to specify the security roles permitted to access methods in an application, as Example 22–1 shows.

Example 22–1 @RolesAllowed

```
@RolesAllowed("Users")
public class Calculator {

    @RolesAllowed("Administrator")
    public void setNewRate(int rate) {
    ...
    }
}
```

You can apply this annotation to a class, method, or both.

When applied to a method, the specification overrides class specification, if present.

For more information on security annotations, see "Using EJB 3.0 Security Annotations" on page 22-12.

Using Deployment XML

The `<method-permission><method>` element is used to specify the security role for one or more methods within an interface or implementation. According to the EJB specification, this definition can be of one of the following forms:

- Defining all methods within a bean by specifying the bean name and using the '*' character to denote all methods within the bean, as follows:

```
<method-permission>
  <role-name>myMgr</role-name>
  <method>
    <ejb-name>EJBNAME</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>
```

- Defining a specific method that is uniquely identified within the bean. Use the appropriate interface name and method name, as follows:

```
<method-permission>
  <role-name>myMgr</role-name>
  <method>
    <ejb-name>myBean</ejb-name>
    <method-name>myMethodInMyBean</method-name>
  </method>
</method-permission>
```

> **Note:** If there are multiple methods with the same overloaded name, the element of this style refers to all the methods with the overloaded name.

- Defining a method with a specific signature among many overloaded versions, as follows:

```
<method-permission>
  <role-name>myMgr</role-name>
  <method>
    <ejb-name>myBean</ejb-name>
    <method-name>myMethod</method-name>
    <method-params>
      <method-param>javax.lang.String</method-param>
      <method-param>javax.lang.String</method-param>
    </method-params>
```

```
  </method>
</method-permission>
```

The parameters are the fully-qualified Java types of the method's input parameters. If the method has no input arguments, the `<method-params>` element contains no elements. Arrays are specified by the array element's type, followed by one or more pair of square brackets, such as `int[ ] [ ]`.

Specifying Unchecked Security for EJB Methods

If you want certain methods to not be checked for security roles, you define these methods as unchecked.

In an EJB 3.0 application, you can use annotations (see "Using Annotations" on page 22-6).

In an EJB 3.0 or EJB 2.1 application, you can use the `ejb-jar.xml` deployment descriptor (see "Using Deployment XML" on page 22-6).

Using Annotations

In an EJB 3.0 application, you can use the `@PermitAll` annotation to specify that all security roles are permitted to access methods in an application, as Example 22–2 shows.

Example 22–2 @PermitAll

```
@RolesAllowed("Users")
public class Calculator {

    @RolesAllowed("Administrator")
    public void setNewRate(int rate) {
    ...
    }
    @PermitAll
    public long convertCurrency(long amount) {
    ...
    }
}
```

You can apply this annotation to a class or method.

When applied to a class, the specification applies to all methods.

When applied to a method, the specification applies only to that method.

When using this annotation, observe the restrictions described in "Using EJB 3.0 Security Annotations" on page 22-12.

Using Deployment XML

The `<method-permission><unchecked>` element is used to specify that all security roles are permitted to access a method, as follows:

```
<method-permission>
  <unchecked/>
  <method>
     <ejb-name>EJBNAME</ejb-name>
     <method-name>*</method-name>
  </method>
</method-permission>
```

Instead of a <role-name> element defined, you define an <unchecked/> element. When executing any methods in the EJBNAME bean, the container does not check for security. Unchecked methods always override any other role definitions.

Specifying the runAs Security Identity

You can specify that all methods of an enterprise bean execute under a specific identity. That is, the container does not check different roles for permission to run specific methods; instead, the container executes all of the enterprise bean methods under the specified security identity. You can specify a particular role or the caller's identity as the security identity.

In an EJB 3.0 application, you can use annotations (see "Using Annotations" on page 22-7).

In an EJB 3.0 or EJB 2.1 application, you can use the ejb-jar.xml deployment descriptor (see "Using Deployment XML" on page 22-7).

Using Annotations

In an EJB 3.0 application, you can use the @RunAs annotation to specify the role of the application during execution in a Java EE container, as Example 22–1 shows.

Example 22–3 @RunAs

```
@RunAs("Admin")
public class Calculator {
    ...
}
```

You can apply this annotation to a class.

For more information on security annotations, see "Using EJB 3.0 Security Annotations" on page 22-12.

Using Deployment XML

Specify the runAs security identity in the <security-identity> element, which is contained in the <enterprise-beans> section. The following XML demonstrates that the POMgr is the role under which all the entity bean methods execute:

```
<enterprise-beans>
  <entity>
  ...
    <security-identity>
      <run-as>
        <role-name>POMgr</role-name>
      </run-as>
    </security-identity>
...
  </entity>
</enterprise-beans>
```

Alternatively, the following XML example demonstrates how to specify that all methods of the bean execute under the identity of the caller:

```
<enterprise-beans>
  <entity>
  ...
    <security-identity>
      <use-caller-identity/>
```

```
    </security-identity>
...
  </entity>
</enterprise-beans>
```

Mapping Logical Roles to Users and Groups

You can use logical roles or actual users and groups in the EJB deployment descriptor. However, if you use logical roles, you must map them to the actual users and groups defined either in the OracleAS JAAS Provider or XML User Managers.

Map the logical roles defined in the application deployment descriptors to OracleAS JAAS Provider or XML User Manager users or groups through the `<security-role-mapping>` element in the OC4J-specific deployment descriptor as follows:

- The `name` attribute of this element defines the logical role that is to be mapped.
- The `group` or `user` element maps the logical role to a group or user name. This group or user must be defined in the OracleAS JAAS Provider or XML User Manager configuration. See the *Oracle Containers for J2EE Services Guide* for a description of the OracleAS JAAS Provider and XML User Managers.

Example 22–4 Mapping Logical Role to Actual Role

This example maps the logical role `POMGR` to the `managers` group in the `orion-ejb-jar.xml` file. Any user that can log in as part of this group is considered to have the `POMGR` role; thus, it can execute the methods of `PurchaseOrderBean`.

```
<security-role-mapping name="POMGR">
 <group name="managers" />
</security-role-mapping>
```

> **Note:** You can map a logical role to a single group or to several groups.

To map this role to a specific user, do the following:

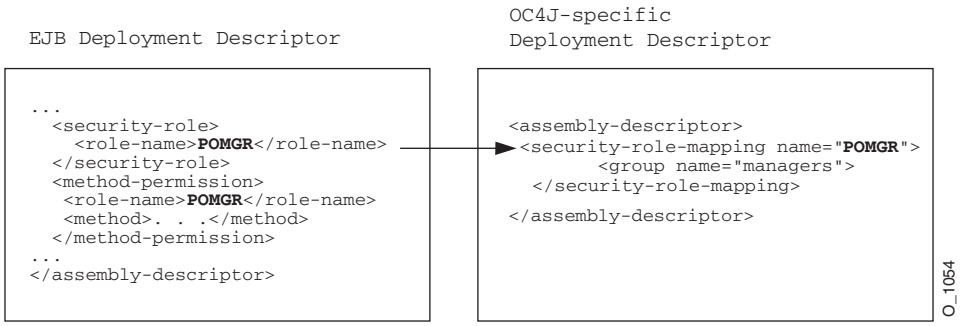
```
<security-role-mapping name="POMGR">
 <user name="guest" />
</security-role-mapping>
```

Lastly, you can map a role to a specific user within a specific group, as follows:

```
<security-role-mapping name="POMGR">
 <group name="managers" />
 <user name="guest" />
</security-role-mapping>
```

As shown in Figure 22–3, the logical role name for `POMGR` defined in the EJB deployment descriptor is mapped to `managers` within the OC4J-specific deployment descriptor in the `<security-role-mapping>` element.

Figure 22–3 Security Mapping



Notice that the `<role-name>` in the EJB deployment descriptor is the same as the name attribute in the `<security-role-mapping>` element in the OC4J-specific deployment descriptor. This is what identifies the mapping.

Specifying a Default Role Mapping for Undefined Methods

If any methods have not been associated with a role mapping, they are mapped to the default security role through the `<default-method-access>` element in the `orion-ejb-jar.xml` file. The following is the automatic mapping for any insecure methods:

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

The default role is `<default-ejb-caller-role>` and is defined in the `name` attribute. You can replace this string with any name for the default role.

The `impliesAll` attribute indicates whether any security role checking occurs for these methods. In the `orion-ejb-jar.xml` file, the `impliesAll` attribute has the following defaults:

- If `<security-role-mapping>` is specified in the `orion-ejb-jar.xml` file and `impliesAll` is not set, then this attribute defaults to `false`: the container checks for this default role on these methods.
- If `<security-role-mapping>` is not specified in the `orion-ejb-jar.xml` file, the OC4J EJB layer defaults this attribute to `true`: no security role checking occurs for these methods.

If the `impliesAll` attribute is false, you must map the default role defined in the `name` attribute to a OracleAS JAAS Provider or XML user or group through the `<user>` and `<group>` elements. The following example shows how all methods not associated with a method permission are mapped to the "`others`" group:

```
<default-method-access>
    <security-role-mapping name="default-role" impliesAll="false" >
        <group name="others" />
    </security-role-mapping>
</default-method-access>
```

Specifying Users and Groups by the Client

In order for the client to access methods that are protected by users and groups, the client must provide the correct user or group name with a password that the OracleAS JAAS Provider or XML User Manager recognizes. And the user or group must be the same one as designated in the security role for the intended method. See "Specifying Credentials in EJB Clients" on page 22-10 for more information.

> **Note:** For basic OC4J security configuration information, including CSiV2, see the *Oracle Containers for J2EE Security Guide*.

Specifying Credentials in EJB Clients

Depending on the type of client, you may need to specify security credentials before your client can access an enterprise bean or other JNDI-accessible resource.

Table 22–1 classifies EJB clients by where they are deployed relative to the target enterprise bean they invoke. Where you deploy the client relative to its target enterprise bean determines whether or not you must specify security credentials.

Table 22–1 Client Security Credential Requirements

Client Type	Relationship to Target EJB	Set Credentials?
Any client	Client and target enterprise bean are collocated	No
Any client	Client and target enterprise bean are deployed in the same application	No
Any client	Target enterprise bean deployed in an application that is designated as the client's parent[1]	No
Any client	Client and target enterprise bean are not collocated, not deployed in the same application, and target EJB application is not client's parent[1].	Yes

[1] See the *Oracle Containers for J2EE Developer's Guide* for more information on how to set the parent of an application.

When you access enterprise beans in a *remote* container (that is, if the client and target enterprise bean are not collocated, not deployed in the same application, and the target enterprise bean application is not the client's parent), you must pass valid credentials to the remote container. How your client passes its credentials depends on the type of client:

- EJB Client: pass credentials within the `InitialContext`, which is created to look up the remote enterprise beans (see "Specifying Credentials in the Initial Context" on page 22-11).
- Standalone Java Client: define credentials in the `jndi.properties` file deployed with the EAR file (see "Specifying Credentials in JNDI Properties" on page 22-11).
- Servlet or JSP Client: pass credentials within the `InitialContext`, which is created to look up the remote enterprise beans (see "Specifying Credentials in the Initial Context" on page 22-11).

In addition, all clients can specify security properties in the `ejb_sec.properties` file (see "Specifying EJB Client Security Properties in the ejb_sec.properties File" on page 22-12).

For more information, see the following:

- "What Type of Client do you Have?" on page 29-1
- *Oracle Containers for J2EE Security Guide*

Specifying Credentials in JNDI Properties

To specify credentials in a `jndi.properties` file, do the following:

1. Create or modify an existing `jndi.properties` file.
2. Configure the appropriate credentials in the `jndi.properties` file, as Example 22–6 shows.

   For property names, see the field definitions in `javax.naming.Context`.

Example 22–5 Specifying Credentials in JNDI Properties

```
java.naming.security.principal=POMGR
java.naming.security.credentials=welcome
java.naming.factory.initial=
    oracle.j2ee.server.ApplicationClientInitialContextFactory
java.naming.provider.url=opmn:ormi://opmnhost:6004:oc4j_inst1/ejbsamples
```

3. Ensure that the `jndi.properties` file is on the client's classpath.
4. Use the JNDI API in your client to look up the JNDI-accessible resource as Example 22–6 shows.

Example 22–6 Looking Up a JNDI-Accessible Resource

```
Context ic = new InitialContext();
CustomerHome = (CustomerHome)ic.lookup("java:comp/env/purchaseOrderBean");
```

At run time, JNDI uses `ClassLoader` method `getResources` to locate all application resource files named `jndi.properties` in the classpath. In doing so, it will use the JNDI properties you set in Example 22–6 to access the `purchaseOrderBean`.

For more information, see "Setting JNDI Properties With the JNDI Properties File" on page 19-22.

Specifying Credentials in the Initial Context

To specify credentials in the initial context you use to look up JNDI-accessible resources, do the following:

1. Create a `HashTable` and populate it with the required properties using `javax.naming.Context` fields as keys and `String` objects as values, as Example 22–7 shows.

Example 22–7 Specifying Credentials in the Initial Context

```
Hashtable env = new Hashtable();
env.put(Context.SECURITY_PRINCIPAL, "POMGR");
env.put(Context.SECURITY_CREDENTIALS, "welcome");
env.put("java.naming.factory.initial",
        "oracle.j2ee.server.ApplicationClientInitialContextFactory");
env.put("java.naming.provider.url",
        "opmn:ormi://opmnhost:6004:oc4j_inst1/ejbsamples");
```

2. When you instantiate the initial context, pass the `HashTable` into the initial context constructor, as Example 22–8 shows.

Example 22–8 Looking Up a JNDI-Accessible Resource

```
Context ic = new InitialContext (env);
```

```
CustomerHome = (CustomerHome) ic.lookup ("java:comp/env/purchaseOrderBean");
```

For more information, see the following:

- "Configuring the Initial Context Factory" on page 19-19
- "Setting JNDI Properties in the Initial Context" on page 19-23

Specifying EJB Client Security Properties in the ejb_sec.properties File

Any client, whether running inside a server or not, has EJB security properties controlled by an `ejb_sec.properties` file. You use this file to specify general security options as well as options specific to the Common Secure Interoperability Version 2 protocol (CSIv2).

For more information, see "Common Secure Interoperability Protocol" in the *Oracle Containers for J2EE Security Guide*.

Using EJB 3.0 Security Annotations

In an EJB 3.0 application, you can use the `javax.annotation.security` annotations defined in JSR250 to configure security options on EJB 3.0 session beans.

Table 22–2 summarizes the security annotations that OC4J supports. For an example of how to use these annotations, see "Using Annotations" on page 22-13.

Table 22–2 Security Annotations

Annotation	Description	Applicable To
`@RunAs`	Defines the role of the application during execution in a Java EE container. The role must map to the user/group information in the container's security realm. For more information, see "Specifying the runAs Security Identity" on page 22-7.	Class
`@RolesAllowed`	Specifies the security roles permitted to access methods in an application. For more information, see "Specifying a Role for an EJB Method" on page 22-4.	Class, method, or both. Method specification overrides class specification if present.
`@PermitAll`	Specifies that all security roles are allowed to invoke the specified methods. For more information, see "Specifying Unchecked Security for EJB Methods" on page 22-6.	Class or method. Class specification applies to all methods. Method specification applies only to that method.
`@DenyAll`	Specifies that no security roles are allowed to invoke the specified methods.	Class or method. Class specification applies to all methods. Method specification applies only to that method.
`@DeclareRoles`	Specifies the security roles used by the application.	Class

When using `@PermitAll`, `@DenyAll` and `@RolesAllowed` annotations, observe the following restrictions:

- `@PermitAll`, `@DenyAll`, and `@RolesAllowed` annotations must not be applied on the same method or class.
- In the following cases, the method level annotations take precedence over the class level annotation:
  - `@PermitAll` is specified at the class level and `@RolesAllowed` or `@DenyAll` are specified on methods of the same class;

- `@DenyAll` is specified at the class level and `@PermitAll` or `@RolesAllowed` are specified on methods of the same class;
- `@RolesAllowed` is specified at the class level and `@PermitAll` or `@DenyAll` are specified on methods of the same class.

Note: You can download an EJB 3.0 security annotation code example from: http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30security/doc/how-to-ejb30-security-ejb.html.

Using Annotations

Example 22–9 shows how to use the `@RolesAllowed` annotation. For more information and examples, see the JSR250 specification.

Example 22–9 @RolesAllowed

```
@RolesAllowed("Users")
public class Calculator {

    @RolesAllowed("Administrator")
    public void setNewRate(int rate) {
    ...
    }
}
```

Retrieving Credentials From an Enterprise Bean Using the JAAS API

OC4J supports the use of standard JAAS API to retrieve the `Subject`, `Principal`, and credentials from within business methods and life cycle methods of session beans (stateless and stateful) and entity beans.

Example 22–10 shows how you can use the JAAS API to retrieve credentials in a business method of an enterprise bean deployed to OC4J.

Example 22–10 Using JAAS API to Retrieve Credentials

```
public class Calculator {
    // Buisness method
    public void setNewRate(int rate) {
    ...
        AccessControlContext actx = AccessController.getContext();
        Subject subject = Subject.getSubject(actx);
        Set principals = subject.getPrincipals();
    ...
    }
}
```

Defining a Custom JAAS Login Module for an EJB Application

Within the JAAS pluggable authentication framework, an application server and any underlying authentication services remain independent from each other. Authentication services can be plugged in through JAAS login modules without requiring modifications to the application server or application code. A login module is primarily responsible for authenticating a user based on supplied credentials (such as a password), and adding the proper principals (such as roles) to the subject.

Possible types of JAAS login modules include a principal-mapping JAAS module, a credential-mapping JAAS module, a Kerberos JAAS module, or a custom login module.

To use a custom JAAS login module with your enterprise beans, the following elements must be configured:

- `<jazn-loginconfig>` in `system-jazn-data.xml`
- `<jazn>` in `orion-application.xml`
- `<namespace-access>` in `orion-application.xml`

For more information, see "Login Modules" in the *Oracle Containers for J2EE Security Guide*.

23

Configuring Message Services

This chapter describes how to configure Java Message Service (JMS) and non-JMS message service providers, including the following:

- Configuring a J2CA Resource Adapter for use With Your Message Service Provider
- Configuring an OEMS JMS Message Service Provider
- Configuring an OEMS JMS Database Message Service Provider

For more information, see the following:

- "What Message Service Providers Can you use With Your MDB?" on page 2-21
- "Implementing an EJB 3.0 Message-Driven Bean" on page 9-1
- "Implementing an EJB 2.1 Message-Driven Bean" on page 17-1
- "Java Message Service" in the *Oracle Containers for J2EE Services Guide*

Configuring a J2CA Resource Adapter for use With Your Message Service Provider

To configure a J2CA resource adapter such as the Oracle JMS Connector (see "Oracle JMS Connector: J2EE Connector Architecture (J2CA)-Based Provider" on page 2-21) for use with your message service provider, you must do the following:

1. Install and configure the J2CA adapter (see "Installing and Configuring a J2CA Adapter" on page 23-2).
2. Choose appropriate JNDI names for your connection factory (see "J2CA Message Service Provider Connection Factory Names" on page 23-2).
3. Configure the appropriate deployment XML files (see "Configuring OC4J J2CA Resource Adapter Deployment XML Files" on page 23-2).

   Using these deployment XML files, you can specify factories that are either not XA-compliant, when two-phase commit (2PC) transactions are not needed, or XA-compliant, when 2PC transactions are needed. For more information on 2PC, see "How do You Participate in a Global or Two-Phase Commit (2PC) Transaction?" on page 2-20.
4. Configure your message-driven beans to access your message service provider using your J2CA resource adapter.

   For more information, see the following:

- "Configuring an EJB 3.0 MDB to Access a Message Service Provider Using J2CA" on page 10-1
- "Configuring an EJB 2.1 MDB to Access a Message Service Provider Using J2CA" on page 18-1

Note: Oracle recommends that you access a message service provider using a J2CA resource adapter such as the Oracle JMS Connector. For more information, see "Restrictions When Accessing a Message Service Provider Without a J2CA Resource Adapter" on page 2-25.

Note: For a complete code example of configuring a J2CA message service provider resource adapter and MDB application, see `http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/how-to-gjra-with-oracleasjms/doc/how-to-gjra-with-oracleasjms.html`.

J2CA Message Service Provider Connection Factory Names

The actual JNDI names for the destination and connection factory depend on your J2CA installation as defined in your `oc4j-connectors.xml` file (see "Configuring OC4J J2CA Resource Adapter Deployment XML Files" on page 23-2).

Typically, it will be composed of `java:<Prefix>/<FactoryName>` where `<Prefix>` is an optional JNDI location like `comp/env/eis`, and `<FactoryName>` is the name of the `javax.cci.ConnectionFactory` for your adapter.

Installing and Configuring a J2CA Adapter

OC4J includes the Oracle JMS Connector: a generic JMS J2CA resource adapter that integrates OC4J with OEMS JMS and OEMS JMS Database message service providers, as well as non-Oracle JMS providers such as WebSphereMQ, Tibco, and SonicMQ.

For more information, see the following:

- "Overview: Administering Resource Adapters" in the *Oracle Containers for J2EE Resource Adapter Administrator's Guide*.
- "Oracle JMS Connector: J2EE Connector Architecture (J2CA)-Based Provider" on page 2-21.

Configuring OC4J J2CA Resource Adapter Deployment XML Files

To configure a J2CA message service provider, you must configure the following deployment XML files:

- `ra.xml`
- `oc4j-ra.xml`
- `oc4j-connectors.xml`

Using these deployment XML files, you can specify factories that are either not XA-compliant, when two-phase commit (2PC) transactions are not needed, or XA-compliant, when 2PC transactions are needed. For more information on 2PC, see

"How do You Participate in a Global or Two-Phase Commit (2PC) Transaction?" on page 2-20.

For more information, see the following:

- "Binding and Configuring a Connection Factory: Basic Settings" in the *Oracle Containers for J2EE Resource Adapter Administrator's Guide*
- "OC4J Resource Adapter Configuration Files" in the *Oracle Containers for J2EE Resource Adapter Administrator's Guide*

Configuring an OEMS JMS Message Service Provider

To configure the OEMS JMS message service provider (see "OEMS JMS: In-Memory or File-Based Provider" on page 2-23), you must do the following:

1. Choose appropriate JNDI names for your destination and connection factory (see "OEMS JMS Destination and Connection Factory Names").
2. Configure the `<OC4J_HOME>/j2ee/home/config/jms.xml` file (see "Configuring jms.xml") to specify the type of destination and connection factory.

   You can specify factories that are either not XA-compliant, when two-phase commit (2PC) transactions are not needed, or XA-compliant, when 2PC transactions are needed. For more information on 2PC, see "How do You Participate in a Global or Two-Phase Commit (2PC) Transaction?" on page 2-20.
3. Optionally, map the actual JNDI names to logical names (see "Configuring an Environment Reference to a JMS Destination or Connection Resource Manager Connection Factory (JMS 1.0)" on page 19-14).
4. Configure your message-driven beans to access your OEMS JMS message service provider.

   For more information, see the following:

   - "Configuring an EJB 3.0 MDB to Access a Message Service Provider Using J2CA" on page 10-1
   - "Configuring an EJB 3.0 MDB to Access a Message Service Provider Directly" on page 10-3
   - "Configuring an EJB 2.1 MDB to Access a Message Service Provider Using J2CA" on page 18-1
   - "Configuring an EJB 2.1 MDB to Access a Message Service Provider Directly" on page 18-3

   Note: Oracle recommends that you access a message service provider using a J2CA resource adapter such as the Oracle JMS Connector. For more information, see "Restrictions When Accessing a Message Service Provider Without a J2CA Resource Adapter" on page 2-25.

OEMS JMS Destination and Connection Factory Names

The actual JNDI names for the JMS destination and connection factory are the ones you specify in the `jms.xml` file (see "Configuring jms.xml" on page 23-4).

Table 23–1 lists the form of these names.

Table 23–1 OEMS JMS Destination and Connection Factory Names

Type	Form
Queue	`jms/Queue/<QName>`
Queue Connection Factory	`jms/Queue/<QCFName>`
Topic	`jms/Topic/<TName>`
Topic Connection Factory	`jms/Topic/<TCFName>`

Configuring jms.xml

You configure OEMS JMS options in the *<OC4J_HOME>*`/j2ee/home/config/jms.xml` file. In this release, the `jms.xml` is defined by the XML schema document (XSD) located at `http://www.oracle.com/technology/oracleas/schema/jms-server-10_1.xsd`.

Some of the options you can configure in the `jms.xml` file include the following:

- JMS `Destination` objects used by the MDB.
- Topic or queue in the `jms.xml` file to which the client sends all messages that are destined for the MDB.
- The name, location, and connection factory for either `Destination` type must be specified.

  You can specify factories that are either not XA-compliant, when two-phase commit (2PC) transactions are not needed, or XA-compliant, when 2PC transactions are needed. For more information on 2PC, see "How do You Participate in a Global or Two-Phase Commit (2PC) Transaction?" on page 2-20.
- If your MDB accesses a database for inquiries and so on, then you can configure the data source used. For information on data source configuration, see the Data Source chapter in the Oracle Containers for J2EE Services Guide.
- Path to a file in which OEMS JMS events and errors are written.

Example 23–1 shows the `jms.xml` file configuration for an EJB 2.1 MDB that specifies a queue (named `jms/Queue/rpTestQueue`) that is used by the message-driven bean `rpTestMdb` (see Example 17–1). The queue connection factory is defined as `jms/Queue/myQCF`. In addition, a topic is defined named `jms/Topic/rpTestTopic`, with a connection factory of `jms/Topic/myTCF`.

Example 23–1 jms.xml For an EJB 2.1 MDB using OEMS JMS Factories

```
<jms>
    <jms-server port="9128">
        <queue location="jms/Queue/rpTestQueue"></queue>
        <queue-connection-factory location="jms/Queue/myQCF"></queue-connection-factory>
        <topic location="jms/Topic/rpTestTopic"></topic>
        <topic-connection-factory location="jms/Topic/myTCF"></topic-connection-factory>
        <log>
             <!-- path to the log-file where JMS-events and errors are written -->
             <file path="../log/jms.log" />
        </log>
    </jms-server>
</jms>
```

Example 23–2 shows the `jms.xml` file configuration for the same MDB using XA factories for two-phase commit (2PC) support.

Example 23–2 jms.xml For an EJB 2.1 MDB using OEMS JMS XA Factories

```
<jms>
    <jms-server port="9128">
        <queue location="jms/Queue/rpTestQueue"></queue>
        <xa-queue-connection-factory location="jms/Queue/myXAQCF"></queue-connection-factory>
        <topic location="jms/Topic/rpTestTopic"></topic>
        <xa-topic-connection-factory location="jms/Topic/myXATCF"></topic-connection-factory>
        <log>
             <!-- path to the log-file where JMS-events and errors are written -->
             <file path="../log/jms.log" />
        </log>
    </jms-server>
</jms>
```

Configuring an OEMS JMS Database Message Service Provider

To configure the OEMS JMS Database message service provider (see "OEMS JMS Database: Advanced Queueing (AQ)-Based Provider" on page 2-24), you must do the following:

1. Install and configure the OEMS JMS Database provider (see "Installing and Configuring the OEMS JMS Database Provider" on page 23-6).

   You can grant privileges that either disable XA-compliant resources, when two-phase commit (2PC) transactions are not needed, or enable XA-compliant resources, when 2PC transactions are needed. For more information on 2PC, see "How do You Participate in a Global or Two-Phase Commit (2PC) Transaction?" on page 2-20.

2. Choose appropriate JNDI names for your destination and connection factory (see "OEMS JMS Database Destination and Connection Factory Names" on page 23-6).

3. Configure the `data-sources.xml` file to identify your database (see "Configuring data-sources.xml" on page 23-8).

4. Optionally, map the actual JNDI names to logical names (see "Configuring an Environment Reference to a JMS Destination or Connection Resource Manager Connection Factory (JMS 1.0)" on page 19-14).

5. Configure the `application.xml` (or `orion-application.xml`) file to identify the JNDI name of the data source that is to be used as the OEMS JMS Database provider within the `<resource-provider>` element (see "Configuring application.xml or orion-application.xml" on page 23-8).

6. Configure your message-driven beans to access your OEMS JMS Database message service provider.

   For more information, see the following:

   - "Configuring an EJB 3.0 MDB to Access a Message Service Provider Using J2CA" on page 10-1
   - "Configuring an EJB 3.0 MDB to Access a Message Service Provider Directly" on page 10-3
   - "Configuring an EJB 2.1 MDB to Access a Message Service Provider Using J2CA" on page 18-1
   - "Configuring an EJB 2.1 MDB to Access a Message Service Provider Directly" on page 18-3

> **Note:** Oracle recommends that you access a message service provider using a J2CA resource adapter such as the Oracle JMS Connector. For more information, see "Restrictions When Accessing a Message Service Provider Without a J2CA Resource Adapter" on page 2-25.

OEMS JMS Database Destination and Connection Factory Names

The actual JNDI names for the JMS destination and connection factory depend on your OEMS JMS Database installation as shown in Table 23–2.

Table 23–2 OEMS JMS Database Destination and Connection Factory Names

Type	Form
Queue	`java:comp/resource/<ProviderName>/Queues/<QName>`
Queue Connection Factory	`java:comp/resource/<ProviderName>/QueueConnectionFactories/<QCFName>`
Topic	`java:comp/resource/<ProviderName>/Topics/<TName>`
Topic Connection Factory	`java:comp/resource/<ProivderName>/TopicConnectionFactories/<TCFName>`

The values for the variables in Table 23–2 are defined as follows:

- `<ProviderName>`: the JNDI name of the data source that is providing OEMS JMS Database service (see "Configuring application.xml or orion-application.xml" on page 23-8)
- `<QName>`: the name of the queue you created in the database (see step 3 b in "Installing and Configuring the OEMS JMS Database Provider" on page 23-6).
- `<QCFName>`: the name of the queue connection factory. You may specify any arbitrary name.
- `<TName>`: the name of the topic you created in the database (see step 3 b in "Installing and Configuring the OEMS JMS Database Provider" on page 23-6).
- `<TCFName>`: the name of the topic connection factory. You may specify any arbitrary name.

Installing and Configuring the OEMS JMS Database Provider

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

1. You or your DBA must install Oracle AQ according to the *Oracle Streams Advanced Queuing User's Guide and Reference*. and generic database manuals.
2. You or your DBA should create an RDBMS user through which the MDB connects to the database and grant this user appropriate access privileges to perform OEMS JMS Database operations.

   The privileges that you need depend on what functionality you are requesting. Refer to the *Oracle Streams Advanced Queuing User's Guide and Reference*. for more information on privileges necessary for each type of function.

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

```
DROP USER jmsuser CASCADE ;

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

connect jmsuser/jmsuser;
```

You may need to grant other privileges, such as XA-compliant, two-phase commit (2PC) privileges or system administration privileges, based on what the user needs.

For more information on 2PC, see the following:

- "How do You Participate in a Global or Two-Phase Commit (2PC) Transaction?" on page 2-20
- Oracle Containers for J2EE Services Guide JTA chapter

3. You or your DBA should create the tables and queues to support the JMS `Destination` objects.

   Refer to the *Oracle Streams Advanced Queuing User's Guide and Reference*. for more information on the `DBMS_AQADM` packages and Oracle AQ messages types.

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

      In OEMS JMS Database, both topics and queues use a queue table. The `rpTestMdb` JMS example creates a single table: `rpTestQTab` for a queue.

      To create the queue table, execute the following SQL:

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

      The `multiple_consumers` parameter denotes whether there are multiple consumers or not; thus, is always `false` for a queue and `true` for a topic.

   b. Create the JMS `Destination`. If you are creating a topic, you must add each subscriber for the topic. The `rpTestMdb` JMS example requires a single queue–`rpTestQueue`.

      The following creates a queue called `rpTestQueue` within the queue table `rpTestQTab`. After creation, the queue is started:

      ```
      DBMS_AQADM.CREATE_QUEUE(
            Queue_name          => 'rpTestQueue',
            Queue_table         => 'rpTestQTab');

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

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

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

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

Note: The names defined here must be the same names used to define the queue or topic in the `orion-ejb-jar.xml` file.

Configuring data-sources.xml

Configure a data source for the database where the OEMS JMS Database provider is installed. The JMS topics and queues use database tables and queues to facilitate messaging. The type of data source you use depends on the functionality you want.

Example 23–3 shows a typical managed data source, which by default, supports global (two-phase commit) transactions.

Example 23–3 Emulated Data Source With Thin JDBC Driver

```
<connection-pool name="ScottConnectionPool">
    <connection-factory
        factory-class="oracle.jdbc.pool.OracleDataSource"
        user="scott"
        password="tiger"
        url="jdbc:oracle:thin:@//localhost:1521/ORCL" >
    </connection-factory>
</connection-pool>

<managed-data-source
    name="OracleDS"
    jndi-name="jdbc/OracleDS"
    connection-pool-name="ScottConnectionPool"
/>
```

For more information, see "Understanding EJB Data Source Services" on page 2-14.

Configuring application.xml or orion-application.xml

Identify the JNDI name of the data source that is to be used as the OEMS JMS Database provider within the `<resource-provider>` element.

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

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

The following code sample shows how to configure the JMS provider using XML syntax for OEMS JMS Database:

- `class` attribute–The OEMS JMS Database provider is implemented by the `oracle.jms.OjmsContext` class, which is configured in the `class` attribute.
- `property` attribute–Identify the data source that is to be used as this JMS provider in the `property` element. The topic or queue connects to this data source to access the tables and queues that facilitate the messaging.

The following example demonstrates that the data source identified by "`jdbc/OracleDS`" is to be used as the OEMS JMS Database provider. This JNDI name is specified in the `managed-data-source` element `jndi-name` attribute in Example 23–3. If this example used a non-emulated data source, then the name would be the same as in the `location` element.

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

24

Configuring OC4J EJB Application Clustering Services

This chapter describes the OC4J application clustering options you can configure for your EJB application, including the following:

- Configuring EJB 3.0 and EJB 2.1 Stateful Session Bean Replication Policy
- Configuring Static Retrieval Load Balancing
- Configuring DNS Load Balancing
- Configuring Load Balancing Behavior

For more information, see "Understanding OC4J EJB Application Clustering Services" on page 2-29.

Configuring EJB 3.0 and EJB 2.1 Stateful Session Bean Replication Policy

The general procedure for configuring EJB application clustering for an EJB 3.0 or EJB 2.1 stateful session bean is as follows:

1. Configure your OC4J application cluster (see "Application Clustering in OC4J" in the *Oracle Containers for J2EE Configuration and Administration Guide*).
2. Configure a replication policy for stateful session beans on each node (see "Using Deployment XML" on page 24-1).
3. Configure load-balancing behavior (see "Configuring Load Balancing Behavior" on page 24-4).
4. Deploy your enterprise bean to any one of the nodes in the cluster.

For more information, see the following:

- "State Replication" on page 2-30
- "Stateful Session EJB State Replication with Oracle Application Server Cluster (OC4J)" in the *Oracle Application Server High Availability Guide*

Using Deployment XML

To configure a replication policy, add a `<replication-policy>` element to one or more of the appropriate deployment descriptor files that Table 24–1 lists. You can specify a single replication policy that OC4J applies globally, or specify finer-grained replication policy at the application level for both Web and EJB components, or EJB components only.

Configure the `trigger` attribute to one of the following:

- `inherited` – The stateful session bean uses the state replication trigger setting you configure at the application level. This is the default value.
- `onRequestEnd` – The state of the stateful session bean is replicated to all hosts in the cluster (with the same multicast address, port) at the end of each EJB method call. If the node loses power, then the state has already been replicated. This method offers less performance than the JVM termination replication mode, because the state is sent out more often. However, the guarantee for reliance is higher.
- `onShutdown` –The state of the stateful session bean is replicated to only one other host in the cluster (with the same multicast address, port) when the JVM is terminating. This option provides the highest performance, because the state is replicated only once. However, it is not very reliable for the following reasons:
  - Your state is not replicated if the host is terminated unexpectedly.
  - The state of the bean exists only on a single host at any time; you carry a higher risk that the state does not replicate and is lost.
- `none`–Disable clustering for this stateful session bean.

> **Note:** Using Application Server Control, you cannot configure the `trigger` attribute to `inherited` or `none`. To set these values, edit the deployment XML manually. For more information, see "Using Oracle Enterprise Manager 10g Application Server Control" on page 31-1

The `scope` attribute is always set to `allAttributes` for a stateful session bean.

For more information, see "State Replication" on page 2-30.

Table 24–1 Deployment XML Files for Replication Policy Configuration

Scope	Components	Deployment XML File	See also ...
Global	Web and EJB	`application.xml`	"Stateful Session EJB State Replication with Oracle Application Server Cluster (OC4J)" in the *Oracle Application Server High Availability Guide*
Application-level	Web and EJB	`orion-application.xml`	"Stateful Session EJB State Replication with Oracle Application Server Cluster (OC4J)" in the *Oracle Application Server High Availability Guide*
Application-level	EJB	`orion-ejb-jar.xml`	"Overriding Application-Level Replication Policy in the orion-ejb-jar.xml File for EJB Components" on page 24-2

Overriding Application-Level Replication Policy in the orion-ejb-jar.xml File for EJB Components

When you configure the `orion-ejb-jar.xml` file with a state replication policy for a stateful session bean (see Example 24–1), each bean can use a different type of replication independent of the Web component replication type.

Example 24–1 The orion-ejb-jar.xml For an Application-Level Replication Policy for EJB

```
<orion-ejb-jar>
...
    <session-deployment
        name="AirlinePOEndpointBean"
        max-tx-retries="0"
        location="AirlinePOEndpointBean"
        persistence-filename="AirlinePOEndpointBean">
```

```
...
        <replication-policy
            trigger="onRequestEnd"
            scope="allAttributes"
        />
...
    </session-deployment>
...
</orion-ejb-jar>
```

Configuring Static Retrieval Load Balancing

To use static retrieval of OC4J instances for load balancing, do the following:

1. Within each client, configure JNDI properties as follows (see "Using JNDI Properties" on page 24-3:
   - For `java.naming.factory.initial`, use any initial context factory.
   - For the `java.naming.provider.url`, provide a comma-delimited list of OC4J nodes in the form `<prefix>://<hostname>:<port>/<application-name>` where `<prefix>` is `opmn:ormi` for OC4J in Oracle Application Server, or `ormi` for OC4J standalone.
2. Configure load balancing behavior (see "Configuring Load Balancing Behavior" on page 24-4).

For more information, see the following:

- "Load Balancing" on page 2-31
- "Configuring the Naming Provider URL for OC4J and Oracle Application Server" on page 19-20
- "Configuring the Naming Provider URL for OC4J Standalone" on page 19-21

Using JNDI Properties

Example 24–2 shows a URL definition that provides the client container with three OC4J nodes (with hostnames `s1`, `s2`, and `s3` and ports 23791, 23792, and 23793, respectively) to use for load balancing.

Example 24–2 JNDI Properties for Static Retrieval Load Balancing

```
java.naming.factory.initial= oracle.j2ee.rmi.RMIInitialContextFactory
java.naming.provider.url=ormi://s1:23791/ejbs, ormi://s2:23792/ejbs, ormi://s3:23793/ejbs;
java.naming.security.principal=admin
java.naming.security.credentials=welcome
```

Configuring DNS Load Balancing

To use DNS load balancing, do the following:

1. Within DNS, map a single host name to several IP addresses. Each of the port numbers must be the same for each IP address. Set up the DNS server to return the addresses either in a round-robin or random fashion.

   The IP address identifies the OC4J running; the port number is an RMI port number.

2. Turn off DNS caching on the client. For UNIX machines, you must turn off DNS caching as follows:
   a. Kill the NSCD daemon process on the client.
   b. Start the OC4J client with the `-Dsun.net.inetaddr.ttl=0` option.
3. Within each client, configure JNDI properties as follows (see "Using JNDI Properties" on page 24-4):
   - For `java.naming.factory.initial`, use any initial context factory.
   - For the `java.naming.provider.url`, provide single host name to which the OC4J IP addresses are mapped and the common RMI port in the form `<prefix>://<hostname>:<port>/<application-name>` where `<prefix>` is `opmn:ormi` for OC4J in Oracle Application Server or `ormi` for OC4J standalone.
4. Configure load balancing behavior (see "Configuring Load Balancing Behavior" on page 24-4).

Each time the lookup occurs on the DNS server, the DNS server hands back one of the many IP addresses that are mapped to it.

For more information, see the following:

- "Load Balancing" on page 2-31
- "Configuring the Naming Provider URL for OC4J and Oracle Application Server" on page 19-20
- "Configuring the Naming Provider URL for OC4J Standalone" on page 19-21

Using JNDI Properties

In Example 24–3, the initial context factory is `RMIInitialContextFactory` (however, you can use any initial context factory for DNS load balancing), `myserver` is the host name set up in the DNS server for the list of servers, and the RMI port is the default port.

Example 24–3 JNDI Properties for DNS Load Balancing

```
java.naming.factory.initial= oracle.j2ee.rmi.RMIInitialContextFactory
java.naming.provider.url=ormi://myserver/applname
java.naming.security.principal=admin
java.naming.security.credentials=welcome
```

Configuring Load Balancing Behavior

For both EJB 3.0 and EJB 2.1 and for all load-balancing strategies (replication-based, static retrieval, or DNS), you can configure how a client's requests are load-balanced across the OC4J instances in your cluster (see "Using System Properties" on page 24-4).

For more information, see "Load Balancing" on page 2-31.

Using System Properties

In this release, configure the `oracle.j2ee.rmi.loadBalance` system property to specify load balancing in an application cluster.

This system property takes one of the following values:

- `client`–The client interacts with the OC4J process that was initially chosen at the first lookup for the entire conversation (default).
- `context`–The client goes to a new server when a separate context is used (similar to deprecated `dedicated.rmicontext`).
- `lookup`–The client goes to a new (randomly selected) server for every request.

Configure this system property either on the OC4J command line as a `-D` argument or as an environment reference (see "Configuring an Environment Reference to an Environment Variable" on page 19-16). This system property applies to all clients.

25

Configuring Timer Services

This chapter describes the following:

- Configuring an Enterprise Bean With a Java EE Timer
- Configuring an Enterprise Bean With an OC4J Cron Timer
- Troubleshooting Timers

> **Note:** EJB timers are supported only in an OC4J instance that runs on a single JVM (where numprocs=1 in the `<process-set>` element of the `opmn.xml` configuration file).
>
> You can download EJB timer code examples from: `http://www.oracle.com/technology/tech/java/oc4j/demos` and `http://www.oracle.com/technology/tech/java/oc4j/1003/how_to/how-to-ejb-timer.html`.

For more information, see "Understanding EJB Timer Services" on page 2-31.

Configuring an Enterprise Bean With a Java EE Timer

You can configure the following types of enterprise beans to use a Java EE timer:

- EJB 3.0 stateless session beans and message-driven beans.
- EJB 2.1 stateless session beans, entity beans with container-managed persistence, entity beans with bean-managed persistence, and message-driven beans.

To configure an enterprise bean with a Java EE timer, do the following:

1. Acquire the `javax.ejb.TimerService` in one of the following ways:
   - For an EJB 3.0 enterprise bean, use the `@Resource` annotation, as Example 25–1 shows.
   - For an EJB 3.0 or EJB 2.1 enterprise bean, use `EJBContext` or `InitalContext` method `getTimerService`, as Example 25–1 shows.
2. Use `TimerService` method `createTimer` to create the appropriate type of timer (see the `javax.ejb.TimerService` API), as Example 25–1 and Example 25–1 show.

   When you create a `Timer` on an EJB 2.1 entity bean, the container invokes the timeout callback method of that particular entity bean instance identified by its

primary key. Timers created for a particular entity bean are removed when the entity bean is removed.

When you create a `Timer` on any other type of enterprise beans, the container invokes the timeout callback method on any instance of that type in the pooled state.

3. Implement a timeout callback method.

   This method must not be `static` or `final` and must have the following signature:

   ```
   void <METHOD>(Timer timer)
   ```

   You can implement a timeout callback method in one of the following ways:

   - For an EJB 3.0 enterprise bean, annotate any bean method using the `@Timeout` annotation, as Example 25–1 shows.
   - For an EJB 3.0 or EJB 2.1 enterprise bean, implement the `javax.ejb.TimedObject` interface, as Example 25–1 shows.

Example 25–1 Configuring a Java EE Timer on an EJB 3.0 Stateless Session Bean

```
import javax.ejb.Stateless;
import javax.ejb.TimerService;
import javax.ejb.Timeout;

import javax.ejb.Timer;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;

@Stateless;
public class TimerServiceBean implement MyTimerService {
    // injection of TimerService
    @Resource TimerService timerService;

    // implement bean business interface MyTimerService
    @TransactionAttribute(value=TransactionAttributeType.REQUIRES_NEW)
    // default TransactionAttributeType.REQUIRED
    public void createTimer(Serializable timerInfo) {
        timerService.createTimer(timeout, info);
    }

    ...

    // user annotated timeout method
    @Timeout
    @TransactionAttribute(value=TransactionAttributeType.REQUIRES_NEW)
    public void timeoutCallback(Timer timer) {
        ...
    }
}
```

Example 25–2 Configuring a Java EE Timer on an EJB 2.1 Stateless Session Bean

```
import java.io.Serializable;
import java.rmi.RemoteException;

import javax.ejb.EJBContext;
import javax.ejb.EJBException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.ejb.TimedObject;
import javax.ejb.Timer;
```

```
import javax.ejb.TimerService;

class ServiceBean_2_1 implements SessionBean, MyTimerService, TimedObject {
    EJBContext ctx;

    // implement bean business interface MyTimerService
    public void createTimer(long duration, Serializable info) {
        TimerService timerService = ctx.getTimerService();
        timerService.createTimer(duration, info);
    }

    // implement TimedObject
    public void ejbTimeout(Timer timer) {
        System.out.println("Timeout: " + timer.getInfo());
    }

    ...
    // implement SessionBean
    public void setSessionContext(SessionContext ctx) throws EJBException,
                    RemoteException {
        this.ctx = ctx;
    }
}
```

Configuring an Enterprise Bean With an OC4J Cron Timer

You can use an OC4J cron timer with the following:

- EJB 3.0 stateless session beans and message-driven beans;
- EJB 2.1 enterprise beans of any type.

You can schedule a timer to execute regularly at specified intervals. In the UNIX world, these are known as cron timers.

Example 25–3 shows examples of the different methods you can use in scheduling a cron timer. Where there is an asterisk, all values are valid.

Example 25–3 How to Configure Different Cron Timers

```
20  * * * * --> 20 minutes after every hour, such as 00:20, 01:20, and so on
 5 22 * * * --> Every day at 10:05 P.M.
 0  8 1 * * --> First day of every month at 8:00 A.M.
 0  8 4 7 * --> The fourth of July at 8:00 A.M.
15 12 * * 5 --> Every Friday at 12:15 P.M.
```

The format of a cron time variable includes five time fields, as follows:

- Minute: 0-59
- Hour: 0-23
- Day of the Month: 1-31
- Month: 1-12 or specify with the following strings: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
- Day of the Week: 0-7 or with the following strings: Sun, Mon, Tue, Wed, Thu, Fri, Sat. Both 0 and 7 signify Sunday.

You can define complex timers by specifying multiple values in a field, separated by commas or a dash, as Example 25–4 shows.

Example 25–4 Complex Cron Timers

```
0  8   *   * 1,3,5 --> Every Monday, Wednesday, and Friday at 8:00 A.M.
0  8  1,15 *  *   --> The first and 15th of every month at 8:00 A.M.
0 8-17 *   * 1-5  --> Every hour from 8 A.M. through 5 P.M., Monday through Friday
```

To configure an enterprise bean with an OC4J cron timer, do the following:

1. Acquire the `oracle.j2ee.ejb.timer.EJBTimerService` in one of the following ways:
   - For an EJB 3.0 enterprise bean, use the `@Resource` annotation, as Example 25–5 shows.
   - For an EJB 3.0 or EJB 2.1 enterprise bean, use `EJBContext` or `InitalContext` method `getTimerService`, as Example 25–6 shows.
2. Use `EJBTimerService` method `createTimer` to create the appropriate type of timer, as Example 25–5 and Example 25–6 show.

   You can use any of the following `EJBTimerService` methods, all of which return an object of type `javax.ejb.Timer` and throw `IllegalArgumentException` and `IllegalStateException`:

   - `createTimer(String cronline, Serializable info)`

     Use this method to create an OC4J cron timer that invokes a timeout callback method on the bean by passing in a `String` cron line, as the following example shows. Use the `info` argument to pass application information to OC4J: this can be `null`:

     ```
     ...
     import oracle.j2ee.ejb.timer.EJBTimerService;
     import javax.ejb.timer.Timer;
     ...
     @Resource EJBTimerService ets;

     String cron = "1 * * * *";
     String info = "";
     Timer et = ets.createTimer(cron, info);
     ...
     ```

   - `createTimer(int minute, int hour, int dayOfMonth, int month, int dayOfWeek, Serializable info)` or

     `createTimer(int minute, int hour, int dayOfMonth, int month, int dayOfWeek, int year, Serializable info)`

     Use this method to create an OC4J cron timer that invokes a timeout callback method on the bean by passing each cron field as a separate argument, as the following example shows. Use the `info` argument to pass application information to OC4J: this can be `null`:

     ```
     ...
     import oracle.j2ee.ejb.timer.EJBTimerService;
     import javax.ejb.timer.Timer;
     ...
     @Resource EJBTimerService ets;

     int min=15;  // minutes
     int hr=13;   // hour (1 PM)
     int dom=28;  // day of month
     int mo=1;    // month (January)
     int dow=3;   // day of week (Wednesday)
     ```

```
String info = "";
Timer et = ets.createTimer(min, hr, dom, mo, dow, info);
...
```

- `createTimer(String cronline, String className, Serializable info)`

  Use this method to create an OC4J cron timer that invokes the specified Java class's `main` method by passing in a `String` cron line, as the following example shows. The `info` argument can be either `null` or a `String[]` of parameters to pass to the `main` method of the class:

```
...
import mypackage.MyClass;
import oracle.j2ee.ejb.timer.EJBTimerService;
import javax.ejb.timer.Timer;
...
@Resource EJBTimerService ets;

String cron = "1 * * * *";
String info = "";
Timer et = ets.createTimer(cron, MyClass.class.getName(), info);
...
```

- `createTimer(int minute, int hour, int dayOfMonth, int month, int dayOfWeek, String className, Serializable info)` or

  `createTimer(int minute, int hour, int dayOfMonth, int month, int dayOfWeek, int year, String className, Serializable info)`

  Use this method to create an OC4J cron timer that invokes the specified Java class's `main` method by passing each cron field as a separate argument, as the following example shows. The `info` argument can be either `null` or a `String[]` of parameters to pass to the `main` method of the class:

```
...
import mypackage.MyClass;
import oracle.j2ee.ejb.timer.EJBTimerService;
import javax.ejb.timer.Timer;
...
@Resource EJBTimerService ets;

int min=15;  // minutes
int hr=13;   // hour (1 PM)
int dom=28;  // day of month
int mo=1;    // month (January)
int dow=3;   // day of week (Wednesday)
String info = "";
Timer et = ets.createTimer(min, hr, dom, mo, dow,
                           MyClass.class.getName(), info);
...
```

When you create a `Timer` on an EJB 2.1 entity bean, the container invokes the timeout callback method of that particular entity bean instance identified by its primary key. Timers created for a particular entity bean are removed when the entity bean is removed.

When you create a `Timer` on any other type of EJB, the container invokes the timeout callback method on any instance of that type in the pooled state.

3. Complete the configuration depending on what action OC4J takes when the timer fires, as follows:

   a. If you created a timer using a `createTimer` method that does not take a `Class`, OC4J invokes a timeout callback method when the timer fires.

      The timeout callback method must not be `static` or `final`, and must have the following signature:

      ```
      void <METHOD>(Timer timer)
      ```

      You can implement a timeout callback method in one of the following ways:

      – For an EJB 3.0 enterprise bean, annotate any bean method using the `@Timeout` annotation, as Example 25–5 shows.

      – For an EJB 3.0 or EJB 2.1 enterprise bean, implement the `javax.ejb.TimedObject` interface, as Example 25–6 shows.

   b. If you created a timer using a `createTimer` method that takes a `Class`, OC4J invokes the `main` method of the `Class` that you specify when the timer fires.

      The `main` method must have the following signature:

      ```
      public static void main(String args[])
      ```

Example 25–5 Configuring an OC4J Cron Timer on an EJB 3.0 Stateless Session Bean

```
import javax.ejb.Stateless;
import javax.annotation.PostConstruct;
import oracle.j2ee.ejb.timer.EJBTimerService;
import javax.ejb.Timer;
import javax.ejb.Timeout;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;

@Stateless
public class MySession {
    @PostConstruct
    @TransactionAttribute(value=REQUIRES_NEW)
    public void initialize() {
        @Resource EJBTimerService ets;

        String cron = "1 * * * *";
        String info = "";
        Timer et = ets.createTimer(cron, info);
    }

    ...

    @Timeout
    @TransactionAttribute(value=REQUIRES_NEW)
    public void timeoutCallback(Timer timer) {
    ...
    }
}
```

Example 25–6 Configuring an OC4J Cron Timer on an EJB 2.1 Stateless Session Bean

```
import javax.ejb.SessionBean;
import oracle.j2ee.ejb.timer.EJBTimerService;
import javax.ejb.TimedObject;

public class MySession implements SessionBean, TimedObject {
```

```
    public void initialize() {
        String cron = "1 * * * *";
        String info = "";
        InitialContext ctx = new InitialContext();
        EJBTimerService ets = (EJBTimerService) ctx.getTimerService();
        Timer et = ets.createTimer(cron, info);
    }
    ...
    public void ejbTimeout(Timer timer) {
    ...
    }
}
```

Troubleshooting Timers

This section describes the following:

- Retrieving Information About a Timer
- Retrieving a Persisted Timer
- Executing a Timer Within the Scope of a Transaction
- What Does a NoSuchObjectLocalException Mean With Timers?

Retrieving Information About a Timer

You can retrieve information and cancel the timer through the `Timer` object. The methods available are `cancel`, `getTimeRemaining`, `getNextTimeout`, `getHandle`, and `getInfo`. To compare for object equality, use the `Timer.equals(Object obj)` method.

Retrieving a Persisted Timer

Timers must be able to be persisted so that they can survive the life cycle of the bean (`ejbLoad`, `ejbStore`, and so on). You can retrieve a persisted `Timer` object through its handle. Retrieve the `TimerHandle` through the `Timer.getHandle` method. Then, you can retrieve the persisted `Timer` object through the `TimerHandle.getTimer` method.

Note: Timers and their handles are local objects; therefore, try not to pass them through the bean remote interface.

Executing a Timer Within the Scope of a Transaction

You usually create and cancel a timer within the scope of a transaction. Thus, you usually configure the bean as being within a transaction by using `RequiresNew`. If the transaction is rolled back, then the container retries the timeout.

For more information on transactions, see the *Oracle Containers for J2EE Services Guide*.

What Does a NoSuchObjectLocalException Mean With Timers?

When you try to invoke a method on a timer object that has been either successfully invoked or cancelled, you will receive a `NoSuchObjectLocalException`.

Part IX

Packaging and Deploying an EJB Application

This part provides procedural information on packaging and deploying a J2EE application using EJB 3.0 or EJB 2.1 enterprise JavaBeans. For conceptual information, see Part I, "EJB Overview".

This part contains the following chapters:

- Chapter 26, "Configuring Deployment Descriptor Files"
- Chapter 27, "Packaging an EJB Application"
- Chapter 28, "Deploying an EJB Application to OC4J"

26

Configuring Deployment Descriptor Files

This chapter describes how to configure the various deployment descriptor files that an OC4J application may use, including the following:

- Configuring the ejb-jar.xml File
- Configuring the toplink-ejb-jar.xml File
- Configuring the orion-ejb-jar.xml File
- Configuring the ejb3-toplink-sessions.xml File
- Configuring the persistence.xml File

For more information, see "Understanding EJB Deployment Descriptor Files" on page 2-4.

Configuring the ejb-jar.xml File

This section describes the following:

- Creating ejb-jar.xml During Migration
- Creating the ejb-jar.xml File at Deployment Time
- Creating ejb-jar.xml With JDeveloper

For more information, see "What is the ejb-jar.xml File?" on page 2-5.

Creating ejb-jar.xml During Migration

For EJB 2.1 only, you can automatically generate the `ejb-jar.xml` file during migration (see "Migrating to the TopLink EJB 2.1 Persistence Manager" on page 3-13). After generation, you can use the TopLink Workbench to customize and reexport this file (see "Using TopLink Workbench" on page 2-2).

Creating the ejb-jar.xml File at Deployment Time

When you deploy an EJB 3.0 application with one or more annotations, OC4J will write its in-memory `ejb-jar.xml` file to the same location as the `orion-ejb-jar.xml` file in the deployment directory: `<`*`ORACLE_HOME`*`>/j2ee/home/application-deployments/my_application/META-INF`.

This `ejb-jar.xml` file represents configuration obtained from both annotations and a deployed `ejb-jar.xml` file (if present).

Creating ejb-jar.xml With JDeveloper

You can use JDeveloper to generate and update the `ejb-jar.xml` file.

For more information, see "Using JDeveloper" on page 2-1.

Configuring the toplink-ejb-jar.xml File

The `toplink-ejb-jar.xml` file is applicable only if you are using the TopLink JPA preview persistence provider.

> **Note:** By default, OC4J uses the TopLink Essentials JPA persistence provider. In this case, you can configure TopLink descriptor-level options (including mappings) using TopLink JPA extensions ("Accessing TopLink API at Run Time With TopLink Essentials JPA Persistence" on page 3-4).

This section describes the following:

- Creating toplink-ejb-jar.xml During Migration
- Creating toplink-ejb-jar.xml With TopLink Workbench

For more information, see the following:

- "What is the toplink-ejb-jar.xml File?" on page 2-6
- "OC4J and the toplink-ejb-jar.xml File" in the *Oracle TopLink Developer's Guide*

Creating toplink-ejb-jar.xml During Migration

For EJB 2.1 projects only, when you migrate an Orion CMP application to TopLink persistence (see "Migrating to the TopLink EJB 2.1 Persistence Manager" on page 3-13), the TopLink migration tool automatically creates a `toplink-ejb-jar.xml` file for you.

After generation, you can use the TopLink Mapping Workbench to customize and reexport (see "Understanding the TopLink Workbench" in the *Oracle TopLink Developer's Guide*).

Creating toplink-ejb-jar.xml With TopLink Workbench

For EJB 3.0 projects, if the only JDK 1.5 language extension that your entity classes use are annotations, you can use the TopLink Workbench to create and configure a `toplink-ejb-jar.xml` file. Oracle recommends using the TopLink Workbench to create and configure this file.

For EJB 2.1 projects, you use the TopLink Workbench to configure persistence properties in the `toplink-ejb-jar.xml` file. When you migrate an Orion CMP application to TopLink persistence (see "Migrating to the TopLink EJB 2.1 Persistence Manager" on page 3-13), the TopLink migration tool automatically creates a TopLink Workbench project for you. You can use the TopLink Workbench project to create a `toplink-ejb-jar.xml` file.

For more information, see the following:

- "Understanding the TopLink Workbench" in the *Oracle TopLink Developer's Guide*
- "Creating project.xml with TopLink Workbench" in the *Oracle TopLink Developer's Guide*.

Configuring the orion-ejb-jar.xml File

To specify OC4J-proprietary options, you can create an `orion-ejb-jar.xml` file and configure the appropriate elements:

- "<session-deployment>" on page A-4
- "<entity-deployment>" on page A-10
- "<message-driven-deployment>" on page A-17

For more information, see "What is the orion-ejb-jar.xml File?" on page 2-6.

Note: Alternatively, in an EJB 3.0 application, you can use OC4J-proprietary annotations for session bean and message-driven beans. Vendor extensions set in the `orion-ejb-jar.xml` file override extensions set using OC4J-proprietary annotations.

For more information, see the following:

- "Configuring OC4J-Proprietary Deployment Options on an EJB 3.0 Session Bean" on page 5-10
- "Configuring OC4J-Proprietary Deployment Options on an EJB 3.0 MDB" on page 10-17

Configuring the ejb3-toplink-sessions.xml File

The `ejb3-toplink-sessions.xml` file is applicable only if you are using the TopLink JPA preview persistence provider.

Note: By default, OC4J uses the TopLink Essentials JPA persistence provider. In this case, you can configure TopLink session-level options using TopLink JPA extensions ("Accessing TopLink API at Run Time With TopLink Essentials JPA Persistence" on page 3-4).

This section describes the following:

- "Creating ejb3-toplink-sessions.xml With TopLink Workbench"

For more information, see "What is the ejb3-toplink-sessions.xml File?" on page 2-7.

Creating ejb3-toplink-sessions.xml With TopLink Workbench

For EJB 3.0 applications, if the only JDK 1.5 language extension that your entity classes use are annotations, you can use the TopLink Workbench to create and configure a `ejb3-toplink-sessions.xml` file. Oracle recommends using the TopLink Workbench to create and configure this file.

For more information, see the following:

- "Understanding the TopLink Workbench" in the *Oracle TopLink Developer's Guide*
- "Creating project.xml with TopLink Workbench" in the *Oracle TopLink Developer's Guide*.

Configuring the persistence.xml File

This section describes the following:

- Configuring the persistence.xml With a Named Persistence Unit File
- Configuring the persistence.xml File for the OC4J Default Persistence Unit
- Specifying a Data Source in a Persistence Unit
- Configuring Vendor Extensions in a Persistence Unit

For more information, see "What is the persistence.xml File?" on page 2-8.

Configuring the persistence.xml With a Named Persistence Unit File

Example 26–1 shows an example `persistence.xml` file that contains one persistence unit.

Example 26–1 Named Persistence Unit

```
<persistence-unit name="OrderManagement5">
    <provider>com.acme.persistence</provider>
    <transaction-type>RESOURCE_LOCAL</transaction-type>
    <mapping-file>order1.xml</mapping-file>
    <jar-file>order.jar</jar-file>
    <class>com.acme.Order</class>
    <properties>
        <property name="com.acme.persistence.sql-logging" value="on"/>
    </properties>
</persistence-unit>
```

This persistence unit is named `OrderManagement5` and uses `EntityManager` provider `com.acme.persistence`. Its `<transaction-type>` specifies that this persistence unit requires only a non-JTA data source. It defines its set of persistent managed classes using all of `<mapping-file>`, `<jar-file>`, and `<class>` elements (see "What Persistent Managed Classes Does This Persistence Unit Include?" on page 26-4). It sets property `com.acme.persistence.sql.logging` to a value of `on` using a `<property>` element.

For detailed descriptions of `<persistence-unit>` element attributes and subelements, see the EJB 3.0 specification.

What Persistent Managed Classes Does This Persistence Unit Include?

You can specify the persistent managed classes associated with a persistence unit by using one or more of the following:

- `<mapping-file>` element: specifies one or more object-relational mapping XML files (`orm.xml` files).
- `<jar-file>` element: specifies one or more JAR files that will be searched for classes.
- `<class>` element: specifies an explicit list of classes.
- The annotated managed persistence classes contained in the root of the persistence unit.

  The root of the persistence unit is the JAR file or directory, whose `META-INF` directory contains the `persistence.xml` file. To exclude managed persistence classes, add an `<exclude-unlisted-classes>` element to the persistence unit.

Configuring the persistence.xml File for the OC4J Default Persistence Unit

Using the OC4J default persistence unit, you can acquire an entity manager without having to specify a persistence unit by name (see "Understanding OC4J Persistence Unit Defaults" on page 2-8).

By default, to use the OC4J default persistence unit, you do not need to deploy a `persistence.xml` file at all.

If you set `orion-ejb-jar.xml` file attribute `disable-default-persistent-unit` to `true`, OC4J will expect a `persistence.xml` file. In this case, you can still use the OC4J default persistence unit if you specify an empty persistence unit: configure your `persistence.xml` file with an empty persistence unit using any of the following:

- Empty `<persistence>` element:

```
<persistence>
</persistence>
```

- Self-closing `<persistence/>` element
- Completely empty (zero length) `persistence.xml` file

You may specify one persistence unit for each scope or module: for example, one for each EJB JAR.

Specifying a Data Source in a Persistence Unit

In a Java EE application, you specify your data source in a `<jta-data-source>` element as Example 26–2 shows. For more information, see "Configuring Data Sources" on page 20-1.

In a Java SE application, you specify your data source using JDBC vendor extensions, as Example 26–3 shows. For more information, see "TopLink JPA Extensions for JDBC (Java SE)" on page 26-7).

Alternatively, using OC4J, you can use the default data source (see "What is a Default Data Source?" on page 2-16).

Configuring Vendor Extensions in a Persistence Unit

This section describes the TopLink JPA vendor extensions that you can define in a persistence unit, including the following:

- TopLink JPA Extensions for JDBC (Java SE)
- TopLink JPA Extensions for Caching
- TopLink JPA Extensions for Logging
- TopLink JPA Extensions for Database, Session, and Application Server
- TopLink JPA Extensions for Customization
- TopLink JPA Extensions for Schema Generation

You can specify these vendor extensions by using a `<properties>` element in your `persistence.xml` file. Example 26–2 shows how to set a TopLink JPA persistence unit extension in a `persistence.xml` file for a Java EE application, and Example 26–3 shows how to do the same for a Java SE application.

Example 26–2 Configuring a Vendor Extension in the Persistence.xml File (Java EE)

```
<persistence-unit name="default" transaction-type="JTA">
    <provider>
        oracle.toplink.essentials.PersistenceProvider
    </provider>
    <jta-data-source>
        jdbc/MyDataSource
    </jta-data-source>
    <properties>
        <property name="toplink.logging.level" value="INFO"/>
    </properties>
</persistence-unit>
```

Example 26–3 Configuring a Vendor Extension in the Persistence.xml File (Java SE)

```
<persistence-unit name="default" transaction-type="RESOURCE_LOCAL">
    <provider>
        oracle.toplink.essentials.PersistenceProvider
    </provider>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
        <property name="toplink.logging.level" value="INFO"/>
        <property name="toplink.jdbc.driver" value="oracle.jdbc.OracleDriver"/>
        <property name="toplink.jdbc.url" value="jdbc:oracle:thin:@myhost:1521:MYSID"/>
        <property name="toplink.jdbc.password" value="tiger"/>
        <property name="toplink.jdbc.user" value="scott"/>
    </properties>
</persistence-unit>
```

Alternatively, you can set a TopLink JPA persistence unit extension in the `Map` of properties you pass into a call to `javax.persistence.Persistence` method `createEntityManagerFactory` as Example 26–4 shows. You can override extensions set in the `persistence.xml` file in this way. When you set an extension in a `Map` of properties, you can set the value using the `public static final` field in the appropriate configuration class in `oracle.toplink.essentials.config`, including the following:

- `CacheType`
- `TargetDatabase`
- `TargetServer`
- `TopLinkProperties`

> **Note:** To access these classes, ensure that the appropriate OC4J persistence JAR is in your classpath. For more information, see "TopLink Essentials JPA Persistence Provider" on page 3-2.

Example 26–4 shows how to set the value of extension `toplink.cache.type.default` using the `CacheType` configuration class.

Example 26–4 Configuring a Vendor Extension When Creating an EntityManagerFactory

```
import oracle.toplink.essentials.config.CacheType;

Map properties = new HashMap();
properties.put(TopLinkProperties.CACHE_TYPE_DEFAULT, CacheType.Full);
EntityManagerFactory emf = Persistence.createEntityManagerFactory("default", properties);
```

TopLink JPA Extensions for JDBC (Java SE)

Table 26–1 lists the TopLink JPA extensions that you can define in a `persistence.xml` file to configure JDBC driver parameters. These extensions apply only when used outside of a EJB container.

Table 26–1 TopLink JPA Extensions for JDBC (Java SE)

Property	Usage	Default

`toplink.jdbc.bind-parameters`	Control whether or not the query uses parameter binding.
	For more information, see "Using Conforming Queries and Descriptors" in the Oracle TopLink Developer's Guide.
	Valid values:
	▪ `true`–bind all parameters.
	▪ `false`–do not bind parameters.
	Example: `persistence.xml` file
	`<property name="toplink.jdbc.bind-parameters" value="true"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.JDBC_BIND_PARAMETERS, "true");`
		`true`

`toplink.jdbc.driver`	The class name of the JDBC driver you want to use, fully qualified by its package name. This class must be on your application classpath.
	Example: `persistence.xml` file
	`<property name="toplink.jdbc.driver" value="oracle.jdbc.driver.OracleDriver"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.JDBC_DRIVER, "oracle.jdbc.driver.OracleDriver");`	

`toplink.jdbc.password`	The password for your JDBC user.
	Example: `persistence.xml` file
	`<property name="toplink.jdbc.password" value="tiger"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.JDBC_PASSWORD, "tiger");`	

`toplink.jdbc.read-connections.max`	The maximum number of connections allowed in the JDBC read connection pool.
	Valid values: `0` to `Integer.MAX_VALUE` (depending on your JDBC driver) as a `String`.
	Example: `persistence.xml` file
	`<property name="toplink.jdbc.read-connections.max" value="3"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.JDBC_READ_CONNECTIONS_MAX, "3");`
		2

Table 26–1 (Cont.) TopLink JPA Extensions for JDBC (Java SE)

Property	Usage	Default

`toplink.jdbc.read-connections.min`	The minimum number of connections allowed in the JDBC read connection pool.
	Valid values: `0` to `Integer.MAX_VALUE` (depending on your JDBC driver) as a `String`.
	Example: `persistence.xml` file
	`<property name="toplink.jdbc.read-connections.min" value="1"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.JDBC_READ_ CONNECTIONS_MIN, "1");`
		2

`toplink.jdbc.read-connections.shared`	Specify whether or not to allow concurrent use of shared read connections.
	Valid values:
	■ `true`–allow concurrent use of shared read connections.
	■ `false`–do not allow the concurrent use of shared read connections; concurrent readers are each allocated their own read connection.
	Example: `persistence.xml` file
	`<property name="toplink.jdbc.read-connections.shared" value="true"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.JDBC_READ_ CONNECTIONS_SHARED, "true");`
		`false`

`toplink.jdbc.url`	The JDBC connection URL required by your JDBC driver.
	Example: `persistence.xml` file
	`<property name="toplink.jdbc.url" value="jdbc:oracle:thin:@MYHOST:1521:MYSID"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.JDBC_URL, "jdbc:oracle:thin:@MYHOST:1521:MYSID");`	

Table 26–1 (Cont.) TopLink JPA Extensions for JDBC (Java SE)

Property	Usage	Default

`toplink.jdbc.user`	The user name for your JDBC user.
	Example: `persistence.xml` file
	`<property name="toplink.jdbc.user" value="scott"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.JDBC_USER, "scott");`	

`toplink.jdbc.write-connections.max`	The maximum number of connections allowed in the JDBC write connection pool.
	Valid values: `0` to `Integer.MAX_VALUE` (depending on your JDBC driver) as a `String`.
	Example: `persistence.xml` file
	`<property name="toplink.jdbc.write-connections.max" value="5"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.JDBC_WRITE_ CONNECTIONS_MAX, "5");`
		10

`toplink.jdbc.write-connections.min`	The minimum number of connections allowed in the JDBC write connection pool.
	Valid values: `0` to `Integer.MAX_VALUE` (depending on your JDBC driver) as a `String`.
	Example: `persistence.xml` file
	`<property name="toplink.jdbc.write-connections.min" value="2"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.JDBC_WRITE_ CONNECTIONS_MIN, "2");`
		5

TopLink JPA Extensions for Caching

Table 26–2 lists the TopLink JPA extensions that you can define in a `persistence.xml` file to configure the TopLink cache.

For more information, see "Understanding the Cache" in the *Oracle TopLink Developer's Guide*.

Table 26–2 TopLink JPA Extensions for Caching

Property	Usage	Default

`toplink.cache.type.default`	The default type of session cache.
	A session cache is a shared cache that services clients attached to a given session. When you read objects from or write objects to the data source using a client session, TopLink saves a copy of the objects in the parent server session's cache and makes them accessible to child client sessions.
	Valid values: `oracle.toplink.essentials.config.CacheType`
	■ `Full`–This option provides full caching and guaranteed identity: objects are never flushed from memory unless they are deleted.
	For more information, see "Full Identity Map" in the Oracle TopLink Developer's Guide.
	■ `HardWeak`–This option is similar to `Weak`, except that it maintains a most frequently used subcache that uses hard references.
	For more information, see "Soft and Hard Cache Weak Identity Maps" in the Oracle TopLink Developer's Guide.
	■ `NONE`–This option does not preserve object identity and does not cache objects. Oracle does not recommend using this option.
	For more information, see "No Identity Map" in the Oracle TopLink Developer's Guide.
	■ `SoftWeak`–This option is similar to `Weak`, except that it maintains a most frequently used subcache that uses soft references. Oracle recommends using this identity map in most circumstances as a means to control memory used by the cache.
	For more information, see "Soft and Hard Cache Weak Identity Maps" in the Oracle TopLink Developer's Guide.
	■ `Weak`–This option is similar to `Full`, except that objects are referenced using weak references. This option uses less memory than `Full`, but does not provide a durable caching strategy across client/server transactions. Oracle recommends using this identity map for transactions that, once started, stay on the server side.
	For more information, see "Weak Identity Map" in the Oracle TopLink Developer's Guide.
	Example: `persistence.xml` file
	`<property name="toplink.cache.type.default" value="Full"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.CacheType;`
	`import oracle.toplink.essentials.config.TopLinkProperties;`
	`propertiesMap.put(ToplinkProperties.CACHE_TYPE_DEFAULT, CacheType.Full);`
		`SoftWeak`

`toplink.cache.size.default`	The default maximum number of objects allowed in a TopLink cache.
	Valid values: `0` to `Integer.MAX_VALUE` as a `String`.
	Example:
	`<property name="toplink.cache.size.default" value="5000"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TopLinkProperties;`
	`propertiesMap.put(ToplinkProperties.CACHE_SIZE_DEFAULT, 1000);`
		1000

Table 26–2 (Cont.) TopLink JPA Extensions for Caching

Property	Usage	Default

`toplink.cache.shared.default`	The default for whether or not the TopLink session cache is shared by multiple client sessions.
	Valid values:
	■ `true`–The session cache services all clients attached to the session. When you read objects from or write objects to the data source using a client session, TopLink saves a copy of the objects in the parent server session's cache and makes them accessible to all other processes in the session.
	■ `false`–The session cache services a single, isolated client exclusively. The isolated client can reference objects in a shared session cache but no client can reference objects in the isolated client's exclusive cache.
	Example:
	`<property name="toplink.cache.shared.default" value="true"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TopLinkProperties;`
	`propertiesMap.put(ToplinkProperties.CACHE_SHARED_DEFAULT, "true");`
		`true`

Table 26–2 (Cont.) TopLink JPA Extensions for Caching

Property	Usage	Default

`toplink.cache.type.<ENTITY>`	The type of session cache for the JPA entity named `<ENTITY>`.
	For more information on entity names, see `@Entity`.
	Valid values: `oracle.toplink.essentials.config.CacheType`
	▪ `Full`–see `toplink.cache.type.default`.
	▪ `HardWeak`–see `toplink.cache.type.default`.
	▪ `NONE`–see `toplink.cache.type.default`.
	▪ `SoftWeak`–see `toplink.cache.type.default`.
	▪ `Weak`–see `toplink.cache.type.default`.
	Example: `persistence.xml` file
	`<property name="toplink.cache.type.Order" value="Full"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.CacheType`
	`import oracle.toplink.essentials.config.TopLinkProperties;`
	`propertiesMap.put(TopLinkProperties.CACHE_TYPE+".Order", CacheType.Full);`
		`SoftWeak`

`toplink.cache.size.<ENTITY>`	The maximum number of JPA entities of the type denoted by JPA entity name `<ENTITY>` allowed in a TopLink cache.
	For more information on entity names, see `@Entity`.
	Valid values: `0` to `Integer.MAX_VALUE` as a `String`.
	Example:
	`<property name="toplink.cache.size.Order" value="5000"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TopLinkProperties;`
	`propertiesMap.put(ToplinkProperties.CACHE_SIZE+".Order", 1000);`
		1000

`toplink.cache.shared.<ENTITY>`	Whether or not the TopLink session cache is shared by multiple client sessions for JPA entities of the type denoted by JPA entity name `<ENTITY>`.
	For more information on entity names, see `@Entity`.
	Valid values:
	▪ `true`–The session cache services all clients attached to the session. When you read objects from or write objects to the data source using a client session, TopLink saves a copy of the objects in the parent server session's cache and makes them accessible to all other processes in the session.
	▪ `false`–The session cache services a single, isolated client exclusively. The isolated client can reference objects in a shared session cache but no client can reference objects in the isolated client's exclusive cache.
	Example:
	`<property name="toplink.cache.shared.Order" value="true"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TopLinkProperties;`
	`propertiesMap.put(ToplinkProperties.CACHE_SHARED+".Order", "true");`
		`true`

TopLink JPA Extensions for Logging

Table 26–3 lists the TopLink JPA extensions that you can define in a `persistence.xml` file to configure TopLink logging.

For more information, see "Configuring Logging" in the *Oracle TopLink Developer's Guide*.

Table 26–3 TopLink JPA Extensions for Logging

Property	Usage	Default

`toplink.logging.level`	Control the amount and detail of log output by configuring the log level (in ascending order of information):
	Valid values: `java.util.logging.Level`
	■ `OFF`–disable logging
	■ `Level.SEVERE`–Logs exceptions indicating TopLink cannot continue, as well as any exceptions generated during login. This includes a stack trace.
	■ `WARNING`–Logs exceptions that do not force TopLink to stop, including all exceptions not logged with severe level. This does not include a stack trace.
	■ `INFO`–Logs the login/logout for each server session, including the user name. After acquiring the session, detailed information is logged.
	■ `CONFIG`–Logs only login, JDBC connection, and database information.
	■ `FINE`–Logs SQL.
	■ `FINER`–Similar to warning. Includes stack trace.
	■ `FINEST`–Includes additional low level information.
	Example: `persistence.xml` file
	`<property name="toplink.logging.level" value="WARNING"/>`
	Example: property `Map`
	`import java.util.logging.Level;`
	`import oracle.toplink.essentials.config.TopLinkProperties;`
	`propertiesMap.put(TopLinkProperties.LOGGING_LEVEL, Level.INFO);`
		`CONFIG`

`toplink.logging.timestamp`	Control whether the timestamp is logged in each log entry.
	Valid values:
	■ `true`–log a timestamp.
	■ `false`–do not log a timestamp.
	Example: `persistence.xml` file
	`<property name="toplink.logging.timestamp" value="true"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TopLinkProperties;`
	`propertiesMap.put(TopLinkProperties.LOGGING_TIMESTAMP, "true");`
		`true`

Table 26–3 (Cont.) TopLink JPA Extensions for Logging

Property	Usage	Default

`toplink.logging.thread`	Control whether a thread identifier is logged in each log entry.
	Valid values:
	▪ `true`–log a thread identifier.
	▪ `false`–do not log a thread identifier.
	Example: `persistence.xml` file
	`<property name="toplink.logging.thread" value="true"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TopLinkProperties;`
	`propertiesMap.put(TopLinkProperties.LOGGING_THREAD, "true");`
		`true`

`toplink.logging.session`	Control whether a TopLink session identifier is logged in each log entry.
	Valid values:
	▪ `true`–log a TopLink session identifier.
	▪ `false`–do not log a TopLink session identifier.
	Example: `persistence.xml` file
	`<property name="toplink.logging.session" value="true"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TopLinkProperties;`
	`propertiesMap.put(TopLinkProperties.LOGGING_SESSION, "true");`
		`true`

`toplink.logging.exceptions`	Control whether the exceptions thrown from within the TopLink code are logged prior to returning the exception to the calling application. Ensures that all exceptions are logged and not masked by the application code.
	Valid values:
	▪ `true`–log all exceptions.
	▪ `false`–do not log exceptions.
	Example: `persistence.xml` file
	`<property name="toplink.logging.exceptions" value="true"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TopLinkProperties;`
	`propertiesMap.put(TopLinkProperties.LOGGING_EXCEPTIONS, "true");`
		`false`

TopLink JPA Extensions for Database, Session, and Application Server

Table 26–4 lists the TopLink JPA extensions that you can define in a `persistence.xml` file to configure TopLink extensions for database, session, and application server.

Table 26–4 TopLink JPA Extensions for Database, Session, and Application Server

Property	Usage	Default

`toplink.target-database`	Specify the type of database that your JPA application uses. The `TargetDatabase` enum contains an entry for many of the more common database types supported.
	Valid values: `oracle.toplink.essentials.config.TargetDatabase`
	■ `Attunity`–configure the persistence provider to use an Attunity database.
	■ `Auto`–TopLink accesses the database and uses the metadata that JDBC provides to determine the target database. Applicable to JDBC drives that support this metadata.
	■ `Cloudscape`–configure the persistence provider to use a Cloudscape database.
	■ `Database`–configure the persistence provider to use a generic choice if your target database is not listed here and your JDBC driver does not support the use of metadata that the `Auto` option requires.
	■ `DB2`–configure the persistence provider to use a DB2 database.
	■ `DB2Mainframe`–configure the persistence provider to use a DB2Mainframe database.
	■ `DBase`–configure the persistence provider to use a DBase database.
	■ `Derby`–configure the persistence provider to use a Derby database.
	■ `HSQL`–configure the persistence provider to use an HSQL database.
	■ `Informix`–configure the persistence provider to use an Informix database.
	■ `JavaDB`–configure the persistence provider to use a JavaDB database.
	■ `MySQL4`–configure the persistence provider to use a MySQL4 database.
	■ `Oracle`–configure the persistence provider to use an Oracle database.
	■ `PointBase`–configure the persistence provider to use a PointBase database.
	■ `PostgreSQL`–configure the persistence provider to use a PostgreSQL database.
	■ `SQLAnyWhere`–configure the persistence provider to use an SQLAnyWhere database.
	■ `SQLServer`–configure the persistence provider to use an SQLServer database.
	■ `Sybase`–configure the persistence provider to use a Sybase database.
	■ `TimesTen`–configure the persistence provider to use a TimesTen database.
	Example: `persistence.xml` file
	`<property name="toplink.target-database" value="Oracle"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TargetDatabase;`
	`import oracle.toplink.essentials.config.TopLinkProperties;`
	`propertiesMap.put(TopLinkProperties.TARGET_DATABASE,`
	`TargetDatabase.Oracle);`
		`Auto`

`toplink.session-name`	Specify the name by which the TopLink session is stored in the static session manager. Use this option if you need to access the TopLink shared session outside of the context of the Java Persistence API.
	Valid values: a valid TopLink session name that is unique in a server deployment.
	Example: `persistence.xml` file
	`<property name="toplink.session-name" value="MySession"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TopLinkProperties;`
	`propertiesMap.put(TopLinkProperties.SESSION_NAME, "MySession");`
		TopLink generated unique name.

Table 26–4 (Cont.) TopLink JPA Extensions for Database, Session, and Application Server

Property	Usage	Default

`toplink.target-server`	Specify the type of application server that your JPA application uses:
	Valid values: oracle.toplink.essentials.config.TargetServer
	▪ `None`–configure the persistence provider to use no application server.
	▪ `OC4J_10_1_3`–configure the persistence provider to use OC4J 10.1.3.0.
	▪ `SunAS9`–configure the persistence provider to use Sun Application Server version 9.
	Example: `persistence.xml` file
	`<property name="toplink.target-server" value="OC4J_10_1_3"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TargetServer;`
	`import oracle.toplink.essentials.config.TopLinkProperties;`
	`propertiesMap.put(TopLinkProperties.TARGET_SERVER,`
	`TargetServer.OC4J_10_1_3);`
		`None`

TopLink JPA Extensions for Customization

Table 26–5 lists the TopLink JPA extensions that you can define in a `persistence.xml` file to configure TopLink customization and validation.

Table 26–5 TopLink JPA Extensions for Customization and Validation

Property	Usage	Default

`toplink.weaving`	Control whether or not the weaving of the entity classes is performed. Weaving is required in order to use lazy fetching of `@OneToOne` and `@ManyToOne` relationships.
	Valid values:
	▪ `true`–weave entity classes.
	▪ `false`–do not weave entity classes.
	▪ `static`–weave entity classes statically.
	Use this option if you plan to execute your application outside of a Java EE 5 container in an environment that does not permit the use of `-javagent:toplink-essentials-agent.jar` on the JVM command line.
	Example: `persistence.xml` file
	`<property name="toplink.weaving" value="true"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(TopLinkProperties.WEAVING, "true");`
		`true`

`toplink.session.customizer`	Specify a TopLink session customizer class: a Java class that implements the `oracle.toplink.essentials.tools.sessionconfiguration.SessionCustomizer` interface and provides a default (zero-argument) constructor. Use this class's `customize` method, which takes an `oracle.toplink.essentials.sessions.Session`, to programmatically access advanced TopLink session API.
	For more information, see "Session Customization".
	Valid values: class name of a `SessionCustomizer` class fully qualified by its package name.
	Example: `persistence.xml` file
	`<property name="toplink.session.customizer" value="acme.sessions.MySessionCustomizer"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(TopLinkProperties.SESSION_CUSTOMIZER, "acme.sessions.MySessionCustomizer");`	

Table 26–5 (Cont.) TopLink JPA Extensions for Customization and Validation

Property	Usage	Default

`toplink.descriptor.customizer.<ENTITY>`	Specify a TopLink descriptor customizer class: a Java class that implements the `oracle.toplink.essentials.tools.sessionconfiguration.DescriptorCustomizer` interface and provides a default (zero-argument) constructor. Use this class's `customize` method, which takes an `oracle.toplink.essentials.descriptors.ClassDescriptor`, to programmatically access advanced TopLink descriptor and mapping API for the descriptor associated with the JPA entity named `<ENTITY>`.
	For more information on entity names, see `@Entity`.
	For more information on TopLink descriptors, see:
	■ "Understanding Descriptors"
	■ "Descriptor Customization"
	Valid values: class name of a `DescriptorCustomizer` class fully qualified by its package name.
	Example: `persistence.xml` file
	`<property name="toplink.descriptor.customizer.Order" value="acme.sessions.MyDescriptorCustomizer"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(TopLinkProperties.DESCRIPTOR_CUSTOMIZER+".Order", "acme.sessions.MyDescriptorCustomizer");`	

TopLink JPA Extensions for Schema Generation

Table 26–4 lists the TopLink JPA extensions that you can define in a `persistence.xml` file to configure schema generation.

Table 26–6 TopLink JPA Extensions for Schema Generation

Property	Usage	Default

`toplink.ddl-generation`	Specify what data definition language (DDL) generation action you want for your JPA entities. To specify the DDL generation target, see `toplink.ddl-generation.output-mode`.
	Valid values: `oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider`
	■ `none`–do not generate DDL; no schema is generated.
	■ `create-tables`–create DDL for non-existent tables; leave existing tables unchanged (see also `toplink.create-ddl-jdbc-file-name`).
	■ `drop-and-create-tables`–create DDL for all tables; drop all existing tables (see also `toplink.create-ddl-jdbc-file-name` and `toplink.drop-ddl-jdbc-file-name`).
	Example: `persistence.xml` file
	`<property name="toplink.ddl-generation" value="create-tables"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider; propertiesMap.put(EntityManagerFactoryProvider.DDL_GENERATION, EntityManagerFactoryProvider.CREATE_ONLY);`
		`none`

`toplink.application-location`	Specify where TopLink should write generated DDL files (see `toplink.create-ddl-jdbc-file-name` and `toplink.drop-ddl-jdbc-file-name`). Files are written if `toplink.ddl-generation` is set to anything other than `none`.
	Valid values: a file specification to a directory in which you have write access. The file specification may be relative to your current working directory or absolute. If it does not end in a file separator, TopLink will append one valid for your operating system.
	Example: `persistence.xml` file
	`<property name="toplink.application-location" value="C:\ddl\"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider; propertiesMap.put(EntityManagerFactoryProvider.APP_LOCATION, "C:\ddl\");`
		`"."+File.separator`

Table 26–6 (Cont.) TopLink JPA Extensions for Schema Generation

Property	Usage	Default

`toplink.create-ddl-jdbc-file-name`	Specify the file name of the SQL file that TopLink generates containing SQL statements to create tables for JPA entities. This file is written to the location specified by `toplink.application-location` when `toplink.ddl-generation` is set to `create-tables` or `drop-and-create-tables`.
	Valid values: a file name valid for your operating system. Optionally, you may prefix the file name with a file path as long as the concatenation of `toplink.application-location` + `toplink.create-ddl-jdbc-file-name` is a valid file specification for your operating system.
	Example: `persistence.xml` file
	`<property name="toplink.create-ddl-jdbc-file-name" value="create.sql"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider; propertiesMap.put(EntityManagerFactoryProvider.CREATE_JDBC_DDL_FILE, "create.sql");`	

`toplink.drop-ddl-jdbc-file-name`	Specify the file name of the SQL file that TopLink generates containing the SQL statements to drop tables for JPA entities. This file is written to the location specified by `toplink.application-location` when `toplink.ddl-generation` is set to `drop-and-create-tables`
	Valid values: a file name valid for your operating system. Optionally, you may prefix the file name with a file path as long as the concatenation of `toplink.application-location` + `toplink.drop-ddl-jdbc-file-name` is a valid file specification for your operating system.
	Example: `persistence.xml` file
	`<property name="toplink.drop-ddl-jdbc-file-name" value="drop.sql"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider; propertiesMap.put(EntityManagerFactoryProvider.DROP_JDBC_DDL_FILE, "drop.sql");`	

Table 26–6 (Cont.) TopLink JPA Extensions for Schema Generation

Property	Usage	Default

`toplink.ddl-generation.output-mode`	Use this property to specify the DDL generation target.
	Valid values: `oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider`
	▪ `both` - generate SQL files and execute them on the database.
	If `toplink.ddl-generation` is set to `create-tables`, then `toplink.create-ddl-jdbc-file-name` is written to `toplink.application-location` and executed on the database.
	If `toplink.ddl-generation` is set to `drop-and-create-tables`, then both `toplink.create-ddl-jdbc-file-name` and `toplink.drop-ddl-jdbc-file-name` are written to `toplink.application-location` and both SQL files are executed on the database.
	▪ `database` - execute SQL on the database only (do not generate SQL files).
	If `toplink.ddl-generation` is set to `create-tables`, then `toplink.create-ddl-jdbc-file-name` is executed on the database. It is not written to `toplink.application-location`.
	If `toplink.ddl-generation` is set to `drop-and-create-tables`, then both `toplink.create-ddl-jdbc-file-name` and `toplink.drop-ddl-jdbc-file-name` are executed on the database. Neither is written to `toplink.application-location`.
	▪ `sql-script` - generate SQL files only (do not execute them on the database).
	If `toplink.ddl-generation` is set to `create-tables`, then `toplink.create-ddl-jdbc-file-name` is written to `toplink.application-location`. It is not executed on the database.
	If `toplink.ddl-generation` is set to `drop-and-create-tables`, then both `toplink.create-ddl-jdbc-file-name` and `toplink.drop-ddl-jdbc-file-name` are written to `toplink.application-location`. Neither is executed on the database.
	Example: `persistence.xml` file
	`<property name="toplink.ddl-generation.output-mode" value="database"/>`
	Example: property `Map`
	`import oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider; propertiesMap.put(EntityManagerFactoryProvider.DDL_GENERATION_MODE, EntityManagerFactoryProvider.DDL_DATABASE_GENERATION);`
		Java EE mode (`createContainerEntityManagerFactory` called): `both`
		Java SE mode (`createEntityManagerFactory` called): `sql-script`

27

Packaging an EJB Application

This section describes the following:

- Packaging a JPA Entity Application
- Packaging an Application With Both EJB 3.0 and EJB 2.1 Enterprise Beans
- Sharing Classes Between EJB Applications

For more information, see the following:

- *Oracle Application Server Enterprise Deployment Guide*
- "Understanding Packaging" on page 2-3

Packaging a JPA Entity Application

When you package an application that uses EJB 3.0 entities, consider the following:

- Packaging a Persistence Unit
- Packaging Mapping Metadata

Packaging a Persistence Unit

Recall that an EJB 3.0 JPA persistence unit is composed of a `persistence.xml` file, one or more optional `orm.xml` files, and the managed entity classes that belong to the persistence unit.

You can package a persistence unit in its own persistence archive and include that archive in whatever Java EE modules require access to it (see "Creating a Persistence Archive" on page 27-2). Alternatively, you can package persistence unit files directly in various Java EE modules (see "Packaging Persistence Unit Files Directly in Java EE Modules" on page 27-2).

The JAR file or directory, whose `META-INF` directory contains the `persistence.xml` file, is called the root of the persistence unit. An EJB 3.0 application that uses entities must define at least one persistence unit root.

The scope of a persistence unit is determined by where you define its persistence unit root.

For more information, see the following:

- "What is the persistence.xml File?" on page 2-8
- "Configuring the persistence.xml File" on page 26-3

Creating a Persistence Archive

A persistence archive is simply a JAR file that contains a `persistence.xml` file, one or more optional `orm.xml` files, and the managed entity classes that belong to the persistence unit, as Persistence ArchiveExample 27–1 shows.

Example 27–1 Persistence Archive

```
employee-persistence.jar
    META-INF/persistence.xml
    META-INF/orm.xml
    com/acme/model/Employee.class
    com/acme/model/Address.class
    ...
```

You package a persistence archive in any of the following:

- WAR: `WEB-INF/lib` directory. The persistence unit is accessible only to the classes within this WAR.
- EAR: the root or application library directory. The persistence unit is accessible to all application components.

Using a persistence archive, you can easily share a persistence unit with multiple Java EE modules.

Packaging Persistence Unit Files Directly in Java EE Modules

You can package persistence unit files in any of the following Java EE modules:

- EJB-JAR file
- WAR file
  - `WEB-INF/classes` directory
  - `WEB-INF/lib` (in this case, `persistence.xml` file must be in a JAR)
- EAR
  - `persistence.xml` file in a JAR in root of EAR
  - `persistence.xml` file in a JAR in the EAR library directory
- Application client JAR

Table 27–1 How OC4J Handles persistence.xml in META-INF and WEB-INF

WEB-INF/classes/META-INF	WEB-INF	OC4J uses persistence.xml in
`persistence.xml`	`persistence.xml`	`META-INF` (ignores file in `WEB-INF`)
	`persistence.xml`	`META-INF`
`persistence.xml`		`WEB-INF`

To decouple persistence unit files from Java EE modules and make it easier to share persistence units with multiple Java EE modules, consider packaging your persistence unit in a persistence archive (see "Creating a Persistence Archive" on page 27-2).

Packaging Mapping Metadata

Recall that you can specify EJB 3.0 JPA mapping metadata using annotations, one or more optional `orm.xml` files, or both. You can package an `orm.xml` file in any of the following:

- `META-INF` directory of the persistence unit root (the JAR file or directory, whose `META-INF` directory contains the `persistence.xml` file);
- `META-INF` directory of any JAR file referenced by the `persistence.xml` file;
- `persistence.xml` file `<persistence-unit>` element `<mapping-file>` subelement;
- persistence archive;

For more information, see the following:

- "What is the orm.xml File?" on page 2-9
- "Packaging a Persistence Unit" on page 27-1
- "Creating a Persistence Archive" on page 27-2

Packaging an Application With Both EJB 3.0 and EJB 2.1 Enterprise Beans

You can combine both EJB 3.0 and EJB 2.1 beans in your application. For example, you could have an application that contains three annotated EJB 3.0 entities without `ejb-jar.xml` file, two EJB 2.1 entity beans with `ejb-jar.xml` file, and three EJB 3.0 session beans with `ebj-jar.xml` file, annotations, or both (in which case, the `ejb-jar.xml` overrides the annotations).

Sharing Classes Between EJB Applications

If you want to share classes between enterprise beans, you can do one of the following:

- If two enterprise beans use the same classes, include all classes and the enterprise beans in the same JAR file. After deployment, both enterprise beans can use the common classes.
- Place the shared classes in its own JAR file in the application. Reference the shared JAR file in the `class-path` of the EJB JAR `manifest.mf` file, as follows:

  ```
  Class-Path:shared_classes.jar
  ```

  The location of the `shared_classes.jar` is relative to where the JAR that references is located in the EAR file. In this example, the `shared_classes.jar` file is at the same level as the EJB JAR.
- If *all* applications reference these classes, archive the shared classes in a JAR file and place this JAR file in the shared library directory of the default application. The `home/lib` is a default shared library. However, you can set shared library directories using Enterprise Manager in the General Properties page of the "default" application.
- If you want only certain applications to reference these classes, archive the shared classes in its own application, deploy the EAR for the application, and have the applications that reference the shared classes declare the shared classes application as its parent. The default parent in OracleAS is the "default" application.

  The children see the namespace of its parent application. This is used in order to share services such as enterprise beans among multiple applications. See the *Oracle Containers for J2EE Developer's Guide* for directions on how to specify a parent application.

If you want to share classes between EJB and Web applications, you should place the referenced classes in a shared JAR.

When sharing classes between EJB applications, be aware of the following issues:

- Handling Out of Memory Exceptions at Run Time
- Handling Class Cast Exceptions at Run Time

Handling Out of Memory Exceptions at Run Time

If you see that the OC4J memory is growing consistently while executing, then you may have invalid symbolic links in your `application.xml` file. OC4J loads all resources using the links in the `application.xml` file. If these links are invalid, then the C heap continues to grow causing OC4J to run out of memory. Ensure that all symbolic links are valid and restart OC4J.

In addition, keep the number of JAR files to a minimum in the directories where the symbolic links point. Eliminate all unused JARs from these directories. OC4J searches all JARs for classes and resources; thus, taking time and memory consumption by the file cache, as well as being mapped into the address space.

Handling Class Cast Exceptions at Run Time

If you receive a `ClassCastException` at run time, then you probably have the following situation:

- You copied EJB interfaces into the WAR where the servlet resides for ease in development and forgot to delete them before creating the WAR file **AND**
- You turned on the `search_local_classes_first` attribute of the `<web-app-class-loader>` element in the `orion-web.xml` file.

To solve this problem, either eliminate the copied classes out of the WAR file, or turn off the `search_local_classes_first` attribute. This attribute tells the class loader to load in the classes in the WAR file before loading in any other classes, including the classes within the EJB JAR file. For more information on this attribute, see the "Loading WAR File Classes Before System Classes in OC4J" section in the "Servlet Development" chapter of the *Oracle Containers for J2EE Servlet Developer's Guide*.

When you have an EJB or Web application that references other shared EJB classes, you should place the referenced classes in a shared JAR. In certain situations, if you copy the shared EJB classes into WAR file or another application that references them, you may receive a `ClassCastException` because of a class loader issue. To be completely safe, never copy referenced EJB classes into the WAR file of its application or into another application.

28

Deploying an EJB Application to OC4J

This section describes the following:

- Deploying a Large EJB Application
- Deploying Incrementally
- Expanded Deployment
- Troubleshooting Application Deployment

For more information, see the following:

> **Note:** If you are using Application Server Control, EJB 3.0 entities deployed with session beans are not visible in the Application Server Control view of the EJB JAR module. For more information, see "Using Oracle Enterprise Manager 10g Application Server Control" on page 31-1.

- *Oracle Application Server Enterprise Deployment Guide*
- "Understanding Deployment" on page 2-3

Deploying a Large EJB Application

This section describes the following:

- Tuning the VM to Avoid Out Of Memory Errors During Deployment
- Configuring the Temp Directory to Avoid Out Of Memory Errors During Deployment
- Disabling Batch Compilation to Avoid Out Of Memory Errors During Deployment

For more information, see "Deploying Large Applications" in the *Oracle Containers for J2EE Deployment Guide*.

Tuning the VM to Avoid Out Of Memory Errors During Deployment

If a very large application (EAR) is deployed to OC4J, an `OutOfMemory` exception may be thrown at deployment time.

Your VM heap and permanent space configuration can cause such an exception. By default, heap and permanent space is set to 64 MB.

If there is a heap space problem, the heap space should be specified as: `java -Xmx750m -Xms512m`.

If there is a permanent space problem, the permanent space should specified as: `java -Xmx750m -Xms512m -XX:PermSize=128m -XX:MaxPermSize=256m`.

Configuring the Temp Directory to Avoid Out Of Memory Errors During Deployment

If the deployment process is interrupted for any reason, you may need to clean up the `temp` directory, which by default is `/var/tmp` on your system. The deployment wizard uses 20 MB in swap space of the temp directory for storing information during the deployment process. At completion, the deployment wizard cleans up the `temp` directory of its additional files. However, if the wizard is interrupted, it may not have the time or opportunity to clean up the `temp` directory. Thus, you must clean up any additional deployment files from this directory yourself. If you do not, this directory may fill up, which will disable any further deployment. If you receive an `OutOfMemory` exception, check for space available in the `temp` directory.

To change the `temp` directory, set the command-line option for the OC4J process to `java.io.tmpdir=<new_tmp_dir>`. You can set this command-line option in the Server Properties page. Drill down to the OC4J Home Page. Scroll down to the Administration Section. Select Server Properties. On this page, Scroll down to the Command Line Options section and add the `java.io.tmpdir` variable definition to the OC4J Options line. All new OC4J processes will start with this property.

Disabling Batch Compilation to Avoid Out Of Memory Errors During Deployment

If your application (EAR) contains multiple JAR files, you can try disabling batch deployment to fix `OutOfMemory` exceptions. However, if your EAR file only has one JAR file, this approach will not fix such exceptions: in this case, you must tune the VM (see "Tuning the VM to Avoid Out Of Memory Errors During Deployment" on page 28-1).

If OC4J throws an `OutOfMemory` exception at deploy time, and you have already tried tuning the VM (see "Tuning the VM to Avoid Out Of Memory Errors During Deployment" on page 28-1) and temp directory (see "Configuring the Temp Directory to Avoid Out Of Memory Errors During Deployment" on page 28-2), you may also attempt to compile in nonbatch mode. Although nonbatch mode requires less memory, this mode will result in a longer deployment time.

To enable or disable batch compilation, use the `<application>` or `<orion-application>` element attribute `batch-compile`.

The default value of `batch-compile` is `true`.

To disable batch compile, set this attribute to `false`.

Example 28–1 shows how to configure this attribute in the `orion-applicatin.xml` deployment descriptor.

Example 28–1 Disabling Batch Compilation in the orion-application.xml File

```
<orion-application batch-compile ="false">
...
</orion-application>
```

If out of memory errors persist, try disabling batch compile.

Deploying Incrementally

OC4J supports incremental or partial redeployment of EJB modules that are part of a deployed application. This feature makes it possible to redeploy only those beans

within an EJB JAR that have changed to be deployed, without requiring the entire module to be redeployed. Previously deployed beans that have not been changed will continue to be used.

This functionality represents a significant enhancement over previous releases of OC4J, which treated an EJB module as a single unit, requiring that the module first be undeployed, then redeployed with any updates.

A restart of OC4J is required only if changes are made to the EJB configuration data during the redeployment process. If no changes are made, a hot deployment can be performed without restarting OC4J.

The incremental redeployment operation will automatically stop the application containing the enterprise bean(s) to be updated, then automatically restart the application when finished.

> **Note:** During redeployment, all idle client connections to the enterprise bean being updated will be lost. All existing requests will be allowed to complete, but no new requests will be allowed until the application is restarted. It is strongly recommended that you stop the application before redeploying the enterprise bean.

For CMP or BMP entity beans, OC4J uses code generation to generate the server implementation of the EJB interfaces (wrappers). In this case, incrementally redeploying only changed beans is most likely to be more efficient than redeploying the entire application.

For session beans, message-driven beans, and EJB 3.0 JPA entities, OC4J uses byte code generation to generate wrappers. Because this approach reduces deployment time so much, it may be just as efficient to redeploy the entire application as to redeploy only changed beans. In this case, incremental redeployment is optional.

The general procedure for using incremental deployment is as follows:

1. Deploy an application with a large number of enterprise beans.
2. Change a bean-related class file in an EJB module and rebuild the EJB JAR file (for example, `myBeans-ejb.jar`).
3. Submit the updated EJB JAR to OC4J using any of the following:
   - JDeveloper
   - EnterpriseManager
   - `<OC4J_HOME>\j2ee\home\admin.jar` or `admin_client.jar` using the `updateEBJModule` command

   Example 28–2 shows how to use the `admin.jar`:

Example 28–2 Incremental Deployment Using the admin.jar

```
java -jar admin.jar ormi://localhost:23791 admin welcome -application -updateEJBModule -jar
myBeans-ejb.jar
```

4. Repeat steps 2 and 3.

For more information see, "Incremental Redeployment of Updated EJB Modules" in the *Oracle Containers for J2EE Deployment Guide*.

Expanded Deployment

Typically, you package your application into an EAR file before deploying it to OC4J. However, you can deploy an application while still in its expanded directory structure. Because you can skip the packaging step, this is convenient during development and testing. For example, using JDeveloper, you work on an application in its expanded directory structure. Using expanded deployment, you can deploy the expanded directory structure as often as you want without having to re-archive before each deployment.

To configure OC4J for expanded deployment, edit the *<OC4J_HOME>*`\j2ee\home\config\server.xml` file and modify the `application` element for your application to specify a `path` to the root of the expanded directory for your application. Example 28–3 shows an `application` element for application `myapp` with its `path` attribute set to the root of its expanded directory.

Example 28–3 server.xml for Expanded Deployment

```
<application-server  ...>
    ...
    <!-- Regular EAR deployment -->
    <application name="app" path="../../home/applications/app.ear" start="true" />

    <!-- Expanded deployment -->
    <application name="myapp" path="C:/projects/myapp" start="true" />
    ...
</application-server>
```

Troubleshooting Application Deployment

When you deploy an EJB 3.0 application with one or more annotations, OC4J will automatically write its in-memory `ejb-jar.xml` file to the same location as the `orion-ejb-jar.xml` file in the deployment directory: *<ORACLE_HOME>*`/j2ee/home/application-deployments/my_application/META-INF`.

This `ejb-jar.xml` file represents configuration obtained from both annotations and a deployed `ejb-jar.xml` file (if present).

When you deploy an EJB 2.1 application, to preserve generated wrapper code, you musts set system property `KeepWrapperCode` (see "Debugging Generated Wrapper Code" on page 31-9).

For more information, see "Troubleshooting an EJB Application" on page 31-8.

Part X

Using an EJB in Your Application

This part provides procedural information on using EJB 3.0 or EJB 2.1 enterprise JavaBeans in a J2EE application. For conceptual information, see Part I, "EJB Overview".

This part contains the following chapters:

29

Accessing an Enterprise Bean From a Client

This chapter explains how to access an EJB from a client, including the following:

- What Type of Client do you Have?
- Configuring the Client
- Accessing an EJB 3.0 Enterprise Bean
- Accessing an EJB 3.0 Enterprise Bean in Another Application
- Accessing a JPA Entity Using an EntityManager
- Sending a Message to a JMS Destination Using EJB 3.0
- Accessing an EJB 3.0 EJBContext
- Accessing an EJB 2.1 Enterprise Bean
- Accessing an EJB 2.1 Enterprise Bean in Another Application
- Sending a Message to a JMS Destination Using EJB 2.1
- Accessing an EJB 2.1 EJBContext
- Handling Parameters
- Handling Exceptions

For more information, see the following:

- "How do you use an Enterprise Bean in Your Application?" on page 2-10
- "Looking Up an EJB 3.0 Resource Manager Connection Factory" on page 19-23
- "Looking Up an EJB 3.0 Environment Variable" on page 19-23
- "Looking Up an EJB 2.1 Resource Manager Connection Factory" on page 19-25
- "Looking Up an EJB 2.1 Enviornment Variable" on page 19-25

Note: You can download EJB code examples from: `http://www.oracle.com/technology/tech/java/oc4j/demos`.

What Type of Client do you Have?

You can access an enterprise bean from a variety of clients, including the following:

- EJB Client
- Standalone Java Client

- Servlet or JSP Client

How you access an enterprise bean, resource, or environment variable is different depending on the type of client and how the application is assembled and deployed.

For more information, see "Configuring the Client" on page 29-2.

EJB Client

When one enterprise bean (call it the source enterprise bean) accesses another enterprise bean (call it the target enterprise bean), the source enterprise bean is the client of the target enterprise bean.

If you are using EJB 3.0, with annotations and dependency injection, OC4J initializes the instance variable that corresponds to the target reference.

If you are using EJB 2.1, you must use JNDI lookup in this scenario.

Standalone Java Client

A standalone Java client is a client that executes outside of OC4J, but accesses EJB resources deployed to OC4J.

Typically, a standalone Java client accesses EJB resources by making use of Java RMI calls. You must code a standalone Java client so that it honors the security and authentication requirements that OC4J enforces.

By default, OC4J is configured to assign RMI ports dynamically within a set range. In this release, you can look up an OC4J-deployed enterprise bean from a standalone Java client without specifying an exact RMI port. You do not need to configure OC4J to use exact port numbers.

If you are using EJB 3.0, note that annotations and dependency injection are not supported for a standalone Java client.

If you are using EJB 2.1, you must configure your initial context to accommodate this scenario (see "Accessing an EJB 2.1 Enterprise Bean Using RMI From a Standalone Java Client" on page 29-22).

Servlet or JSP Client

A servlet or JSP can access an enterprise bean.

In this release, OC4J supports annotations and resource injection in the Web tier (see "Annotations in the Web Tier" on page 1-9).

You can use dependency injection from a servlet or JSP client for EJB 3.0 applications.

You can use JNDI lookup from a servlet or JSP client for both EJB 3.0 and EJB 2.1 applications.

Configuring the Client

Before you can access an enterprise bean from a client, you must consider the following:

- Configuring the Client Classpath for OC4J
- Selecting an Initial Context Factory Class
- Specifying Security Credentials

- Selecting an EJB Reference

Configuring the Client Classpath for OC4J

Table 29–1 lists the OC4J-specific JAR files that you must install on the client depending on what your client looks up. The Source column indicates from where you get a copy of the required JAR from <*OC4J_HOME*>.

Only the `oc4jclient.jar` should be on the client classpath. All other JAR files required by the client are referenced in the `oc4jclient.jar` manifest classpath.

Table 29–1 OC4J Client Classpath Requirements

OC4J JAR	Source (Relative to <OC4J_HOME>)	EJB Lookup	JMS Connector Lookup	OEMS JMS Lookup	OEMS JMS Database Lookup
adminclient.jar	/j2ee/<*instance*>/lib		✔		✔
bcel.jar	/j2ee/<*instance*>/lib				✔
connector.jar	/j2ee/<*instance*>/lib		✔		
dms.jar	/j2ee/<*instance*>/lib				✔
ejb.jar	/j2ee/<*instance*>/lib	✔			✔
javax77.jar	/j2ee/<*instance*>/lib		✔	✔	✔
jazncore.jar	/j2ee/<*instance*>		✔		
jdbc.jar	/j2ee/<*instance*>/../../lib				
jms.jar	/j2ee/<*instance*>/lib		✔	✔	✔
jmxri.jar	/j2ee/<*instance*>/lib		✔		
jndi.jar	/j2ee/<*instance*>/lib		✔	✔	✔
jta.jar	/j2ee/<*instance*>/lib		✔	✔	✔
oc4j.jar	/j2ee/<*instance*>		✔		
oc4jclient.jar	/j2ee/<*instance*>	✔	✔	✔	✔
oc4j-internal.jar	/j2ee/<*instance*>/lib		✔		
ojdbc14dms.jar	/j2ee/<*instance*>/../../oracle/jdbc/lib				✔
optic.jar[1]	/opmn/lib			✔	✔

[1] Required only if you plan to use the `opmn:ormi` prefix in JNDI look up with `Context.PROVIDER_URL` (see "Configuring an Oracle Initial Context Factory" on page 19-20).

If you download any of these JAR files into a browser, you must grant certain permissions (see "Granting Permissions in Browser" on page 22-1).

Selecting an Initial Context Factory Class

You use an initial context factory to obtain an initial context–a reference to the JNDI namespace. Using the initial context, you can use the JNDI API to look up an enterprise bean, resource manager connection factory, environment variable, or other JNDI-accessible object. The type of initial context factory you use depends on your

client type and how you are using OC4J: standalone, or as part of Oracle Application Server (see "Configuring the Initial Context Factory" on page 19-19).

Specifying Security Credentials

If the client and target enterprise bean are not collocated, not deployed in the same application, and the target EJB application is not the client's parent, then your client must specify its credentials before accessing the target enterprise bean (see "Specifying Credentials in EJB Clients" on page 22-10).

Selecting an EJB Reference

In EJB 3.0, to access an EJB 3.0 enterprise bean or resource in an EJB client, you can use annotations, resource injection, and default JNDI names (based on class and interface names) instead of doing a JNDI lookup with a predefined environment references.

In EJB 2.1 or in EJB 3.0 (for standalone Java clients), to access an enterprise bean or resource, you must do a JNDI lookup on a predefined environment references (see "Configuring Environment References" on page 19-1). To access an EJB 2.1 enterprise bean or resource, choose the appropriate predefined environment reference (actual or logical; local or remote) and look it up using a JNDI initial context (see "Selecting an Initial Context Factory Class" on page 29-3).

If you access an enterprise bean by reference from within your client implementation, perform a JNDI lookup using the `<ejb-ref-name>` defined in the EJB deployment descriptor. For more information on defining an EJB reference to a target enterprise bean, see "EJB Environment References" on page 19-2.

Table 29–2 shows when to prefix the reference with `java:comp/env/ejb/`, which is where the container places the EJB references defined in the deployment descriptor.

Table 29–2 When to Use the java:comp/env/ejb/ Prefix

Client	Initial Context Factory	Use Prefix?
EJB Client	Default `RMIInitialContext`	Optional Not Used
Standalone Java Client	Default `ApplicationClientInitialContext`	Optional Mandatory
Servlet or JSP Client	Default `RMIInitialContext`	Optional Not Used

Example 29–1 shows how to look up an enterprise bean with logical name `ejb/HelloWorld` using the `java:comp/env/ejb/` prefix, and Example 29–2 shows how to look up this enterprise bean without the prefix.

Example 29–1 Looking Up an Enterprise Bean With the Prefix

```
InitialContext ic = new InitialContext();
HelloHome hh = (HelloHome)ic.lookup("java:comp/env/ejb/HelloWorld");
```

Example 29–2 Looking Up an Enterprise Bean Without the Prefix

```
InitialContext ic = new InitialContext();
HelloHome hh = (HelloHome)ic.lookup("ejb/HelloWorld");
```

Accessing an EJB 3.0 Enterprise Bean

You can directly look up a bean instance from JNDI (or use resource injection in an EJB 3.0 EJB client) and retrieve a bean instance without the home interface. If the `<home>` or `<local-home>` element is removed from an EJB reference, a bean instance is returned from JNDI instead of the home.

The bean instance is created by executing the no-argument `create` method on the home interface. Stateful session beans and entity beans can also use this shortcut, but they must have a no-argument `create` method, or otherwise an exception will be thrown at lookup time.

In both cases, the syntax used in obtaining the reference to the EJB business interface is independent of whether the business interface is local or remote. In the case of remote access, the actual location of a referenced enterprise bean and EJB container are, in general, transparent to the client using the remote business interface of the bean.

Using EJB 3.0, you can look up an enterprise bean using resource injection (see "Using Annotations" on page 29-5) or the `InitialContext` (see "Using Initial Context" on page 29-5).

Alternatively, you can define an environment reference to an EJB 3.0 bean using OC4J-proprietary annotations or deployment XML (see "EJB Environment References" on page 19-2).

Using Annotations

Example 29–3 shows how to use annotations and dependency injection to access an EJB 3.0 enterprise bean from an EJB client.

Example 29–3 Injecting an EJB 3.0 Enterprise Bean in an EJB 3.0 EJB Client

```
@EJB AdminService bean;

    public void privilegedTask() {
        bean.adminTask();
    }
```

Using Initial Context

This section describes the following:

- Looking Up the Remote Interface of an EJB 3.0 Enterprise Bean Using ejb-ref
- Looking Up the Remote Interface of an EJB 3.0 Enterprise Bean Using location
- Looking up the Local Interface of an EJB 3.0 Enterprise Bean Using local-ref
- Looking up the Local Interface of an EJB 3.0 Enterprise Bean Using local-location

For more information, see "Configuring the Initial Context Factory" on page 19-19.

Looking Up the Remote Interface of an EJB 3.0 Enterprise Bean Using ejb-ref

To look up the remote interface of an enterprise bean using an `ejb-ref`, do the following:

1. Define an `ejb-ref` element for the enterprise bean in the `ejb-jar.xml` file.

Example 29–4 ejb-jar.xml For an ejb-ref Element

```
<ejb-ref>
```

```
    <ejb-ref-name>ejb/Test</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local>Test</local>
</ejb-ref>
```

For more information, see "Configuring an Environment Reference to a Remote EJB: Clustered or Combined Web Tier and EJB Tier" on page 19-4).

2. Determine whether or not a prefix is required (see "Selecting an EJB Reference" on page 29-4).
3. Look up the enterprise bean using the `ejb-ref-name` element and the appropriate prefix (if required).

Example 29–5 Looking Up Using ejb-ref in an EJB 3.0 EJB Client Using Initial Context

```
InitialContext ic = new InitialContext();
Cart cart = (Cart)ic.lookup("java:comp/env/ejb/Test");
```

For more information, see "Configuring the Initial Context Factory" on page 19-19.

Looking Up the Remote Interface of an EJB 3.0 Enterprise Bean Using location

To look up the remote interface of an EJB using its `location`, do the following:

1. Define the `location` attribute for a `entity-deployment` element in the `orion-ejb-jar.xml` file.

Example 29–6 orion-ejb-jar.xml for location Attribute

```
<entity-deployment
    name="Test"
    location="app/Test"
    ...
>
...
</entity-deployment>
```

The default value for `location` attribute is the value of `entity-deployment` attribute `name`.

2. Determine whether or not a prefix is required (see "Selecting an EJB Reference" on page 29-4).
3. Look up the enterprise bean using the `location`.

Example 29–7 Looking Up Using location in an EJB 3.0 EJB Client Using Initial Context

```
InitialContext ic = new InitialContext();
Cart cart = (Cart)ic.lookup("java:comp/env/app/Test");
```

For more information, see "Configuring the Initial Context Factory" on page 19-19.

Looking up the Local Interface of an EJB 3.0 Enterprise Bean Using local-ref

To look up the remote interface of an EJB using an `ejb-local-ref`, do the following:

1. Define an `ejb-local-ref` element for the enterprise bean in the `ejb-jar.xml` file.

Example 29–8 ejb-jar.xml For an ejb-local-ref Element

```
<ejb-local-ref>
    <ejb-ref-name>ejb/Test</ejb-ref-name>
```

```
    <ejb-ref-type>Session</ejb-ref-type>
    <local>Test</local>
</ejb-local-ref>
```

For more information, see "Configuring an Environment Reference to a Local EJB" on page 19-9).

2. Determine whether or not a prefix is required (see "Selecting an EJB Reference" on page 29-4).
3. Look up the enterprise bean using the `ejb-ref-name` and the appropriate prefix (if required).

Example 29–9 Looking Up Using local-ref in an EJB 3.0 EJB Client Using Initial Context

```
InitialContext ic = new InitialContext();
Cart cart = (Cart)ctx.lookup("java:comp/env/ejb/Test");
```

For more information, see "Configuring the Initial Context Factory" on page 19-19.

Looking up the Local Interface of an EJB 3.0 Enterprise Bean Using local-location

To look up the local interface of an EJB using its `local-location`, do the following:

1. Define the `local-location` attribute of the `entity-deployment` element in the `orion-ejb-jar.xml` file.

Example 29–10 orion-ejb-jar.xml for local-location Attribute

```
<entity-deployment
    name="Test"
    local-location="app/Test"
    ...
>
...
</entity-deployment>
```

The default value for `local-location` is the value of `entity-deployment` attribute `name`.

2. Determine whether or not a prefix is required (see "Selecting an EJB Reference" on page 29-4).
3. Look up the enterprise bean using the `local-location`.

Example 29–11 Looking Up Using local-location in an EJB 3.0 EJB Client Using Initial Context

```
InitialContext ic = new InitialContext();
Cart cart = (Cart)ctx.lookup("java:comp/env/app/Test");
```

For more information, see "Configuring the Initial Context Factory" on page 19-19.

Accessing an EJB 3.0 Enterprise Bean in Another Application

Normally, you cannot have enterprise beans communicating across EAR files, that is, across applications that are deployed in separate EAR files. The only way for an enterprise bean to access an enterprise bean that was deployed in a separate EAR file is to declare it to be the parent of the client. Only children can invoke methods in a parent.

For example, there are two enterprise beans, each deployed within their EAR file, called `Sales` and `Inventory`, where the `Sales` enterprise bean needs to invoke the `Inventory` enterprise bean to check to see if enough widgets are available. Unless the `Sales` enterprise bean defines the `Inventory` enterprise bean to be its parent, the `Sales` enterprise bean cannot invoke any methods in the `Inventory` enterprise bean, because they are both deployed in separate EAR files. So, define the `Inventory` enterprise bean to be the parent of the `Sales` enterprise bean, and the `Sales` enterprise bean can now invoke any method in its parent.

You can only define the parent during deployment with the deployment wizard. See the "Deploying/Undeploying Applications" section in the "Using the oc4jadmin.jar Command Line Utility" chapter in the *Oracle Containers for J2EE Configuration and Administration Guide* on how to define the parent application of a bean.

Accessing a JPA Entity Using an EntityManager

In an EJB 3.0 application, the `javax.persistence.EntityManager` is the run-time access point for persisting entities to and loading entities from the database.

This section describes the following:

- Acquiring an EntityManager
- Creating a New Entity Instance
- Querying for a JPA Entity Using the EntityManager
- Modifying an Entity Instance
- Detaching and Merging an Entity Bean Instance

For more information, see "How do you Query for a JPA Entity?" on page 1-39.

Note: You can download a JPA entity manager code example from: `http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30entitymanager/doc/how-to-ejb30-entitymanager.html`.

Acquiring an EntityManager

Before you can use an `EntityManager`, you must acquire an `EntityManager` instance. How you acquire an entity manager depends on your client type ("What Type of Client do you Have?" on page 29-1).

When you acquire an entity manager, you specify a persistence unit. The persistence unit defines the entity manager's configuration, including details such as which factories to use, which persistent managed classes the entity manager can manage, and what object-relational mapping metadata to use. You can only acquire an entity manager for a particular persistence unit, if your client is in the persistence unit's scope. For more information, see "What is the persistence.xml File?" on page 2-8.

You can acquire an entity manager by doing the following:

- Acquiring the OC4J Default Entity Manager
- Acquiring a Named Entity Manager
- Acquiring an Entity Manager Using JNDI
- Acquiring an Entity Manager in a Web Client
- Acquiring an Entity Manager in a Helper Class

Acquiring the OC4J Default Entity Manager

You can use the `@PersistenceContext` annotation to inject an `EntityManager` in an EJB 3.0 client (such as a stateful or stateless session bean, message-driven bean, or servlet). You can use `@PersistenceContext` without specifying a `unitName` attribute to use the OC4J default persistence unit, as Example 29–12 shows.

Example 29–12 Using @PersistenceContext With the OC4J Default Persistence Unit

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession {

    @PersistenceContext protected EntityManager entityManager;

    public void createEmployee(String fName, String lName) {
        Employee employee  = new Employee();
        employee.setFirstName(fName);
        employee.setLastName(lName);
        entityManager.persist(employee);
    }
...
}
```

For more information, see "Understanding OC4J Persistence Unit Defaults" on page 2-8.

Acquiring a Named Entity Manager

You can use the `@PersistenceContext` annotation to inject an `EntityManager` in an EJB 3.0 client (such as a stateful or stateless session bean, message-driven bean, or servlet). You can use `@PersistenceContext` attribute `unitName` to specify a persistence unit by name, as Example 29–13 shows. In this case, you must configure the persistence unit in a `persistence.xml` file.

Example 29–13 Using @PersistenceContext With a Named Persistence Unit

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession {

    @PersistenceContext(unitName="myPersistenceUnit") protected EntityManager entityManager;

    public void createEmployee(String fName, String lName) {
        Employee employee  = new Employee();
        employee.setFirstName(fName);
        employee.setLastName(lName);
        entityManager.persist(employee);
    }
...
}
```

For more information, see the following:

- "What is the persistence.xml File?" on page 2-8
- "Configuring the persistence.xml File" on page 26-3

Acquiring an Entity Manager Using JNDI

Alternatively, you can use annotations to inject a persistence context and then use JNDI to look up the entity manager, as Example 29–14 shows. In this case, you must define the persistence unit in a `persistence.xml` file.

Example 29–14 Using InitialContext to Lookup an EntityManager in a Stateless Session Bean

```
@PersistenceContext(
    name="persistence/InventoryAppMgr",
    unitName=InventoryManagement  // defined in a persistence.xml file
)
@Stateless
public class InventoryManagerBean implements InventoryManager {

    EJBContext ejbContext;
    public void updateInventory(...) {
        ...
        // obtain the initial JNDI context
        Context initCtx = new InitialContext();
        // perform JNDI lookup to obtain container-managed entity manager
        javax.persistence.EntityManager entityManager = (javax.persistence.EntityManager)
            initCtx.lookup("java:comp/env/persistence/InventoryAppMgr");
        ...
    }
}
```

For more information, see the following:

- "Configuring the Initial Context Factory" on page 19-19
- "What is the persistence.xml File?" on page 2-8
- "Configuring the persistence.xml File" on page 26-3

Acquiring an Entity Manager in a Web Client

In this release, you can also use the `@PersistenceContext` annotation to inject an `EntityManager` in a Web client such as a servlet, as Example 29–15 shows. This example injects the default `EntityManager`; you can also inject a named entity manager as Example 29–13 shows. For more information, see "Annotations in the Web Tier" on page 1-9.

Example 29–15 Using @PersistenceContext to Inject an EntityManager in a Servlet

```
@Resource
UserTransaction ut;
@PersistenceContext
EntityManager entityManager;
...
try {
    ut.begin();

    Employee employee = new Employee();
    employee.setEmpNo(empId);
    employee.setEname(name);
    employee.setSal(sal);

    entityManager.persist(employee);
    ut.commit();

    this.getServletContext().getRequestDispatcher(
        "/jsp/success.jsp").forward(request, response);
}
catch(Exception e) {
...
}
```

Acquiring an Entity Manager in a Helper Class

To acquire an entity manager in a class that does not support annotations and injection, namely a helper class, you must do the following:

1. Define a persistence unit in a `persistence.xml` file.

   For more information, see the following:

   - "What is the persistence.xml File?" on page 2-8
   - "Configuring the persistence.xml File" on page 26-3

2. Declare a reference to this persistence unit at the class level in each Java EE component that makes use of the helper class. The persistence unit will appear in the Java EE component's environment (`java:comp/env`).

   You can do this in one of the following ways:

   a. Using the `@PersistenceContext` annotation in the Java EE component that makes use of the helper class as follows:

   ```
   @PersistenceContext(name="helperPC", unitName="HelperPU")
   @Stateless
   public class EmployeeDemoSessionEJB implements EmployeeDemoSession {
       import com.acme.Helper;
       ...
       void doSomething() {
           Helper.createNewEmployee();
       }
   }
   ```

   In the `@PersistenceContext` annotation, you specify:

   – `name`: the name by which you will look up the persistence context

   – `unitName`: the name of the persistence unit you created in step 1, that defines the characteristics of the returned entity manager.

   b. Using a `persistence-context-ref` in the appropriate deployment descriptor file for the Java EE component that makes use of the helper class (see "Configuring an Environment Reference to a Persistence Context" on page 19-18).

   In the `persistence-context-ref`, you specify the following:

   – `persistence-context-ref`: the name by which you will look up the persistence context.

   – `persistence-unit-name`: the name of the persistence unit you created in step 1, that defines the characteristics of the returned entity manager.

3. In the helper class, use JNDI to look up the entity manager using the persistence unit name you defined:

   ```
   public class Helper {
       ...
       int createNewEmployee()
       {
           UserTransaction ut = null;
           ...
           try {
               Context initCtx = new InitialContext();

               ut = (UserTransaction)initCtx.lookup("java:comp/UserTransaction");
               ut.begin();
   ```

```
            Employee employee = new Employee();
            employee.setEmpNo(empId);

            // obtain the initial JNDI context
            Context initCtx = new InitialContext();
            javax.persistence.EntityManager entityManager =
                (javax.persistence.EntityManager)initCtx.lookup(
                    "java:comp/env/helperPC"
                );

            entityManager.persist(employee);

            ut.commit();
        }
        catch(Exception e) {
            ...
        }
    }
}
```

> **Note:** In the helper class, when you use the `EntityManager`, you must manually demarcate a transaction using the `UserTransaction` API, because you must use the `EntityManager` within a transaction.

For more information, see "Configuring the Initial Context Factory" on page 19-19.

Creating a New Entity Instance

To create a new entity instance, after acquiring an `EntityManager` ("Acquiring an EntityManager" on page 29-8), use `EntityManager` method `persist` passing in the entity `Object`, as Example 29–16 shows. When you call this method, it marks the new instance for insert into the database. This method returns the same instance that you passed in.

You must call this method within a transaction context.

> **Note:** Only use `EntityManager` method `persist` on a new entity. If you make changes to an existing entity, they are written to the database when the current transaction commits (see also "Using Flush" on page 29-17).

Example 29–16 Creating an Entity With the EntityManager

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession {

    @PersistenceContext protected EntityManager entityManager;
...
    public void createEmployee(String fName, String lName) {
        Employee employee  = new Employee();
        employee.setFirstName(fName);
        employee.setLastName(lName);
        entityManager.persist(employee);
    }
...
}
```

Querying for a JPA Entity Using the EntityManager

This section describes how to use the `EntityManager` to query for EJB 3.0 entities, including:

- Finding an Entity by Primary Key With the Entity Manager
- Creating a Named Query With the EntityManager
- Creating a Dynamic Java Persistence Query Language Query With the Entity Manager
- Creating a Dynamic TopLink Expression Query With the EntityManager
- Creating a Dynamic Native SQL Query With the EntityManager
- Executing a Query

For more information, see the following:

- "How do you Query for a JPA Entity?" on page 1-39
- "Implementing JPA Queries" on page 8-1
- "Configuring TopLink Query Hints in a JPA Query" on page 8-3

Finding an Entity by Primary Key With the Entity Manager

As Example 29–17 shows, f you know the primary key, you can use EntityManager method find to retrieve the corresponding entity from the database without having to create a query.

Example 29–17 Finding an Entity by Primary Key Using the EntityManager

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession {
...
    public void removeEmployee(Integer employeeId) {
        Employee employee = (Employee)entityManager.find("Employee", employeeId);
        ...
        entityManager.remove(employee);
    }
...
}
```

Creating a Named Query With the EntityManager

After you implement a named query (see "Implementing a JPA Named Query" on page 8-1), you can acquire it at run time using `EntityManager` method `createNamedQuery`, as Example 29–18 Creating a Named Query with the EntityManager shows. If the named query takes parameters, you set them using `Query` method `setParameter`.

Example 29–18 Creating a Named Query with the EntityManager

```
Query queryEmployeesByFirstName = entityManager.createNamedQuery(
    "findAllEmployeesByFirstName"
);
queryEmployeeByFirstName.setParameter("firstName", "John");
Collection employees = queryEmployessByFirstName.getResultList();
```

Optionally, you can configure your query with query hints to use JPA persistence provider vendor extensions (see "Configuring TopLink Query Hints in a JPA Query" on page 8-3).

Creating a Dynamic Java Persistence Query Language Query With the Entity Manager

Example 29–19 shows how to create an ad hoc EJB QL query at run time using `EntityManager` method `createQuery`.

Example 29–19 Creating a Dynamic Query Using the EntityManager

```
Query queryEmployees = entityManager.createQuery(
    "SELECT OBJECT(employee) FROM Employee employee"
);
```

Example 29–20 shows how to create an ad hoc query that takes a parameter named `firstname` using `EntityManager` method `createQuery`. You set the parameter using `Query` method `setParameter`.

Example 29–20 Creating a Dynamic Java Persistence Query Language Query with Parameters Using the EntityManager

```
Query queryEmployees = entityManager.createQuery(
    "SELECT OBJECT(emp) FROM Employee emp WHERE emp.firstName = :firstname"
);
queryEmployeeByFirstName.setParameter("firstName", "John");
```

Optionally, you can configure your query with query hints to use JPA persistence provider vendor extensions (see "Configuring TopLink Query Hints in a JPA Query" on page 8-3).

Creating a Dynamic TopLink Expression Query With the EntityManager

As Example 29–21 shows, using the `oracle.toplink.ejb.cmp3.EntityManager` method `createQuery(Expression expression, Class resultType)`, you can create a query based on a TopLink `Expression`.

Optionally, you can configure your query with query hints to use JPA persistence provider vendor extensions (see "Configuring TopLink Query Hints in a JPA Query" on page 8-3).

For more information, see "Understanding TopLink Expressions" in the *Oracle TopLink Developer's Guide*.

Example 29–21 Creating a Dynamic TopLink Expression Query Using the Entity Manager

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession {
...
    public Collection findManyProjectsByQuery(Vector params) {
        ExpressionBuilder builder = new ExpressionBuilder();
        Query query = ((oracle.toplink.ejb.cmp3.EntityManager)em).createQuery(
            builder.get("name").equals(builder.getParameter("projectName")),
            Project.class);
        query.setParameter("projectName", params.firstElement());
        Collection projects = query.getResultList();
        return projects;
    }
...
}
```

Creating a Dynamic Native SQL Query With the EntityManager

Using `EntityManager` methods `createNativeQuery(String sqlString)` or `createNativeQuery(String sqlString, Class resultType)`, you can create a query based on a native SQL string that you supply, as Example 29–22 shows.

Example 29–22 Creating a Dynamic Native SQL Query with the EntityManager

```
Query queryEmployees = entityManager.createNativeQuery(
    "Select * from EMP_TABLE where Salary < 50000", Employee.class
);
```

Example 29–23 shows how to create an ad hoc native SQL query that takes a parameter named `salary` using `EntityManager` method `createNativeQuer(String sqlString, Class resultClass)`. You set the parameter using `Query` method `setParameter`.

Example 29–23 Creating a Dynamic Native SQL Query with Parameters Using the EntityManager

```
Query queryEmployees = entityManager.createNativeQuery(
    "Select * from EMP_TABLE where Salary < #salary", Employee.class
);
queryEmployeeByFirstName.setParameter("salary", 50000);
```

Optionally, you can configure your query with query hints to use JPA persistence provider vendor extensions (see "Configuring TopLink Query Hints in a JPA Query" on page 8-3).

Executing a Query

As Example 29–24 shows, to execute a query that returns multiple results, use `Query` method `getResultList`. This method returns a `java.util.List`.

Example 29–24 Executing a Query that Returns Multiple Results

```
Collection employees = queryEmployees.getResultList();
```

As Example 29–25 shows, to execute a query that returns a single result, use `Query` method `getSingleResult`. This method returns a `java.lang.Object`.

Example 29–25 Executing a Query that Returns a Single Result

```
Object obj = query.getSingleResult();
```

As Example 29–26 shows, to execute a query that updates (modifies or deletes) entities, use `Query` method `executeUpdate`. This method returns the number of rows affected (updated or deleted) as an `int`.

Example 29–26 Executing an Updating Query

```
Query queryRenameCity = entityManager.createQuery(
    "UPDATE Address add SET add.city = 'Ottawa' WHERE add.city = 'Nepean'");
int rowCount = queryRenameCity.executeUpdate();
```

Modifying an Entity Instance

You can modify an entity instance in one the following ways:

- Using an Updating Query

- Using the Entity's Public API
- Refreshing From the Database
- Removing an Entity

You must perform these operations within a transaction context. When the current transaction commits, your updates will be committed to the database.

You can also send updates to the database within a transaction before commit (see "Using Flush" on page 29-17).

Using an Updating Query

Create an updating query (see "Creating a Named Query With the EntityManager" on page 29-13 or "Creating a Dynamic Java Persistence Query Language Query With the Entity Manager" on page 29-14) and execute the query using the `EntityManager` (see "Executing a Query" on page 29-15).

Using the Entity's Public API

Use the EntityManager to find or otherwise query for the entity (see "Querying for a JPA Entity Using the EntityManager" on page 29-13).

Use the entity's public API to change its persistent state.

Refreshing From the Database

As Example 29–27 shows, you can overwrite the current state of an entity instance with the currently committed state from the database using the `EntityManager` method refresh.

Example 29–27 Refreshing an Entity from the Database

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession {
...
    public void undoUpdateEmployee(Integer employeeId) {
        Employee employee = (Employee)entityManager.find("Employee", employeeId);
        em.refresh(employee);
    }
...
}
```

Removing an Entity

As Example 29–28 shows, you can use `EntityManager` method `remove` to delete an entity from the database.

Example 29–28 Removing an Entity

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession {
...
    public void removeEmployee(Integer employeeId) {
        Employee employee = (Employee)entityManager.find("Employee", employeeId);
        ...
        entityManager.remove(employee);
    }
...
}
```

Using Flush

As Example 29–29 shows, you can use `EntityManager` method `flush` to send updates to the database within a transaction before the transaction is committed. Subsequent queries within the same transaction will return the updated data. This is useful if a particular transaction spans multiple operations.

Example 29–29 Sending Updates to the Database Within a Transaction

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession {
...
    public void terminateEmployee(Integer employeeId, Date endDate) {
        Employee employee = (Employee) entityManager.find("Employee", employeeId);
        employee.getPeriod().setEndDate(endDate);
        entityManager.flush();
    }
...
}
```

Detaching and Merging an Entity Bean Instance

An `EntityManager` is said to have a persistence context. When you create (see "Creating a New Entity Instance" on page 29-12) or find (see "Querying for a JPA Entity Using the EntityManager" on page 29-13) an entity using an `EntityManager` instance, the entity is said to be part of the persistence context of that `EntityManager`.

While an entity is part of the persistence context of an `EntityManager`, it is said to be a persistent entity.

When an entity is no longer part of this persistence context, it is said to be a detached entity.

An entity is detached from the persistence context when the persistence context ends or when an entity is serialized (for example, to a separate application tier).

As Example 29–30 shows, you can use `EntityManager` method `merge` to merge the state of detached entity into the current persistence context of the `EntityManager`.

Example 29–30 Merging an Entity into the Persistence Context of an EntityManager

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession {
...
    public void updateAddress(Address addressExample) {
        entityManager.merge(addressExample);
    }
...
}
```

For more information about persistence context, see the following:

- "What is the persistence.xml File?" on page 2-8
- "Configuring the persistence.xml File" on page 26-3

Sending a Message to a JMS Destination Using EJB 3.0

A client never accesses an MDB directly: rather, the client accesses an MDB by sending a message through the JMS destination (queue or topic) associated with the MDB.

To send a message to a JMS destination using EJB 3.0, do the following:

1. Inject both the JMS destination (queue or topic) and its connection factory.

   You can inject these resources using a predefined logical name or the explicit JNDI name you defined when you configured your JMS provider. Oracle recommends that you use logical names as shown in this procedure and its examples.

   For more information, see the following:

   - "Configuring an Environment Reference to a JMS Destination Resource Manager Connection Factory (JMS 1.1)" on page 19-13
   - "Configuring an Environment Reference to a JMS Destination or Connection Resource Manager Connection Factory (JMS 1.0)" on page 19-14
2. Use the connection factory to create a connection.

   If you are receiving messages for a queue, then start the connection.
3. Create a session over the connection.
4. Use the retrieved JMS destination to create a sender for a queue or a publisher for a topic.
5. Create the message.
6. Send the message using either the queue sender or the topic publisher.
7. Close the queue session.
8. Close the connection.

Example 29–31 shows how a servlet client sends a message to a queue.

Example 29–31 Servlet Client Sends Message to a Queue

```
public final class testResourceProvider extends HttpServlet {

  private String resProvider = "myResProvider";
  private HashMap msgMap = new HashMap();

  // 1a. Rely on Servlet container to inject queue connection factory
  @Resource(name=resProvider+"QueueConnectionFactories/myQCF")
  private QueueConnectionFactory qcf;

  // 1b. Rely on Servlet container to inject queue
  @Resource(name=resProvider+"/Queues/rpTestQueue")
  private Queue queue;

  public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    doPost(req, res);
 }

  public void doPost(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
   // Retrieve the name of the JMS provider from the request,
   // which is to be used in creating the JNDI string for retrieval
    String rp = req.getParameter ("provider");
    if (rp != null) resProvider = rp;

    try {
      // 2a. Create queue connection using the connection factory
      QueueConnection qconn = qcf.createQueueConnection();
      // 2b. You are receiving messages, so start the connection
      qconn.start();

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

```
      // 4. Since this is for a queue, create a sender on top of the session
      // This is used to send out the message over the queue
     QueueSender snd = sess.createSender (q);

   drainQueue (sess, q);
   TextMessage msg = null;

   // Send messages to queue
   for (int i = 0; i < 3; i++) {
     // 5. Create message
     msg = sess.createTextMessage();
     msg.setText ("TestMessage:" + i);

     // Set property of the recipient to be the MDB
     // and set the reply destination.
     msg.setStringProperty ("RECIPIENT", "MDB");
     msg.setJMSReplyTo(q);

     // 6. Send the message using the sender
     snd.send (msg);

     // You can store the messages IDs and sent-time  in a map (msgMap),
     // so that when messages are received, you can verify if you
     // *only* received those messages that you were
     // expecting. See receiveFromMDB() method where msgMap gets used
     msgMap.put( msg.getJMSMessageID(), new Long (msg.getJMSTimestamp()));
   }

   // receive a reply from the MDB
   receiveFromMDB (sess, q);

    // 7. Close sender, session, and connection for queue
    snd.close();
    sess.close();
    qconn.close();
   }
   catch (Exception e) {
     System.err.println ("** TEST FAILED **"+ e.toString());
     e.printStackTrace();
   }
   finally {
   }
}

 // Receive any messages sent to you through the MDB
 private void receiveFromMDB (QueueSession sess, Queue q)
   throws Exception {
   // The MDB sends out a message (as a reply) to this client. The MDB sets
   // the receipient as CLIENT. Thus, You will only receive messages that have
   // RECIPIENT set to 'CLIENT'
   QueueReceiver rcv = sess.createReceiver (q, "RECIPIENT = 'CLIENT'");

   int nrcvd = 0;
   long trtimes = 0L;
   long tctimes = 0L;
   // First message needs to come from MDB.
   // May take  a little while receiving messages
   for (Message msg = rcv.receive (30000); msg != null; msg = rcv.receive (30000)) {
     nrcvd++;
     String rcp = msg.getStringProperty ("RECIPIENT");

     // Verify if message is in message Map
     // Check the msgMap to see if this is the message that you are expecting
     String corrid = msg.getJMSCorrelationID();
     if (msgMap.containsKey(corrid)) {
```

```
        msgMap.remove(corrid);
      }
      else {
        System.err.println ("** received unexpected message [" + corrid + "] **");
      }
    }
    rcv.close();
  }

  // Drain messages from queue
  private int drainQueue (QueueSession sess, Queue q)
    throws Exception {
    QueueReceiver rcv = sess.createReceiver (q);
    int nrcvd = 0;

    // First drain any old messages from queue
    for (Message msg = rcv.receive(1000); msg != null; msg = rcv.receive(1000))
      nrcvd++;

    rcv.close();

    return nrcvd;
  }
}
```

Accessing an EJB 3.0 EJBContext

For EJB 3.0 session and message-driven beans, you can access the `EJBContext` that OC4J provides (see "Using Resource Injection" on page 29-20).

For more information, see the following:

- "What is EJB Context?" on page 1-6
- "What is Session Context?" on page 1-34
- "What is Message Driven Context?" on page 1-58

Using Resource Injection

In an EJB 3.0 EJB client, you can use `@Resource` injection to access the `EJBContext`, as Example 29–32 shows.

Example 29–32 Accessing EJBContext Using @Resource

```
@Resource SessionContext ctx;
```

Accessing an EJB 2.1 Enterprise Bean

This section describes the following:

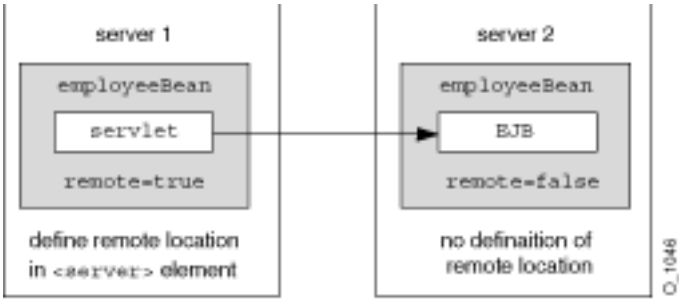
- Accessing an EJB 2.1 Enterprise Bean Remotely
- Accessing an EJB 2.1 Enterprise Bean Locally
- Accessing an EJB 2.1 Enterprise Bean Using RMI From a Standalone Java Client
- Accessing an EJB 2.1 Enterprise Bean From an EJB 3.0 Client

Accessing an EJB 2.1 Enterprise Bean Remotely

A remote multitier situation exists when you have the servlets executing in one server, which are to connect and communicate with enterprise beans in another server. Both the servlets and enterprise beans are contained in the same application. When you deploy the application to two different servers, the servlets normally look for the local enterprise bean first.

In Figure 29–1, the `HelloBean` application is deployed to both server 1 and 2. In order to ensure that the servlets only call out from server 1 to the enterprise beans in server 2, you must set the `remote` attribute appropriately in the application before deploying on both servers.

Figure 29–1 Multitier Example



The `remote` attribute in the `<ejb-module>` element in `orion-application.xml` for the EJB module denotes whether the enterprise beans for this application are deployed or not.

1. In server 1, you must set `remote=true` in the `<ejb-module>` element of the `orion-application.xml` file ,and then deploy the application. The EJB module within the application will not be deployed. Thus, the servlets will not look for the enterprise beans locally, but will go out to the remote server for the EJB requests.
2. In server 2, you must set `remote=false` in the `<ejb-module>` element of the `orion-application.xml` file and then deploy the application. The application, including the EJB module, is deployed as normal. The default for the `remote` attribute is `false`; thus, simply ensure that the `remote` attribute is not `true` and redeploy the application.
3. Configure RMI options:
   - In a standalone OC4J, specify RMI server data in the RMI configuration file, `rmi.xml`. Specify the location of this file in `server.xml`, the OC4J configuration file. By default, both these files are installed in `<`*`ORACLE_HOME`*`>/j2ee/home/config`.

     For more information, see "Configuring RMI in a Standalone OC4J Installation" in the *Oracle Containers for J2EE Services Guide*.
   - In an Oracle Application Server environment, you must edit the `opmn.xml` file to specify the port range, on which this local RMI server listens for RMI requests. Note that manual changes to configuration files in an Oracle Application Server environment must be manually updated on each OC4J instance.

     For more information, see "Configuring RMI in an Oracle Application Server Environment" in the *Oracle Containers for J2EE Services Guide*.

4. Set JNDI properties `java.naming.provider.url` and `java.naming.factory.initial`.

   For more information see the following:

   - "Configuring the Initial Context Factory" on page 19-19
   - "Setting JNDI Properties for RMI" in the *Oracle Containers for J2EE Services Guide*.

5. Look up the remote enterprise bean.

   If multiple remote servers are configured, OC4J searches all remote servers for the intended EJB application.

For more information, see "Using Remote Method Invocation in OC4J" in the *Oracle Containers for J2EE Services Guide*.

Accessing an EJB 2.1 Enterprise Bean Locally

A local multitier situation exists when both the servlets and enterprise beans are contained in the same application and deployed to the same server.

The `remote` attribute in the `<ejb-module>` element in `orion-application.xml` for the EJB module denotes whether the enterprise beans for this application are deployed or not.

1. In the server, to which you deploy your application, you must set `remote=false` in the `<ejb-module>` element of the `orion-application.xml` file, and then deploy the application. The application, including the EJB module, is deployed as normal. The default for the `remote` attribute is `false`.
2. Set JNDI properties `java.naming.provider.url` and `java.naming.factory.initial`.

   For more information see the following:

   - "Configuring the Initial Context Factory" on page 19-19
   - "Setting JNDI Properties for RMI" in the *Oracle Containers for J2EE Services Guide*.

3. Look up the local EJB.

Accessing an EJB 2.1 Enterprise Bean Using RMI From a Standalone Java Client

Example 29–33 shows the type of lookup that you can use from a standalone Java client (see "Standalone Java Client" on page 29-2) in this release to look up an OC4J-deployed enterprise bean without having to specify an RMI port. Example 29–33 shows how to look up the enterprise bean named `MyCart` in the Java EE application `ejbsamples` deployed to the OC4J instance named `oc4j_inst1` running on host `myServer`.

Example 29–33 Accessing an EJB 2.1 Enterprise Bean Using RMI from a Standalone Java Client

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,"oracle.j2ee.rmi.RMIInitialContextFactory");
env.put(Context.SECURITY_PRINCIPAL, "oc4jadmin");
env.put(Context.SECURITY_CREDENTIALS, "password");
env.put(Context.PROVIDER_URL,"opmn:ormi://myServer:oc4j_inst1/ejbsamples");

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

```
Object homeObject = context.lookup("MyCart");
CartHome home = (CartHome)PortableRemoteObject.narrow(homeObject,CartHome.class);
```

For more information, see the following:

- "Configuring an Oracle Initial Context Factory" on page 19-20
- "Configuring the Naming Provider URL for OC4J and Oracle Application Server" on page 19-20
- "Configuring the Naming Provider URL for OC4J Standalone" on page 19-21

Accessing an EJB 2.1 Enterprise Bean From an EJB 3.0 Client

To access an EJB 2.1 enterprise bean from an EJB 3.0 client:

1. Create an environement reference to the EJB 2.1 enterprise bean's home and remote interface as Example 29–34 shows.

   In this example, you configure an environment reference to the home and remote interface of the EJB 2.1 `Scheduler` bean. For more information on Job Scheduler, see the *Oracle Containers for J2EE Job Scheduler Developer's Guide*.

Example 29–34 Creating an Environment Reference to an EJB 2.1 Enterprise Bean's Home and Remote Interface

```
<ejb-ref>
    <ejb-ref-name>ejb/scheduler</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>oracle.ias.scheduler.SchedulerHome</home>
    <remote>oracle.ias.scheduler.SchedulerRemote</remote>
</ejb-ref>
```

   For more information, see Chapter 19, "Configuring JNDI Services"

2. Access the EJB 2.1 enterprise bean from the EJB 3.0 client:

   An EJB 3.0 client can access an EJB 2.1 enterprise bean in a variety of ways, including, but not limited to, the following:

   a. Inject the EJB 2.1home interface using the `@EJB` annotation as Example 29–35 shows.

      In this example, you set the `@EJB` annotation name attribute to the `<ejb-ref-name>` of the EJB 2.1 enterprise bean.

Example 29–35 Injecting an EJB 2.1 Home Interface Using @EJB

```
...
public class MyEJB30Client {

    @EJB(name="ejb/scheduler")
    SchedulerHome home;

    public void bar() {
        home.create();
        ...
    }
}
```

   b. Inject the EJB 2.1 home interface using the `<injection-target>` element in deployment XML.

Example 29–36 shows how to add an `<injection-target>` element to the deployment XML to associate the EJB 2.1 home interface with an instance variable named `home`. As Example 29–37 shows, at deployment time, OC4J will ensure that instance variable `home` in the EJB 3.0 client is initialized appropriately.

Example 29–36 Adding an <injection-target> to the Deployment XML

```
<ejb-ref>
    <ejb-ref-name>ejb/scheduler</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>oracle.ias.scheduler.SchedulerHome</home>
    <remote>oracle.ias.scheduler.SchedulerRemote</remote>
    <injection-target>
        <injection-target-name>home</injection-target-name>
    </injection-target>
</ejb-ref>
```

Example 29–37 Injecting an EJB 2.1 Home Interface Into an Instance Variable Using

```
...
public class MyEJB30Client {

    SchedulerHome home;

    public void bar() {
        home.create();
        ...
    }
}
```

c. Look up the EJB 2.1 home interface using JNDI as Example 29–38 shows.

In this example, you look up the EJB 2.1 enterprise bean's `<ejb-ref-name>` prefixed with `java:comp/env/`.

Example 29–38 Performing a JNDI Lookup of the Home Interface

```
...
public class MyEJB30Client {

    SchedulerHome home;

    public void bar() {
        InitalContext ic = new InitialContext();
        home = ic.lookup("java:comp/env/ejb/scheduler");
        home.create();
        ...
    }
}
```

Accessing an EJB 2.1 Enterprise Bean in Another Application

Normally, you cannot have enterprise beans communicating across EAR files, that is, across applications that are deployed in separate EAR files. The only way for an enterprise bean to access an enterprise bean that was deployed in a separate EAR file is to declare it to be the parent of the client. Only children can invoke methods in a parent.

For example, there are two enterprise beans, each deployed within their EAR file, called `Sales` and `Inventory`, where the `Sales` enterprise bean needs to invoke the `Inventory` enterprise bean to check to see if enough widgets are available. Unless the `Sales` enterprise bean defines the `Inventory` enterprise bean to be its parent, the `Sales` enterprise bean cannot invoke any methods in the `Inventory` enterprise bean, because they are both deployed in separate EAR files. So, define the `Inventory` enterprise bean to be the parent of the `Sales` enterprise bean and the `Sales` enterprise bean can now invoke any method in its parent.

You can only define the parent during deployment with the deployment wizard. See the "Deploying/Undeploying Applications" section in the "Using the oc4jadmin.jar Command Line Utility" chapter in the *Oracle Containers for J2EE Configuration and Administration Guide* on how to define the parent application of a bean.

Sending a Message to a JMS Destination Using EJB 2.1

A client never accesses an MDB directly: rather, the client accesses an MDB by sending a message through the JMS destination (queue or topic) associated with the MDB.

To send a message to a JMS destination using EJB 2.1, do the following:

1. Look up both the JMS destination (queue or topic) and its connection factory.

   You can look up these resources using a predefined logical name or the explicit JNDI name you defined when you configured your JMS provider. Oracle recommends that you use logical names as shown in this procedure and its examples.

   For more information, see the following:

   - "Configuring an Environment Reference to a JMS Destination Resource Manager Connection Factory (JMS 1.1)" on page 19-13
   - "Configuring an Environment Reference to a JMS Destination or Connection Resource Manager Connection Factory (JMS 1.0)" on page 19-14
2. Use the connection factory to create a connection.

   If you are receiving messages for a queue, then start the connection.
3. Create a session over the connection.
4. Use the retrieved JMS destination to create a sender for a queue, or a publisher for a topic.
5. Create the message.
6. Send the message using either the queue sender or the topic publisher.
7. Close the queue session.
8. Close the connection.

Example 29–39 shows how a servlet client sends a message to a queue.

Example 29–39 Servlet Client Sends Message to a Queue

```
public final class testResourceProvider extends HttpServlet {

  private String resProvider = "myResProvider";
  private HashMap msgMap = new HashMap();
  Context ctx = new InitialContext();

  public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
```

```
  doPost(req, res);
}

public void doPost(HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException {
 // Retrieve the name of the JMS provider from the request,
 // which is to be used in creating the JNDI string for retrieval
  String rp = req.getParameter ("provider");
  if (rp != null) resProvider = rp;

  try {
    // 1a. Look up the Queue Connection Factory
    QueueConnectionFactory qcf = (QueueConnectionFactory)
        ctx.lookup("java:comp/resource/" + resProvider +
                   "/QueueConnectionFactories/myQCF");
    // 1b. Lookup the Queue
    Queue queue = (Queue)ctx.lookup("java:comp/resource/" +
                         resProvider + "/Queues/rpTestQueue");

    // 2a. Create queue connection using the connection factory
    QueueConnection qconn = qcf.createQueueConnection();
    // 2a. You are receiving messages, so start the connection
    qconn.start();

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

     // 4. Since this is for a queue, create a sender on top of the session
     //This is used to send out the message over the queue
    QueueSender snd = sess.createSender (q);

  drainQueue (sess, q);
  TextMessage msg = null;

  // Send msgs to queue
  for (int i = 0; i < 3; i++) {
    // 5. Create message
    msg = sess.createTextMessage();
    msg.setText ("TestMessage:" + i);

    // Set property of the recipient to be the MDB
    // and set the reply destination
    msg.setStringProperty ("RECIPIENT", "MDB");
    msg.setJMSReplyTo(q);

    //6. Send the message using the sender
    snd.send (msg);

    // You can store the messages IDs and sent-time  in a map (msgMap),
    // so that when messages are received, you can verify if you
    // *only* received those messages that you were
    // expecting. See receiveFromMDB() method where msgMap gets used
    msgMap.put( msg.getJMSMessageID(), new Long (msg.getJMSTimestamp()));
  }

  // receive a reply from the MDB
  receiveFromMDB (sess, q);

   // 7. Close sender, session, and connection for queue
   snd.close();
   sess.close();
   qconn.close();
  }
  catch (Exception e) {
    System.err.println ("** TEST FAILED **"+ e.toString());
    e.printStackTrace();
```

```
    }
    finally {
    }
  }

  // Receive any messages sent through the MDB
  private void receiveFromMDB (QueueSession sess, Queue q)
    throws Exception {
    // The MDB sends out a message (as a reply) to this client. The MDB sets
    // the receipient as CLIENT. Thus, you will only receive messages that have
    // RECIPIENT set to 'CLIENT'
    QueueReceiver rcv = sess.createReceiver (q, "RECIPIENT = 'CLIENT'");

    int nrcvd = 0;
    long trtimes = 0L;
    long tctimes = 0L;
    // First message needs to come from MDB. May take
    // a while receiving messages
    for (Message msg = rcv.receive (30000); msg != null; msg = rcv.receive (30000)) {
      nrcvd++;
      String rcp = msg.getStringProperty ("RECIPIENT");

      // Verify if messages in message Map
      // Check the msgMap to see if this is the message that you are expecting
      String corrid = msg.getJMSCorrelationID();
      if (msgMap.containsKey(corrid)) {
        msgMap.remove(corrid);
      }
      else {
        System.err.println ("** received unexpected message [" + corrid + "] **");
      }
    }
    rcv.close();
  }

  // Drain messages from queue
  private int drainQueue (QueueSession sess, Queue q)
    throws Exception {
    QueueReceiver rcv = sess.createReceiver (q);
    int nrcvd = 0;

    // First drain any old messages from queue
    for (Message msg = rcv.receive(1000); msg != null; msg = rcv.receive(1000))
      nrcvd++;

    rcv.close();

    return nrcvd;
  }
}
```

Accessing an EJB 2.1 EJBContext

For EJB 2.1 session, entity, and message-driven beans, you can access the `EJBContext` that OC4J provides by providing an appropriate getter and setter method when you implement your bean.

For more information, see the following:

- "What is EJB Context?" on page 1-6
- "Implementing the setSessionContext Method" on page 11-9
- "Implementing the setEntityContext and unsetEntityContext Methods" on page 13-20

- "Implementing the setMessageDrivenContext Method" on page 17-6

Handling Parameters

This section describes the following:

- Passing Parameters Into an Enterprise Bean
- Handling Parameters Returned by an Enterprise Bean

Passing Parameters Into an Enterprise Bean

When you implement an enterprise bean or write the client code that calls EJB methods, you must be aware of the parameter-passing conventions used with enterprise beans.

A parameter that you pass to a bean method (or a return value from a bean method) can be any Java type that is serializable. Java primitive types, such as `int`, `double`, are serializable. Any nonremote object that implements the `java.io.Serializable` interface can be passed. A nonremote object that is passed as a parameter to a bean, or returned from a bean, is passed by *value*, not by reference. So, for example, if you call a bean method as follows:

```
public class theNumber {
   int x;
}
...
bean.method1(theNumber);
```

then `method1()` in the bean receives a copy of `theNumber`. If the bean changes the value of `theNumber` object on the server, this change is not reflected back to the client, because of pass-by-value semantics.

If the nonremote object is complex (such as a class containing several fields) only the nonstatic and nontransient fields are copied.

When passing a remote object as a parameter, the stub for the remote object is passed. A remote object passed as a parameter must extend remote interfaces.

The next section demonstrates parameter passing to a bean, and remote objects as return values.

Handling Parameters Returned by an Enterprise Bean

The `EmployeeBean` method `getEmployee` returns an `EmpRecord` object, so this object must be defined somewhere in the application. In this example, an `EmpRecord` class is included in the same package as the EJB interfaces.

The class is declared as `public` and must implement the `java.io.Serializable` interface so that it can be passed back to the client by value as a serialized remote object. The declaration is as follows:

```
package employee;

public class EmpRecord implements java.io.Serializable {
  public String ename;
  public int empno;
  public double sal;
}
```

Note: The `java.io.Serializable` interface specifies no methods; it just indicates that the class is serializable. Therefore, there is no need to implement extra methods in the `EmpRecord` class.

Handling Exceptions

This section describes the following:

- Recovering From a NamingException While Accessing a Remote Enterprise Bean
- Recovering From a NullPointerException While Accessing a Remote Enterprise Bean
- Recovering From Deadlock Conditions

Recovering From a NamingException While Accessing a Remote Enterprise Bean

If you are trying to remotely access an enterprise bean and you receive an `javax.naming.NamingException` error, your JNDI properties are probably not initialized properly. See "Load Balancing" on page 2-31 for a discussion on setting up JNDI properties when accessing an enterprise bean from a remote object or remote servlet.

Recovering From a NullPointerException While Accessing a Remote Enterprise Bean

When accessing a remote enterprise bean from a Web application, you receive the following error: "`java.lang.NullPointerException: domain was null`". In this case, you must set an environment property in your client while accessing the enterprise bean set `dedicated.rmicontext` to `true`.

The following demonstrates how to use this additional environment property:

```
Hashtable env = new Hashtable( );
env.put (Context.INITIAL_CONTEXT_FACTORY,
  "oracle.j2ee.rmi.RMIInitialContextFactory");
env.put (Context.SECURITY_PRINCIPAL, "oc4jadmin");
env.put (Context.SECURITY_CREDENTIALS, "oc4jadmin");
env.put (Context.PROVIDER_URL, "ormi://myhost-us/ejbsamples");
env.put ("dedicated.rmicontext", "true"); // for 9.0.2.1 and later
Context context = new InitialContext (env);
```

See "Load Balancing" on page 2-31 for more information on `dedicated.rmicontext`.

Recovering From Deadlock Conditions

If the call sequence of several beans causes a deadlock scenario, OC4J notices the deadlock condition and throws a `Remote` exception that details the deadlock condition in one of the offending beans.

30

Using EJB and Web Services

This section describes the following:

- Exposing a Stateless Session Bean as a Web Service
- Accessing a Web Service From an Enterprise Bean

For more information, see the *Oracle Application Server Web Services Developer's Guide*.

Exposing a Stateless Session Bean as a Web Service

The client of a stateless session bean may be a Web service client. Only a stateless session bean may provide a Web service client view. A Web service client makes use of the enterprise bean's Web service client view, as described by a WSDL document. The bean's client view Web service endpoint interface is a JAX-RPC interface.

Using EJB 3.0, you can use annotations to easily expose a stateless session bean as a Web service (see "Using Annotations" on page 30-2).

Using EJB 2.1, you can also expose a stateless session bean as a Web service (see "Assembling a Web Service with EJBs" in the *Oracle Application Server Web Services Developer's Guide*).

Using Annotations

Using the `@WebService` and `@WebMethod` annotations, you can define a Web service endpoint interface, as Example 30–1 shows, and implement the Web service as a stateless session bean, as Example 30–2 shows.

Example 30–1 Annotated Web Service Endpoint Interface

```
package oracle.ejb30.ws;

import javax.ejb.Remote;
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
/**
* This is an Enterprise Java Bean Service Endpoint Interface
*/
public interface HelloServiceInf extends java.rmi.Remote {

    /**
    * @param phrase java.lang.String
    * @return java.lang.String
    * @throws String The exception description.
    */
```

```
    @WebMethod
    java.lang.String sayHello(java.lang.String name) throws java.rmi.RemoteException;
}
```

Example 30–2 Implementing the Web Service as a Stateless Session Bean

```
package oracle.ejb30.ws;

import java.rmi.RemoteException;
import java.util.Properties;
import javax.ejb.Stateless;

/**
* This is a session bean class
*/
@Stateless(name="HelloServiceEJB")
public class HelloServiceBean implements HelloServiceInf {

    public String sayHello(String name) {
        return("Hello "+name +" from first EJB3.0 Web Service");
    }
}
```

OC4J supports J2SE 5.0 Web Service annotations (also known as the Web Services Metadata for the Java Platform JSR-181) specification. The specification defines an annotated Java syntax for programming Web services.

For more information on using Web service annotations including Oracle extensions, see "Assembling Web Services with Annotations" in the *Oracle Application Server Web Services Developer's Guide*.

For other EJB Web service examples see the stateless session EJB Web service how-to or Adventure Builder how-to at `http://www.oracle.com/technology/tech/java/ejb30.html`.

Accessing a Web Service From an Enterprise Bean

From within an enterprise bean, you can obtain a Web service and invoke its methods.

Using EJB 3.0, you can use annotations and resource injection (see "Using Annotations" on page 30-2) without having to create an environment reference for the Web service.

Using EJB 2.1, you must use the initial context (see "Using Initial Context" on page 30-3) and you must create an environment reference for the Web service (see "Configuring an Environment Reference to a Web Service" on page 19-17) before you can look it up.

For more information, see "Assembling a J2EE Web Service Client " in the *Oracle Application Server Web Services Developer's Guide*.

Using Annotations

Given the Web service that Example 30–3 shows, you can access the Web service from an EJB 3.0 stateless session bean using resource injection, as Example 30–4 shows.

Example 30–3 Annotating a Web Service

```
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class StockQuoteProvider {
```

```
    @WebMethod
    public Float getLastTradePrice() {
        ...
    }
}
```

Example 30–4 Calling Out to a Web Service Obtained by Resource Injection

```
@Stateless
public class InvestmentBean implements Investment {

    public void checkPortfolio(...) {
        ...
        @Resource StockQuoteProvider sqp;

        // Get a quote
        Float quotePrice = sqp.getLastTradePrice(...);
        ...
    }
}
```

Using Initial Context

After you define an environment reference to a Web service (see "Configuring an Environment Reference to a Web Service" on page 19-17), you can use the initial context to look up the Web service and invoke its methods from within your stateless session bean, as Example 30–5 shows.

Example 30–5 Calling Out to a Web Service Obtained from the Initial Context

```
@Stateless
public class InvestmentBean implements Investment {

    public void checkPortfolio(...) {
        ...
        // Obtain the default initial JNDI context
        Context initCtx = new InitialContext();
        // Look up the stock quote service in the environment
        com.example.StockQuoteService sqs = (com.example.StockQuoteService)initCtx.lookup(
            "java:comp/env/service/StockQuoteService");
        // Get the stub for the service endpoint
        com.example.StockQuoteProvider sqp = sqs.getStockQuoteProviderPort();
        // Get a quote
        float quotePrice = sqp.getLastTradePrice(...);
        ...
    }
}
```

31

Administrating an EJB Application

This chapter describes the following:

- OC4J EJB JMX Support
- Using Oracle Enterprise Manager 10g Application Server Control
- Configuring EJB Logging
- Managing the Bean Instance Pool
- Starting and Stopping an EJB Application
- Troubleshooting an EJB Application

For more information, see "Understanding EJB Administration" on page 2-12.

OC4J EJB JMX Support

OC4J deploys MBeans to collect JSR77 statistics and Oracle Dynamic Monitoring System (DMS) sensor data for all types of EJB.

You can access these statistics and sensors using any JMX-compliant management tool, such as the Application Server Control (see "Using Oracle Enterprise Manager 10g Application Server Control" on page 31-1).

Using Oracle Enterprise Manager 10*g* Application Server Control

The Application Server Control is a JMX-compliant, Web-based user interface for deploying, configuring and monitoring applications within OC4J, as well as managing the OC4J server instance and the Web services used by your applications.

Using the Application Server Control JMX administrative task, you can modify properties of all EJB types deployed to OC4J without having to restart Oracle Application Server or redploy your application, as follows:

1. Launch Application Server Control.
2. Click the **Administration** link.
3. Click **System MBean Browser**.
4. Specific MBean instances are accessed through the navigation pane to the left of the console. Expand a node in the navigation pane and drill down to the MBean you wish to access.

   For example, for a standalone OC4J, select: **J2EEServer** > **standalone** > **J2EEApplication** > *application-name* > **EJBModule** > *module-name*

5. Select the type of an enterprise bean, such as **StatelessSessionBean**, **MessageDrivenBean**, or **WebServicePort**.
6. Select an MBean instance.
7. Click the appropriate tab in the right-hand pane:
   - Click the **Attributes** tab to access the MBean's attributes. If you modify any attribute values, click **Apply Changes** to apply your changes to the OC4J runtime.
   - Click the **Operations** tab to access the MBean's operations. After selecting a specific operation, click the **Invoke** to call it.
   - Click the **Notifications** tab to subscribe to the MBean's notifications. After selecting a specific notification, click **Apply** to subscribe to it.
   - Click the **Statistics** tab to view the MBean's statistics.

You can use Application Server Control for most administration tasks.

For more information, see the following:

- "Oracle Enterprise Manager 10g Application Server Control Console" in the *Oracle Containers for J2EE Configuration and Administration Guide*
- the online Help provided with Application Server Control

Configuring EJB Logging

OC4J uses the standard JDK `java.util.logging` package and, by default, writes log messages to the `<OC4J_HOME>/j2ee/home/log/<group>/oc4j/log.xml` file.

This section describes the following:

Logging Namespaces

You can configure loggers for the following `java.util.logging` namespaces:

- `oracle.j2ee.ejb.annotation`
- `oracle.j2ee.ejb.compilation`
- `oracle.j2ee.ejb.database`
- `oracle.j2ee.ejb.deployment`
- `oracle.j2ee.ejb.lifecycle`
- `oracle.j2ee.ejb.pooling`
- `oracle.j2ee.ejb.runtime`
- `oracle.j2ee.ejb.transaction`

Logging Levels

You can configure the following log levels: `FINER`, `FINE`, `CONFIG`, `INFO`, `WARNING`, and `SEVERE`.

Configuring Logging With Application Server Control Logging MBean

The simplest way to configure OC4J logging is to use Application Server Control (see "Using Oracle Enterprise Manager 10g Application Server Control" on page 31-1).

Application Server Control shows all EJB-related logger names, and you can specify attributes such as log level using the Application Server Control interface.

Configuring Logging Using the j2ee-logging.xml File

You can configure OC4J logging using the `<OC4J_HOME>/j2ee/home/config/j2ee-logging.xml` file, as Example 31–1 shows.

Example 31–1 j2ee-logging.xml File

```
<logger
    name='oracle.j2ee.ejb'
    level='NOTIFICATION:1'
    useParentHandlers='false'>
    <handler name='oc4j-handler'/>
    <handler name='console-handler'/>
</logger>
```

For more information, see the following:

- "Logging Namespaces" on page 31-2
- "Logging Levels" on page 31-3

Configuring Logging Using System Properties

You can configure OC4J logging using the `oracle.j2ee.logging` system property. This system property has the following format:

```
oracle.j2ee.logging.<log-level>=<log-namespace>
```

where:

- `<log-level>` is one of `fine`, `finer`, or `finest`.
- `<log-namspace>` is an `oracle.j2ee.ejb` namespace (see "Logging Namespaces" on page 31-2).

Example 31–2 shows how to configure the logger for the `oracle.j2ee.ejb.deployment` namespace to finest.

Example 31–2 Configuring a Logger with a System Property

```
oracle.j2ee.logging.finest=oracle.j2ee.ejb.deployment
```

Configuring TopLink Logging

For EJB 3.0 JPA applications, you can use vendor extensions to customize how the TopLink JPA persistence provider logs.

For more information, see "TopLink JPA Extensions for Logging" on page 26-13.

Configuring Oracle JMS Connector Logging

For applications that use the Oracle JMS Connector to access a JMS message service, you can customize how the Oracle JMS Connector logs using activation configuration property `LogLevel`.

For more information, see "Configuring an EJB 3.0 MDB to Access a Message Service Provider Using J2CA" on page 10-1.

Managing the Bean Instance Pool

OC4J provides EJB pooling attributes that you can configure to improve performance by reducing the frequency of bean instance creation.

This section describes the following:

- Configuring Bean Instance Pool Size
- Configuring Bean Instance Pool Timeouts for Session Beans
- Configuring Bean Instance Pool Timeouts for Entity Beans

Configuring Bean Instance Pool Size

You can set the minimum and maximum number of the bean instance pool for session beans, entities, and message-driven beans.

You can configure the bean pool size as follows:

- Using Oracle Enterprise Manager 10g Application Server Control
- Using Annotations
- Using Deployment XML

Configuration in the deployment XML overrides the corresponding configuration made using annotations.

Using Annotations

You can specify bean instance pool size for EJB 3.0 session and message-driven beans using the following OC4J-proprietary annotations and their attributes:

- `@StatelessDeployment` attributes:
  - `maxInstances`
  - `minInstances`

  For more information about these attributes, see Table A–1.
- `@StatefulDeployment` attributes:
  - `maxInstances`
  - `maxInstancesThreshold`

  For more information about these attributes, see Table A–1.
- `@MessageDrivenDeployment` attributes:
  - maxInstances
  - minInstances

  For more information about these attributes, see Table A–3.

Example 31–3 shows how to configure these attributes for an EJB 3.0 stateless session bean using the `@StatelessDeployment` annotation.

Example 31–3 @StatelessDeployment poolCacheTimeout Attribute

```
import javax.ejb.Stateless;
import oracle.j2ee.ejb.StatelessDeployment;

@Stateless
@StatelessDeployment(
    maxInstances=10,
    minInstances=3
)
public class HelloWorldBean implements HelloWorld {

    public void sayHello(String name) {
        System.out.println("Hello "+name +" from first EJB3.0");
    }
}
```

Using Deployment XML

You can specify bean instance pool size for EJB 3.0 session and message-driven beans using the following `orion-ejb-jar.xml` file elements and their attributes:

- `<session-deployment>` attributes for stateless session beans:
  - `max-instances`
  - `min-instances`

  For more information about these attributes, see Table A–1.
- `<session-deployment>` attributes for stateful session beans:
  - `max-instances`
  - `max-instances-threshold`

  For more information about these attributes, see Table A–1.
- `<message-driven-deployment>` attributes for message-driven beans:
  - `max-instances`
  - `min-instances`

  For more information about these attributes, see Table A–3.

Example 31–4 shows how to configure these attributes for an EJB 3.0 stateless session bean using the `orion-ejb-jar.xml` file.

Example 31–4 orion-ejb-jar.xml for Bean Instance Pool Size for a Stateless Session Bean

```
<?xml version="1.0" encoding="utf-8"?>
<orion-ejb-jar
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/orion-ejb-jar-10_
0.xsd"
    deployment-version="10.1.3.1.0"
    deployment-time="10b1fb5cdd0"
    schema-major-version="10"
    schema-minor-version="0"
>
    <enterprise-beans>
        <session-deployment
```

```
            max-instances="10"
            min-instances="3"
            ...
        >
        </session-deployment>
    ...
    </enterprise-beans>
    ...
</orion-ejb-jar>
```

If you change this property using this method, you must restart OC4J to apply your changes. Alternatively, you can use Application Server Control Console to modify this parameter dynamically without restarting OC4J (see "Using Oracle Enterprise Manager 10g Application Server Control" on page 31-1).

Configuring Bean Instance Pool Timeouts for Session Beans

You can set the maximum amount of time that session beans are cached in the bean instance pool.

You can configure pool timeouts for session beans as follows:

- Using Oracle Enterprise Manager 10g Application Server Control
- Using Annotations
- Using Deployment XML

Configuration in the deployment XML overrides the corresponding configuration made using annotations.

Using Annotations

Example 31–5 shows how to configure the bean instance pool timeout for an EJB 3.0 stateless session bean using the `@StatelessDeployment` annotation `poolCacheTimeout` attribute.

For more information on this `@StatelessDeployment` attribute, see Table A–1. For more information on the `@StatelessDeployment` annotation, see "Configuring OC4J-Proprietary Deployment Options on an EJB 3.0 Session Bean" on page 5-10.

Example 31–5 @StatelessDeployment poolCacheTimeout Attribute

```
import javax.ejb.Stateless;
import oracle.j2ee.ejb.StatelessDeployment;

@Stateless
@StatelessDeployment(
    poolCacheTimeout=90
)
public class HelloWorldBean implements HelloWorld {

    public void sayHello(String name) {
        System.out.println("Hello "+name +" from first EJB3.0");
    }
}
```

Example 31–6 shows how to configure the bean instance pool timeout for an EJB 3.0 stateful session bean using the `@StatefulDeployment` annotation `timeout` attribute.

For more information on this `@StatelessDeployment` attribute, see Table A–1. For more information on the `@StatelessDeployment` annotation, see "Configuring OC4J-Proprietary Deployment Options on an EJB 3.0 Session Bean" on page 5-10.

Example 31–6 @StatefulDeployment timeout Attribute

```
import javax.ejb.Stateful
import oracle.j2ee.ejb.StatefulDeployment;

@Stateful
@StatefulDeployment(
    timeout=100
)
public class CartBean implements Cart {

    private ArrayList items;
    ...
}
```

Using Deployment XML

In the `orion-ejb-jar.xml` file you set the bean pool timeout with the following attributes of the `<session-deployment>` element for session beans:

- The `pool-cache-timeout` attribute is applicable to stateless session beans and sets how long to keep stateless sessions cached in the pool. The default is 0 seconds, which means never timeout.

  For example, if you wanted to set the `pool-cache-timeout` to 90 seconds, you would do as follows:

  ```
  <session-deployment ...  pool-cache-timeout="90"
    ...
  </session-deployment>
  ```

- The `timeout` attribute is applicable to stateful session beans and sets how long a stateful session bean can remain inactive before it is removed from the bean instance pool. The default is 1800 seconds.

  For example, if you wanted to set the stateful session bean inactivity timeout to 900 seconds, you would do as follows:

  ```
  <session-deployment ...  timeout="900"
    ...
  </session-deployment>
  ```

If you change this property using this method, you must restart OC4J to apply your changes. Alternatively, you can use Application Server Control Console to modify this parameter dynamically without restarting OC4J (see "Using Oracle Enterprise Manager 10g Application Server Control" on page 31-1).

Configuring Bean Instance Pool Timeouts for Entity Beans

You can set the maximum amount of time that entities are cached in the bean instance pool.

You can configure pool timeouts for entities as follows:

- Using Oracle Enterprise Manager 10g Application Server Control
- Using Deployment XML

Using Deployment XML

In the `orion-ejb-jar.xml` file you set the bean pool timeout with the following attributes of the `<entity-deployment>` element for entities:

- The `pool-cache-timeout` attribute sets how long entity bean implementation instances are to be kept in the "pooled" (unassigned) state. The default is 60 seconds. Setting this attribute to `never` means never timeout.

  For example, if you wanted to set the `pool-cache-timeout` for entities to 90 seconds, you would do as follows:

```
<entity-deployment ...  pool-cache-timeout="90"
  ...
</entity-deployment>
```

If you change this property using this method, you must restart OC4J to apply your changes. Alternatively, you can use Application Server Control Console to modify this parameter dynamically without restarting OC4J (see "Using Oracle Enterprise Manager 10g Application Server Control" on page 31-1).

Starting and Stopping an EJB Application

You can use Application Server Control to stop and start an EJB application.

While an application is stopped, clients cannot access it.

For more information, see "Using Oracle Enterprise Manager 10g Application Server Control" on page 31-1.

Troubleshooting an EJB Application

This section describes the following:

- Validating XML Files
- Debugging the ejb-jar.xml File
- Debugging Generated Wrapper Code

Validating XML Files

To configure OC4J to validate XML files, add the `-validateXML` option to the command line used in the OC4J start up script (`<OC4J_HOME>/BIN/oc4j.cmd` or `oc4j`).

Example 31–7 shows how to set this option in the `oc4j.cmd` file.

Example 31–7 Setting -validateXML in oc4j.cmd

```
...
"%JAVA_HOME%\bin\java" %JVMARGS% -jar %OC4J_JAR% %CMDARGS% -validateXML
...
```

With this option set, OC4J strictly validates XML files against their specified schema when OC4J reads them. OC4J logs any errors (see "Configuring EJB Logging" on page 31-2).

Debugging the ejb-jar.xml File

When you deploy an EJB 3.0 application with one or more annotations, OC4J will automatically write its in-memory `ejb-jar.xml` file to the same location as the `orion-ejb-jar.xml` file in the deployment directory: `<`*`ORACLE_HOME`*`>/j2ee/home/application-deployments/my_application/META-INF`.

This `ejb-jar.xml` file represents configuration obtained from both annotations and a deployed `ejb-jar.xml` file (if present).

When you deploy an EJB 2.1 application, to preserve generated wrapper code, you musts set system property `KeepWrapperCode` (see "Debugging Generated Wrapper Code" on page 31-9).

See also "Validating XML Files" on page 31-8.

Debugging Generated Wrapper Code

By default, when OC4J deploys an EJB 2.1 CMP application, it generates wrapper code in `<OC4J_HOME>/j2ee/home/application-deployments/<ear-name>/<ejb-name>/generated`, compiles it, creates a JAR file that contains the compiled classes, and then deletes the wrapper code it generates.

You can configure OC4J to preserve the wrapper code that it generates. Examining the wrapper code can aid in debugging some application problems.

This section describes the following:

- Preserving Generated Wrapper Code in the Default Directory
- Preserving Generated Wrapper Code in a Directory You Specify
- Modifying Generated Wrapper Code
- Disabling Generated Wrapper Code Preservation

> **Note:** Debugging generated wrapper code is deprecated in this release.
>
> These options apply only to EJB 2.1 entity beans with container-managed persistence: they do not apply to session beans, message-driven beans, or EJB 3.0 entities. OC4J generates only one file for each EJB 2.1 entity bean with container-managed persistence. OC4J does not generate any artifacts if you use only EJB 3.0 entities.

Preserving Generated Wrapper Code in the Default Directory

To configure OC4J to preserve generated code, set system property `KeepWrapperCode` to `true` on the OC4J startup command line, as Example 31–8 shows for the `<OC4J_HOME>/bin/oc4j.cmd` file.

Example 31–8 Setting KeepWrapperCode in oc4j.cmd

```
...
"%JAVA_HOME%\bin\java" %JVMARGS% -DKeepWrapperCode=true -jar "%OC4J_JAR%" %CMDARGS%
...
```

When `KeepWrapperCode` is `true`, OC4J preserves the wrapper code it generates in the default directory `<OC4J_HOME>/j2ee/home/application-deployments/<ear-name>/<ejb-name>/generated`. Alternatively, you can specify the directory OC4J uses to preserve wrapper code (see "Preserving Generated Wrapper Code in a Directory You Specify" on page 31-10).

If you undeploy your application, OC4J deletes the wrapper code in this directory.

Preserving Generated Wrapper Code in a Directory You Specify

If you set both system property `KeepWrapperCode` to `true` and system property `WrapperCodeDir` to a directory (call it `<specified-wrapper-dir>`), OC4J generates wrapper code to this directory and preserves the wrapper code even if you undeploy the application, as Example 31–9 shows for the `<OC4J_HOME>/bin/oc4j.cmd` file.

Example 31–9 Setting KeepWrapperCode and WrapperCodeDir in oc4j.cmd

```
...
"%JAVA_HOME%\bin\java" %JVMARGS% -DKeepWrapperCode=true -DWrapperCodeDir=C:\wrappers -jar
"%OC4J_JAR%" %CMDARGS%
...
```

The `<specified-wrapper-dir>` may be absolute (such as `C:\wrappers`) or relative (such as `./wrappers`): relative paths are relative to `<OC4J_HOME>/j2ee/home`.

If OC4J cannot generate to the directory you specify (for example, due to a permission problem or lack of space), OC4J generates wrapper code to the default directory `<OC4J_HOME>/j2ee/home/application-deployments/<ear-name>/<ejb-name>/generated` and preserves this wrapper code even if you undeploy the application.

Modifying Generated Wrapper Code

If you set both system property `KeepWrapperCode` to `true` and system property `DoNotReGenerateWrapperCode` to `true`, OC4J generates wrapper code and preserves the wrapper code even if you undeploy the application, as Example 31–10 shows for the `<OC4J_HOME>/bin/oc4j.cmd` file. In this case, when you redeploy, OC4J will not regenerate wrapper code, but instead will use the version of wrapper code in the default directory ("Preserving Generated Wrapper Code in the Default Directory" on page 31-9) or in the directory you specified (see "Preserving Generated Wrapper Code in a Directory You Specify" on page 31-10).

Example 31–10 Setting KeepWrapperCode and DoNotReGenerateWrapperCode in oc4j.cmd

```
...
"%JAVA_HOME%\bin\java" %JVMARGS% -DKeepWrapperCode=true -DDoNotReGenerateWrapperCode=true
-jar "%OC4J_JAR%" %CMDARGS%
...
```

Using these system properties, you can modify wrapper code, for example, to add debugging statements, and when you redeploy, OC4J recompiles and uses the preserved version of wrapper code that you modified.

Disabling Generated Wrapper Code Preservation

To disable generated wrapper code perservation, set system property `KeepWrapperCode` to `false` and system property `DoNotReGenerateWrapperCode` to `false`, or leave these system properties unset.

32

Optimizing EJB Performance

This chapter briefly summarizes some of the important options that you can use to improve EJB performance, including the following:

- Session Bean Performance
- JPA Entity Performance
- Performance of an EJB 2.1 Entity Bean With Container-Managed Persistence
- Performance of an EJB 2.1 Entity Bean With Bean-Managed Persistence
- Message-Driven Bean Performance

For complete performance information, see the *Oracle Application Server Performance Guide*.

Session Bean Performance

To improve session bean performance, consider the following:

- Bean Instance Pooling
- Singleton Interceptors

Bean Instance Pooling

For session beans, using bean instance pooling can increase performance by reducing bean creation overhead. For more information, see "Managing the Bean Instance Pool" on page 31-4.

Singleton Interceptors

For EJB 3.0 session beans, if you are using interceptors and your interceptors are stateless, you can specify singleton interceptors. This OC4J EJB 3.0 extension creates a singleton for each interceptor class. This singleton is shared by all session bean instances that use that interceptor class. This reduces memory requirements and life cycle management. For more information, see "Singleton Interceptors" on page 2-12.

JPA Entity Performance

To improve JPA entity performance, consider the following:

- Bean Instance Pooling
- Fetch Type

Bean Instance Pooling

For EJB 3.0 entities, using bean instance pooling can increase performance by reducing bean creation overhead. For more information, see "Managing the Bean Instance Pool" on page 31-4.

Fetch Type

For all EJB 3.0 mapping types, you can define the strategy for fetching data from the database as either lazy or eager. This can help improve performance when you know that certain portions of an entity are infrequently accessed. This is especially valuable for relationship mappings where it can reduce the amount of SQL that is executed, reduce query execution time, and reduce object loading time. For more information, see "Configuring Lazy Loading" on page 7-16.

Performance of an EJB 2.1 Entity Bean With Container-Managed Persistence

To improve performance of an EJB 2.1 entity bean with container-managed persistence, consider the following:

- Bean Instance Pooling
- Read-Only Entity Beans With Container-Managed Persistence

For more information, see "Improving EJB CMP 2.1 Performance" in the *Oracle Application Server Performance Guide*.

Bean Instance Pooling

For EJB 2.1 entity beans with container-managed persistence, using bean instance pooling can increase performance by reducing bean creation overhead. For more information, see "Managing the Bean Instance Pool" on page 31-4.

Read-Only Entity Beans With Container-Managed Persistence

For EJB 2.1 entity beans with container-managed persistence that do not change after activation, you can specify a locking mode of read-only. For more information, see "Concurrency (Locking) Mode" on page 1-60.

Performance of an EJB 2.1 Entity Bean With Bean-Managed Persistence

To improve performance of an EJB 2.1 entity bean with bean-managed persistence, consider the following:

- Read-Only Entity Beans With Bean-Managed Persistence
- Commit Option A

Read-Only Entity Beans With Bean-Managed Persistence

For EJB 2.1 entity beans with bean-managed persistence that do not change after activation, you can specify the entity bean as read-only. When you configure an entity bean with bean-managed persistence as read-only, OC4J uses a special case of commit option A to improve performance by performing the following:

- caching the instance;

- not calling `ejbLoad` after activation;
- not updating the instance or calling `ejbStore` when the transaction commits.

For more information, see "Configuring a Read-Only Entity Bean With Bean-Managed Persistence" on page 15-4.

Commit Option A

For EJB 2.1 BMP applications, you can configure the BMP commit option as A or C. Commit option A offers a performance improvement by postponing a call to `ejbLoad` method. For more information, see "Commit Options and BMP Applications" on page 1-50.

If you configure a read-only entity bean with bean-managed persistence to use commit option A, you can further improve performance by taking advantage of caching of the read-only entity bean with bean-managed persistence. For more information, see "Read-Only Entity Beans With Bean-Managed Persistence" on page 32-2.

Message-Driven Bean Performance

To improve message-driven bean performance, consider the following:

- Bean Instance Pooling
- Singleton Interceptors

For more information, see "Improving MDB Performance" in the *Oracle Application Server Performance Guide*.

Bean Instance Pooling

For message-driven beans, using bean instance pooling can increase performance by reducing bean creation overhead. For more information, see "Managing the Bean Instance Pool" on page 31-4.

Singleton Interceptors

For EJB 3.0 message-driven beans, if you are using interceptors and your interceptors are stateless, you can specify singleton interceptors. This OC4J EJB 3.0 extension creates a singleton for each interceptor class. This singleton is shared by all message-driven bean instances that use that interceptor class. This reduces memory requirements and life cycle management. For more information, see "Singleton Interceptors" on page 2-12.

A

XML Reference for orion-ejb-jar.xml Elements

This appendix describes the elements contained within the OC4J-specific EJB deployment descriptor `orion-ejb-jar.xml`, including the following:

For more information, see the following:

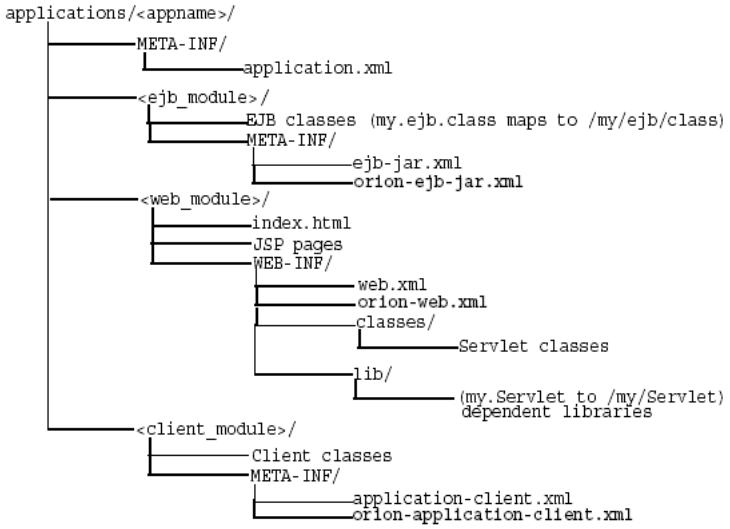
- "Understanding EJB Deployment Descriptor Files" on page 2-4
- `http://www.oracle.com/technology/oracleas/schema/index.html`

OC4J and the orion-ejb-jar.xml File

Whenever you deploy an application, OC4J automatically generates the OC4J-specific XML file with the default elements. If you want to change these defaults, you must copy the `orion-ejb-jar.xml` file to where your original `ejb-jar.xml` file is located and change it in this location. If you change the XML file within the deployed location, OC4J overwrites these changes when the application is deployed again. The changes only stay constant when changed in the development directories.

Oracle recommends that you add your OC4J-specific XML files within the recommended development structure, as Figure A–1 shows.

Figure A–1 Development Application Directory Structure



TopLink Persistence Support

Table A–2 describes all the attributes of the `orion-ejb-jar.xml` file `<entity-deployment>` element and indicates which options you configure in the `orion-ejb-jar.xml` file and which you configure using TopLink persistence API.

For example:

- To configure `<entity-deployment>` attribute `call-timeout`, you must use the corresponding TopLink persistence API. If you set the `call-timeout` attribute in the `orion-ejb-jar.xml` file, OC4J will ignore it.
- To configure `<entity-deployment>` attribute `clustering-schema`, you must use the `orion-ejb-jar.xml` file; there is no corresponding TopLink persistence API.

For EJB 3.0 applications, you access TopLink persistence API by augmenting `orion-ejb-jar.xml` configuration with TopLink-specific deployment descriptor files `ejb3-toplink-sessions.xml` and `toplink-ejb-jar.xml`. For more information, see "Customizing the JPA Persistence Provider" on page 3-3.

For EJB 2.1 applications, you access TopLink persistence API using `orion-ejb-jar.xml` element `pm-properties`. For more information, see "Customizing the TopLink EJB 2.1 Persistence Manager" on page 3-13.

Note: To modify TopLink deployment descriptor files, use the TopLink Workbench.

For more information, see the following:

- "Migrating OC4J Orion Persistence to OC4J TopLink Persistence" in the *Oracle TopLink Developer's Guide*
- "Understanding TopLink Workbench" in the *Oracle TopLink Developer's Guide*
- TopLink-specific deployment descriptor XML schema documents located at `<OC4J_HOME>\toplink\config\xsds`.

<orion-ejb-jar>

The OC4J-specific deployment descriptor contains extended deployment information for session beans, entity beans, message driven beans, and security for these enterprise beans. The major element structure within this deployment descriptor has the following structure:

```
<orion-ejb-jar deployment-time=... deployment-version=...>
 <enterprise-beans>
   <persistence-manager ...></persistence-manager>
   <session-deployment ...></session-deployment>
   <entity-deployment ...></entity-deployment>
   <message-driven-deployment ...></message-driven-deployment>
 </enterprise-beans>
 <assembly-descriptor>
   <security-role-mapping ...></security-role-mapping>
   <default-method-access></default-method-access>
 </assembly-descriptor>
</orion-ejb-jar>
```

Each section under the `<orion-ejb-jar>` main tag has its own purpose. These are described in the following sections:

- <enterprise-beans>
- <assembly-descriptor>

<enterprise-beans>

The `<enterprise-beans>` section defines additional deployment information for all enterprise beans: session beans, entity beans, and message driven beans. There is a section for each type of EJB.

The following sections describe the elements within `<enterprise-beans>` element:

- <persistence-manager>
- <session-deployment>
- <entity-deployment>
- <message-driven-deployment>
- <cmp-field-mapping>

<persistence-manager>

The `<persistence-manager>` section provides additional deployment information for the TopLink persistence manager for EJB 2.1 applications only. For EJB 3.0 applications, OC4J always uses the TopLink entity manager.

The `<persistence-manager>` section contains the following structure:

```
<persistence-manager name=... class=... descriptor=... >
  <pm-properties>
    <session-name>...</session-name>
    <project-class>...</project-class>
    <db-platform-class>...</db-platform-class>
    <default-mapping db-table-gen=... >...</default-mapping>
    <remote-relationships>...</remote-relationships>
    <cache-synchronization mode=... >...</cache-synchronization>
    <customization-class>...</customization-class>
  </pm-properties>
```

```
</persistence-manager>
```

Multiple definitions of the `<persistence-manager>` element are not valid. If OC4J detects multiple definitions of the `<persistence-manager>` element at parse time, OC4J logs a warning message. In this case, OC4J uses only the first entry and ignores any subsequent entries.

If you want to explicitly specify the persistence manager, use the `<persistence-manager>` element `name` attribute. The following are valid values:

- `toplink`: selects the TopLink persistence manager (default).
- `orion`: selects the deprecated Orion persistence manager.

If you are using the TopLink persistence manager and you name your TopLink deployment descriptor something other than `toplink-ejb-jar.xml` (see "What is the toplink-ejb-jar.xml File?" on page 2-6), specify the name using the `<persistence-manager>` element `descriptor` attribute.

The `<pm-properites>` element applies only to the TopLink persistence manager.

For more information, see the following:

- "Understanding EJB Persistence Services" on page 2-12
- "Customizing the JPA Persistence Provider" on page 3-3
- "Customizing the TopLink EJB 2.1 Persistence Manager" on page 3-13
- "Configuring pm-properties" in the *Oracle TopLink Developer's Guide*

<session-deployment>

The `<session-deployment>` section provides additional deployment information for a session bean deployed within this JAR file.

The `<session-deployment>` section contains the following structure:

```
<session-deployment pool-cache-timeout=... call-timeout=... copy-by-value=...
      location=... max-instances=... min-instances=... max-tx-retries=...
      tx-retry-wait=... name=... persistence-filename=... replication=...
      timeout=... idletime=... memory-threshold=... max-instances-threshold=...
      resource-check-interval=... passivate-count=... wrapper=...
      local-wrapper=... interceptor-type= ...
  <ior-security-config>
    <transport-config>
      <integrity></integrity>
      <confidentiality></confidentiality>
      <establish-trust-in-target></establish-trust-in-target>
      <establish-trust-in-client></establish-trust-in-client>
    </transport-config>
    <as-context>
      <auth-method></auth-method>
      <realm></realm>
      <required></required>
    </as-context>
    <sas-context>
      <caller-propagation></caller-propagation>
    </sas-context>
  </ior-security-config>
  <env-entry-mapping name=...> </env-entry-mapping
  <ejb-ref-mapping location=... name=... remote=... jndi-properties=... />
  <resource-ref-mapping location=... name=... >
```

```
    <lookup-context location=...>
       <context-attribute name=... value=... />
    </lookup-context>
  </resource-ref-mapping>
  <resource-env-ref-mapping location=... name=... />
  <message-destination-ref-mapping location=... name=... />
</session-deployment>
```

> **Note:** Alternatively, in an EJB 3.0 application, you can use the OC4J-proprietary annotations `@StatelessDeployment` and `@StatefulDeployment`. You can use the `orion-ejb-jar.xml` file `<session-deployment>` configuration to override `@StatelessDeployment` and `@StatefulDeployment` configuration. For more information, see "Configuring OC4J-Proprietary Deployment Options on an EJB 3.0 Session Bean" on page 5-10.

For information on each of these elements and sub-elements, see the following:

- <session-deployment> Attributes
- <ior-security-config>
- <env-entry-mapping>
- <ejb-ref-mapping>
- <resource-ref-mapping>
- <resource-env-ref-mapping>
- <message-destination-ref-mapping>

Examples

For session bean examples, which include `<session-deployment>`, `@StatefulDeployment`, or `@StatelessDeployment` configuration (where relevant), see the following:

- "Implementing an EJB 3.0 Session Bean" on page 4-1
- "Implementing an EJB 2.1 Session Bean" on page 11-1
- "Configuring OC4J-Proprietary Deployment Options on an EJB 3.0 Session Bean" on page 5-10

<session-deployment> Attributes

Table A–1 lists the attributes for the `<session-deployment>` element, their `@StatelessDeployment` and `@StatefulDeployment` annotation attribute equivalents (where appropriate), and indicates which are applicable to stateless session beans only, stateful session beans only, or both.

Table A–1 Attributes for the <session-deployment> Element

Attribute	@StatelessDeployment Equivalent	@StatefulDeployment Equivalent	Stateless	Stateful	Description
`call-timeout`	`callTimeout`	`callTimeout`	✔	✔	This parameter specifies the maximum time to wait for any resource to make a business/life cycle method invocation. This is not a timeout for how long a business method invocation can take. If the timeout is reached, a `TimedOutException` is thrown. This excludes database connections. The default value is 90000 milliseconds. Set to `0` if you want the timeout to be forever. See the EJB section in the *Oracle Application Server Performance Guide* for more information.
`copy-by-value`	`copyByValue`	`copyByValue`	✔	✔	Whether or not to copy (clone) all the incoming and outgoing parameters in EJB calls. Set to `false` if you are certain that your application does not assume copy-by-value semantics for a speed-up. The default value is `true`.
`idletime`		`idletime`		✔	You can set an idle timeout for each bean. When this timeout expires, passivation occurs. Set this attribute to the appropriate number of seconds. Default: 300 seconds. (5 minutes). To disable, specify any negative number.
`interceptor-type`	`interceptorType`	`interceptorType`	✔	✔	The attribute indicates how OC4J handles interceptor class life cycle. You can set the interceptor type to `bean` (stateful class) or `singleton` (stateless class). When set to `bean`, OC4J creates a separate interceptor class instance for each session bean instance that you associate with that interceptor class. This is in accordance with the EJB 3.0 specification. In this case, the interceptor class must be stateful. When set to `singleton`, OC4J creates a single interceptor class instance that all session bean instances share. In this case, the interceptor class must be stateless. For more information, see "Singleton Interceptors" on page 2-12.
`local-location`	`localLocation`	`localLocation`	✔	✔	The local JNDI name, to which this enterprise bean will be bound.
`local-wrapper`			✔	✔	Name of the OC4J local home wrapper class for this bean. This is an internal server value and should not be edited.
`location`	`location`	`location`	✔	✔	The JNDI-name to which this bean will be bound.

Table A–1 (Cont.) Attributes for the <session-deployment> Element

Attribute	@StatelessDeployment Equivalent	@StatefulDeployment Equivalent	Stateless	Stateful	Description
max-instances	maxInstances	maxInstances	✔	✔	The number of bean instances allowed in memory: either instantiated or pooled. When this value is reached, OC4J attempts to passivate beans using the least recently used (LRU) algorithm. To allow an infinite number of bean instances, the max-instances attribute can be set to zero. Default is 0, which means infinite. This applies to both stateless and stateful session beans. To disable instance pooling, set max-instances to any negative number. This will create a new instance at the start of the EJB call and release it at the end of the call. For more information, see the following: ■ "Configuring Passivation Criteria" on page 12-2 ■ "Configuring Bean Instance Pool Size" on page 31-4
max-instances-threshold		maxInstancesThreshold		✔	Percentage of max-instances number of beans that can be in memory before passivation occurs. Specify an integer that is translated as a percentage. If you define that the max-instances is 100 and the max-instances-threshold is 90%, then when the active bean instances is greater than or equal to 90, passivation of beans occurs. Default: 90%. To disable, specify any negative number.
max-tx-retries	maxTransactionRetries	maxTransactionRetries	✔	✔	This parameter specifies the number of times to retry a transaction that was rolled back due to system-level failures. The default is 0. For a stateful session bean, if a RuntimeException, Error, or RemoteException is thrown, the OC4J does not do a retry. Generally, Oracle recommend that you add retries only where errors are seen that could be resolved through retries. For example, if you are using serializable isolation and you want to retry the transaction automatically if there is a conflict, you might want to use retries. However, if the bean wants to be notified when there is a conflict, then in this case, you should leave max-tx-retries=0. See the EJB section in the *Oracle Application Server Performance Guide* for more information.
memory-threshold		memoryThreshold		✔	This attribute defines a threshold for how much used JVM memory is allowed before passivation should occur. Specify an integer that is translated as a percentage. When reached, beans are passivated, even if their idle timeout has not expired. Default: 80%. To disable, specify any negative number.

Table A–1 (Cont.) Attributes for the <session-deployment> Element

Attribute	@StatelessDeployment Equivalent	@StatefulDeployment Equivalent	Stateless	Stateful	Description
min-instances	minInstances		✔		The number of minimum bean implementation instances to be kept instantiated or pooled. The default is 0. This setting is valid for stateless session beans only.
name			✔	✔	The name of the bean, which matches the name of a bean in the assembly section of the EJB deployment descriptor (ejb-jar.xml).
passivate-count		passivateCount		✔	This attribute is an integer that defines the number of beans to be passivated if any of the resource thresholds have been reached. Passivation of beans is performed using the least recently used algorithm. Default: one-third of the max-instances attribute. You can disable this attribute by setting the count to zero or a negative number.
persistence-filename		persistenceFileName		✔	Path to the file where sessions are stored across restarts.
pool-cache-timeout	poolCacheTimeout		✔		The pool-cache-timeout applies to stateless session enterprise beans. This parameter specifies how long to keep stateless sessions cached in the pool. For stateless session beans, if you specify a pool-cache-timeout, then at every pool-cache-timeout interval all beans of the corresponding bean type in the pool are removed. If the value specified is zero or negative, then the pool-cache-timeout is disabled and beans are not removed from the pool. The default value is 60 (seconds)
replication		replicationType		✔	Configuration of the state replication for stateful session beans. Values can be inherited (default) onShutdown, onRequestEnd, or none. See "State Replication" on page 2-30 for more information.
resource-check-interval		resourceCheckInterval		✔	The container checks all resources at this time interval. At this time, if any of the thresholds have been reached, passivation occurs. Default: 180 sec. (3 min.). To disable, specify any negative number.

Table A–1 (Cont.) Attributes for the <session-deployment> Element

Attribute	@StatelessDeployment Equivalent	@StatefulDeployment Equivalent	Stateless	Stateful	Description
timeout		timeout		✔	The maximum number of seconds that a stateful session bean may be inactive before being subject to pool clean-up. If the value is zero or negative, then all timeouts are disabled. Every 30 seconds the pool clean up logic is invoked. Within the pool clean up logic, only the sessions that timed out, by passing the timeout value, are deleted. Adjust the timeout based on your applications use of stateful session beans. For example, if stateful session beans are not removed explicitly by your application, and the application creates many stateful session beans, then you may want to lower the timeout value. If your application requires that a stateful session bean be available for longer than 1800 seconds (equal to 30 minutes), then adjust the timeout value accordingly. The default value is 1800 seconds.
transaction-timeout	transactionTimeout	transactionTimeout	✔	✔	The maximum number of seconds that OC4J will wait for a transaction started by this stateless or stateful session bean to commit or rollback. If the value is zero or negative, the timeout is disabled.
tx-retry-wait	transactionRetryWait	transactionRetryWait	✔	✔	This parameter specifies the time to wait in seconds between retrying the transaction. The default value is 60 seconds.
wrapper			✔	✔	Name of the OC4J wrapper class for this bean. This is an internal server value and should not be edited.

<ior-security-config>

The `<ior-security-config>` element is an interoperability element, which is discussed fully in the Interoperability chapter in the *Oracle Containers for J2EE Services Guide*.

<env-entry-mapping>

The `<env-entry-mapping>` element maps environment variables to JNDI names and is discussed in "Configuring an Environment Reference to an Environment Variable" on page 19-16.

<ejb-ref-mapping>

The `<ejb-ref-mapping>` element maps any EJB references to JNDI names and is discussed in "EJB Environment References" on page 19-2.

<resource-ref-mapping>

The `<resource-ref-mapping>` element maps any EJB references to JNDI names and is discussed in "Resource Manager Connection Factory Environment References" on page 19-2.

<resource-env-ref-mapping>

The <resource-env-ref-mapping> element is used to map an administered object for a resource. For example, to use JMS, the bean must obtain both a JMS factory object and a destination object. These objects are retrieved at the same time from JNDI. The <resource-ref> element declares the JMS factory and the <resource-env-ref> element is used to declare the destination. Thus, the <resource-env-ref-mapping> element maps the destination object. See "Resource Manager Connection Factory Environment References" on page 19-2 for more information.

<message-destination-ref-mapping>

The <message-destination-ref-mapping> element is only used if you are using JMS 1.1. Use this element to map the message-destination-ref-name in the client deployment descriptor to another location that is available in the OC4J environment. It provides means of linking message consumers and producers to one or more common logical destinations. For more information, see "Configuring an Environment Reference to a JMS Destination Resource Manager Connection Factory (JMS 1.1)" on page 19-13.

<entity-deployment>

The <entity-deployment> section provides additional deployment information for an EJB 2.x or EJB 1.1 entity bean deployed within this JAR file.

> **Note:** All <entity-deployment> attributes and sub-elements apply only to EJB 2.x or EJB 1.1 entity beans. They are not applicable to JPA entities.

The <entity-deployment> section contains the following structure:

```
<entity-deployment call-timeout=... clustering-schema=...
      copy-by-value=... data-source=... exclusive-write-access=...
      disable-default-persistent-unit=...
      do-select-before-insert=... isolation=...
      location=... local-location=... locking-mode=...
      max-instances=... min-instances=...
      max-tx-retries=... tx-retry-wait=... update-changed-fields-only=...
      name=... pool-cache-timeout=...
      table=... validity-timeout=... force-update=...
      wrapper=... local-wrapper=... delay-updates-until-commit=...
      findByPrimaryKey-lazy-loading=... >
  <ior-security-config>
    <transport-config>
      <integrity></integrity>
      <confidentiality></confidentiality>
      <establish-trust-in-target></establish-trust-in-target>
      <establish-trust-in-client></establish-trust-in-client>
    </transport-config>
    <as-context>
      <auth-method></auth-method>
      <realm></realm>
      <required></required>
    </as-context>
    <sas-context>
      <caller-propagation></caller-propagation>
```

```
    </sas-context>
  </ior-security-config>
  <primkey-mapping>
   <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
      persistence-type=...></cmp-field-mapping>
  </primkey-mapping>
  <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
      persistence-type=...> </cmp-field-mapping>
  <finder-method partial=... query=... lazy-loading=... prefetch-size=... >
   <method></method>
  </finder-method>
  <env-entry-mapping name=...></env-entry-mapping>
  <ejb-ref-mapping location=... name=... remote=... jndi-properties=... />
  <resource-ref-mapping location=... name=... >
    <lookup-context location=...>
       <context-attribute name=... value=... />
    </lookup-context>
  </resource-ref-mapping>
  <resource-env-ref-mapping location=... name=... />
</entity-deployment>
```

For information on each of these elements and sub-elements, see the following:

- <entity-deployment> Attributes
- <ior-security-config>
- <primkey-mapping>
- <cmp-field-mapping>
- <finder-method>
- <env-entry-mapping>
- <ejb-ref-mapping>
- <service-ref-mapping>
- <resource-ref-mapping>
- <resource-env-ref-mapping>
- <message-destination-ref-mapping>
- <commit-option>

Examples

For entity bean examples, which include `<entity-deployment>` configuration (where relevant), see "Implementing an EJB 2.1 Entity Bean" on page 13-1.

<entity-deployment> Attributes

Table A–2 lists the attributes for the `<entity-deployment>` element.

For more information about OC4J support for TopLink persistence, see "TopLink Persistence Support" on page A-2.

Table A–2 Attributes for the <entity-deployment> Element

Attribute	Configurable in orion-ejb-jar.xml	Configurable Using TopLink Persistence API	Description
`call-timeout`		✔	Using TopLink persistence API, you can specify the maximum time OC4J will wait for a query to return a result. A query timeout ensures that your application does not block forever over a hung or lengthy query that does not return in a timely fashion. You can specify a query timeout at the descriptor and query level. A descriptor-level query timeout applies to all queries on the descriptor's reference class. Specify a descriptor-level query timeout to apply the same timeout to all queries on a particular object type. A query-level query timeout applies to that query only. For more information, see the following: ■ "Configuring Query Timeout at the Descriptor Level" in the *Oracle TopLink Developer's Guide* "Configuring Named Query Advanced Options" in the *Oracle TopLink Developer's Guide* ■ "Configuring Query Timeout at the Query Level" in the *Oracle TopLink Developer's Guide*
`clustering-schema`	✔		Do not use. Not needed in this release.
`copy-by-value`	✔		Whether or not to copy (clone) all the incoming and outgoing parameters in EJB calls. Set to `false` if you are certain that your application does not assume copy-by-value semantics for a speed-up. The default value is `true`.
`data-source`	✔		The name of the data source used if using container-managed persistence.
`delay-updates-until-commit`		✔	Using TopLink persistence API, you can configure OC4J for either deferred or non-deferred changes. By default, TopLink defers all changes until commit time: this is the most efficient approach that produces the least number of data source interactions. Alternatively, you can configure an entity bean's descriptor for nondeferred changes. This means that as you change the persistent fields of the entity bean, OC4J modifies the relational schema immediately. For more information, see "Non-Deferred Changes" in the *Oracle TopLink Developer's Guide*.
`disable-default-persistent-unit`	✔		By default, OC4J enables the deployment of EJB 3.0 entities without a `persistence.xml` file if your application only uses the OC4J default persistence unit. To disable this feature, set to `true`. The default is `false`. For more information, see "Understanding OC4J Persistence Unit Defaults" on page 2-8.
`do-select-before-insert`			TopLink does not perform a select before writing out changes. Oracle recommends using optimistic locking to handle the possibility of concurrent overwrites. For more information, see "Concurrency (Locking) Mode" on page 1-60.
`exclusive-write-access`		✔	Using TopLink persistence API, OC4J assumes exclusive write access to the entity instances, because TopLink uses its unit of work transaction space to calculate change sets and write out the changes. The unit of work transaction space is separate from the shared session cache. For more information, see "Unit of Work Architecture" in the *Oracle TopLink Developer's Guide*.

Table A–2 (Cont.) Attributes for the <entity-deployment> Element

Attribute	Configurable in orion-ejb-jar.xml	Configurable Using TopLink Persistence API	Description
`findByPrimaryKey-lazy-loading`		✔	Using TopLink persistence API, you can configure fetch groups, which let you retrieve a subset of a bean's attributes. This is equivalent of lazy loading. For more information, see "Using Queries with Fetch Groups" in the *Oracle TopLink Developer's Guide*.
`force-update`		✔	Using TopLink persistence API, you can configure whether or not OC4J executes persistence-related life cycle methods, even if OC4J does not believe that any of the persistence data has changed. When set to `true`, this option means that OC4J will still execute the EJB life cycle by invoking the `ejbStore` method. This manages data in transient fields and sets appropriate persistent fields during the `ejbStore` method. For example, an image might be kept in one format in memory, but stored in a different format in the database. The default value is `false`. For more information, see "Configuring a Descriptor With EJB Information" in the *Oracle TopLink Developer's Guide*
`isolation`		✔	Using TopLink persistence API, database transaction isolation levels are not that relevant to TopLink, because it provides an object cache and unit of work transaction space. Consider configuring TopLink unit of work and cache isolation levels instead. Handling locking through database isolation levels is rarely done. Typically, locking is done through optimistic or pessimistic locking. You can configure transaction isolation level on a TopLink database login: this setting applies to all beans and transactions that use the database login. By default, TopLink uses whatever isolation level is set on the database. For more information, see the following: ■ "How do you Avoid Database Resource Contention?" on page 1-59 ■ "Concurrency (Locking) Mode" on page 1-60 ■ "Database Transaction Isolation Levels" in the *Oracle TopLink Developer's Guide* ■ *Oracle Application Server Performance Guide.*
`local-location`	✔		Defines the local JNDI name to which this bean will be bound
`local-wrapper`	✔		Name of the OC4J local home wrapper class for this bean. This is an internal server value and should not be edited.
`location`	✔		The JNDI-name to which this bean will be bound.

Table A–2 (Cont.) Attributes for the <entity-deployment> Element

Attribute	Configurable in orion-ejb-jar.xml	Configurable Using TopLink Persistence API	Description
`locking-mode`		✔	Using TopLink persistence API, you can configure the following locking modes: ▪ Optimistic Locking: Multiple users have read access to the data. When a user attempts to make a change, the application checks to ensure the data has not changed since the user read the data. TopLink supports version (recommended), timestamp, and field-level locking. ▪ Pessimistic Locking: The first user who accesses the data with the purpose of updating it locks the data until completing the update. This manages resource contention and does not allow parallel execution. Only one user at a time is allowed to execute the entity bean at a single time. ▪ Read-only: Multiple users can execute the entity bean in parallel. The container does not allow any updates to the bean's state. For more information, see "Concurrency (Locking) Mode" on page 1-60.
`max-instances`	✔		The maximum number of bean implementation instances to be kept instantiated or pooled. The default is `0`, which means infinite. To disable instance pooling, set `max-instances` to any negative number. This will create a new instance at the start of the EJB call and release it at the end of the call. See "Configuring Bean Instance Pool Size" on page 31-4 for more information.
`max-tx-retries`	✔		This parameter specifies the number of times to retry a transaction that was rolled back due to system-level failures. The default value is `0`. Generally, Oracle recommends that you add retries only where errors are seen that could be resolved through retries. For example, if you are using serializable isolation and you want to retry the transaction automatically if there is a conflict, you might want to use retries. However, if the bean wants to be notified when there is a conflict, then, in this case, you should leave `max-tx-retries=0`. See the EJB section in the *Oracle Application Server Performance Guide* for more information.
`min-instances`	✔		The minimum number of bean implementation instances to be kept instantiated or pooled. The default value is `0`. See "Configuring Bean Instance Pool Size" on page 31-4 for more information.
`name`	✔		The name of the bean, which matches the name of a bean in the assembly section of the EJB deployment descriptor (`ejb-jar.xml`).
`pool-cache-timeout`	✔		The amount of time in seconds that the bean implementation instances are to be kept in the "pooled" (unassigned) state. Specifying any negative number retains the instances until they are garbage collected. The default is 60. See "Configuring Bean Instance Pool Timeouts for Entity Beans" on page 31-7 for more information
`table`		✔	Using TopLink persistence API, you can specify the name of the database table associated with this bean. For more information, see "Configuring Associated Tables" in the *Oracle TopLink Developer's Guide*
`tx-retry-wait`	✔		This parameter specifies the time to wait in seconds between retrying the transaction. The default is 60 seconds.[1]

Table A–2 (Cont.) Attributes for the <entity-deployment> Element

Attribute	Configurable in orion-ejb-jar.xml	Configurable Using TopLink Persistence API	Description
`update-changed-fields-only`		✔	Using TopLink persistence API, the TopLink unit of work always calculates a change set and generates an update statement for changed fields only.
`validity-timeout`		✔	Using TopLink persistence API, you can configure an invalidation policy to For more information, see "Cache Invalidation" in the *Oracle TopLink Developer's Guide*.
`wrapper`	✔		Name of the OC4J remote home wrapper class for this bean. This is an internal server value and should not be edited.

[1] you specify this attribute, you cannot enable XML validation (see "Validating XML Files" on page 31-8).

<ior-security-config>

The `<ior-security-config>` element configures CSIv2 security policies for interoperability, which is discussed fully in the Interoperability chapter in the *Oracle Containers for J2EE Services Guide.*

<primkey-mapping>

The `<primkey-mapping>` element maps the primary key to the container-managed persistent field it represents. In this release, this feature is not configured in `orion-ejb-jar.xml` file. OC4J automatically makes this configuration. To manually configure this feature, you use TopLink persistence API.

For more information, see the following:

- "Customizing the JPA Persistence Provider" on page 3-3
- "Customizing the TopLink EJB 2.1 Persistence Manager" on page 3-13
- "Understanding Sequencing in Relational Projects" in the *Oracle TopLink Developer's Guide*

<cmp-field-mapping>

If you still use EJB 1.1 entity beans with container-managed persistence, use the `<cmp-field-mapping>` element to map the container-managed persistent fields to the database.

The following are the XML elements used for container-managed persistent data field mapping within the `orion-ejb-jar.xml` file for EJB 1.1 entity beans with container-managed persistence:

```
<cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
      persistence-type=...>
   <fields>
      <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
          persistence-type=...></cmp-field-mapping>
   </fields>
   <properties>
      <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
            persistence-type=...></cmp-field-mapping>
   </properties>
   <entity-ref home=...>
      <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
          persistence-type=...></cmp-field-mapping>
```

```
   </entity-ref>
   <collection-mapping table=...>
      <primkey-mapping>
         <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
             persistence-type=...></cmp-field-mapping>
      </primkey-mapping>
      <value-mapping immutable="true|false" type=...>
         <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
             persistence-type=...></cmp-field-mapping>
      </value-mapping>
   </collection-mapping>
   <set-mapping table=...>
      <primkey-mapping>
         <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
             persistence-type=...></cmp-field-mapping>
      </primkey-mapping>
      <value-mapping immutable="true|false" type=...>
         <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
             persistence-type=...></cmp-field-mapping>
      </value-mapping>
   </set-mapping>
</cmp-field-mapping>
```

For EJB 3.0 entities and EJB 2.1 entity beans, this feature is not configured in `orion-ejb-jar.xml` file. OC4J automatically makes this configuration. To manually configure this feature, you use TopLink persistence API.

For more information, see the following:

- "What are Container-Managed Persistent Fields?" on page 1-42
- "Customizing the JPA Persistence Provider" on page 3-3
- "Customizing the TopLink EJB 2.1 Persistence Manager" on page 3-13
- "Understanding Relational Mappings" in the *Oracle TopLink Developer's Guide*

<finder-method>

The `<finder-method>` element is used to create finder methods for EJB 1.1 entity beans.

For more information, see the following:

- "How do you Query for a JPA Entity?" on page 1-39
- "How do you Query for an EJB 2.1 Entity Bean?" on page 1-50

<env-entry-mapping>

The `<env-entry-mapping>` element maps environment variables to JNDI names and is discussed in "Configuring an Environment Reference to an Environment Variable" on page 19-16.

<ejb-ref-mapping>

The `<ejb-ref-mapping>` element maps any EJB references to JNDI names and is discussed in "EJB Environment References" on page 19-2.

<service-ref-mapping>

The <service-ref-mapping> element maps any EJB references to a Web service and is discussed in "Configuring an Environment Reference to a Web Service" on page 19-17

<resource-ref-mapping>

The <resource-ref-mapping> element maps any EJB references to JNDI names and is discussed in "Resource Manager Connection Factory Environment References" on page 19-2.

<resource-env-ref-mapping>

The <resource-env-ref-mapping> element is used to map an administered object for a resource. For example, to use JMS, the bean must obtain both a JMS factory object and a destination object. These objects are retrieved at the same time from JNDI. The <resource-ref> element declares the JMS factory and the <resource-env-ref> element is used to declare the destination. Thus, the <resource-env-ref-mapping> element maps the destination object. See "Resource Manager Connection Factory Environment References" on page 19-2 for more information.

<message-destination-ref-mapping>

The <message-destination-ref-mapping> element is only used if you are using JMS 1.1. Use this element to map the message-destination-ref-name in the client deployment descriptor to another location that is available in the OC4J environment. It provides means of linking message consumers and producers to one or more common logical destinations. For more information, see "Configuring an Environment Reference to a JMS Destination Resource Manager Connection Factory (JMS 1.1)" on page 19-13.

<commit-option>

The <commit-option> element determines an entity bean instance's state at transaction commit time and offers the flexibility to allow OC4J to optimize certain application conditions. This is discussed in "What are Entity Bean Commit Options?" on page 1-48.

<message-driven-deployment>

The <message-driven-deployment> section provides additional deployment information for a message driven bean deployed within this JAR file.

The <message-driven-deployment> section contains the following structure:

```
<message-driven-deployment cache-timeout=... connection-factory-location=...
      destination-location=... name=... subscription-name=...
      listener-threads=... transaction-timeout=...
      dequeue-retry-count=... dequeue-retry-interval=... interceptor-type=... >
  <env-entry-mapping name=...></env-entry-mapping>
  <ejb-ref-mapping location=... name=... remote=... jndi-properties=... />
  <resource-ref-mapping location=... name=... >
    <lookup-context location=...>
       <context-attribute name=... value=... />
    </lookup-context>
  </resource-ref-mapping>
```

```
  <resource-env-ref-mapping location=... name=... />
  <message-destination-ref-mapping location=... name=... />
  <config-property>
    <config-property-name> ... </config-property-name>
    <config-property-value> ... </config-property-value>
  </config-property>
</message-driven-deployment>
```

> **Note:** Alternatively, in an EJB 3.0 application, you can use the OC4J-proprietary annotation `@MessageDrivenDeployment`. You can use the `orion-ejb-jar.xml` file `<message-driven-deployment>` configuration to override `@MessageDrivenDeployment` configuration. For more information, see "Configuring OC4J-Proprietary Deployment Options on an EJB 3.0 MDB" on page 10-17.

For information on each of these elements and sub-elements, see: the following

- <message-driven-deployment> Attributes
- <env-entry-mapping>
- <ejb-ref-mapping>
- <resource-ref-mapping>
- <resource-env-ref-mapping>
- <message-destination-ref-mapping>
- <config-property>

Examples

A message-driven bean example, which includes the `<message-driven-deployment>` element, is described in the following:

- "Implementing an EJB 3.0 Message-Driven Bean" on page 9-1
- "Implementing an EJB 2.1 Message-Driven Bean" on page 17-1
- "Configuring OC4J-Proprietary Deployment Options on an EJB 3.0 Session Bean" on page 5-10

<message-driven-deployment> Attributes

Table A–3 lists the attributes of the `<message-driven-deployment>` element that you can use to configure message service options. This table also lists the following corresponding configuration alternatives:

- attributes of the `@MessageDrivenDeployment` annotation;
- activation configuration property names that you can use in the following annotations and elements:
  - `<config-property>` element owned by a `<message-driven-deployment>` element;
  - `@ActivationConfigProperty` annotation owned by a `@MessageDrivenDeployment` or `@MessageDriven` annotation.

Note: If you configure using attributes, your application can only access a message service provider without a J2CA resource adapter. If later you decide to access your message service provider using a J2CA resource adapter, your application will fail to deploy. If you configure using activation configuration properties, your application can access a message service provider with or without a J2CA resource adapter. Oracle recommends that you use `<config-property>` or `@ActivationConfigProperty` options.

For more information, see the following:

- "Message Service Configuration Options: Annotations or XML? Attributes or Activation Configuration Properties?" on page 2-26
- "What Message Service Providers Can you use With Your MDB?" on page 2-21

Table A–3 Attributes for the <message-driven-deployment> Element

<message-driven-deployment> Attribute	@MessageDriven Deployment Attribute	Description	Activation Configuration Property Name[1]
`cache-timeout`	poolCacheTimeout	This parameter specifies how long to keep message-driven beans cached in the pool. If you specify a pool `cache-timeout`, then at every cache timeout interval, all beans of the corresponding bean type in the pool are removed. If the value specified is zero or negative, then the cache timeout is disabled and beans are not removed from the pool. The default value is 60 (seconds).	N/A
`connection-factory-location`	N/A	The JNDI location of the connection factory to use. The JMS `Destination Connection Factory` is specified in this attribute. The syntax is `java:comp/resource` + resource provider name + `TopicConnectionFactories` OR `QueueConnectionFactories` + user defined name. The `nnnConnectionFactories` details what type of factory is being defined.	See `ConnectionFactoryJndiName` in Table B–1.
`dequeue-retry-count`	dequeueRetryCount	Specifies how often the listener thread tries to re-acquire the JMS session once database failover has ocurred. This is applicable to only container-managed transactions in an MDB. The default value is `0`. For more information, see the following: ■ "Configuring Connection Failure Recovery for an EJB 2.1 MDB" on page 18-9 ■ "Understanding OC4J EJB Application Clustering Services" on page 2-29	N/A See `EndpointFailureRetryInterval` in Table B–2.
`dequeue-retry-interval`	dequeueRetryInterval	Specifies the interval between retries. The default value is 60 seconds. For more information, see the following: ■ "Configuring Connection Failure Recovery for an EJB 2.1 MDB" on page 18-9 ■ "Understanding OC4J EJB Application Clustering Services" on page 2-29	N/A See `EndpointFailureRetryInterval` in Table B–2.

Table A–3 (Cont.) Attributes for the <message-driven-deployment> Element

<message-driven-deployment> Attribute	@MessageDriven Deployment Attribute	Description	Activation Configuration Property Name[1]
destination-location	destinationLocation	The JNDI location of the destination (queue/topic) to use. The JMS Destination is specified in the destination-location attribute. The syntax is java:comp/resource + resource provider name + Topics OR Queues + Destination name. The Topic or Queue details what type of Destination is being defined. The Destination name is the actual queue or topic name defined in the database.	See DestinationName in Table B–1.
interceptor-type	interceptorType	The attribute indicates how OC4J handles interceptor class life cycle. You can set the interceptor type to bean (stateful class) or singleton (stateless class). When set to bean, OC4J creates a separate interceptor class instance for each session bean instance that you associate with that interceptor class. This is in accordance with the EJB 3.0 specification. In this case, the interceptor class must be stateful. When set to singleton, OC4J creates a single interceptor class instance that all session bean instances share. In this case, the interceptor class must be stateless. For more information, see "Singleton Interceptors" on page 2-12.	N/A
listener-threads	listenerThreads	The listener threads are used to concurrently consume JMS messages. The default is one thread. Topics can only have one thread. Queues can have more than one. For more information, see "Configuring Parallel Message Processing" on page 18-7.	See ReceiverThreads in Table B–2.
max-delivery-count	maxDeliveryCount	The maximum number of times OC4J will attempt the immediate redelivery of a message to a message-driven bean's onMessage method if that method returns failure (fails to invoke an acknowledgment operation, throws an exception, or both). After this number of redeliveries, the message is deemed undeliverable and is handled according to the policies of your message service provider. For example, OEMS JMS will put the message on its exception queue (jms/Oc4jJmsExceptionQueue). For more information, see "Configuring Maximum Delivery Count" on page 18-8.	See MaxDeliveryCnt in Table B–2.
max-instances	maxInstances	The maximum number of bean implementation instances to be kept instantiated or pooled. The default is 0, which means infinite. To disable instance pooling, set max-instances to any negative number. This will create a new instance at the start of the EJB call and release it at the end of the call. For message-driven beans, the default pooling setting is typically appropriate. Change this value only if MDB life cycle methods are very expensive and you need fine-grained control over how often instances are created and managed in the pool. See "Configuring Bean Instance Pool Size" on page 31-4 for more information.	N/A
min-instances	minInstances	The minimum number of bean implementation instances to be kept instantiated or pooled. The default value is 0. See "Configuring Bean Instance Pool Size" on page 31-4 for more information.	N/A

Table A–3 (Cont.) Attributes for the <message-driven-deployment> Element

<message-driven-deployment> Attribute	@MessageDriven Deployment Attribute	Description	Activation Configuration Property Name[1]
`name`	`name`	The name of the bean, which matches the name of a bean in the assembly section of the EJB deployment descriptor (`ejb-jar.xml`).	N/A
`resource-adapter`	resourceAdapter	The name of the resource adapter instance that this MDB uses. Applicable only if this MDB is using a J2CA message service provider. In order for the MDB to be activated by messages received by the resource adapter, the MDB and resource adapter must be connected. For more information, see "Configuring a J2CA Resource Adapter for use With Your Message Service Provider" on page 23-1.	N/A
`subscription-name`	subscriptionName	The name of the topic to which this message-drive bean subscribes.	See `SubscriptionName` in Table B–2.
`transaction-timeout`	transactionTimeout	This attribute controls the transaction timeout interval (in seconds) for any container-managed transactional MDB. The default is one day or 86,400 seconds. If the transaction has not completed in this time frame, the transaction is rolled back. This applies to both normal JMS and J2CA resource adapter-based message providers. For more information, see "Configuring a Transaction Timeout for a Message-Driven Bean" on page 21-7	See `TransactionTimeout` in Table B–2.

[1] For use in a `<message-driven-deployment>` element `<config-property>` subelement, or in an `@ActivationConfigProperty` annotation owned by a `@MessageDrivenDeployment` or `@MessageDriven` annotation.

<env-entry-mapping>

The `<env-entry-mapping>` element maps environment variables to JNDI names and is discussed in "Configuring an Environment Reference to an Environment Variable" on page 19-16.

<ejb-ref-mapping>

The `<ejb-ref-mapping>` element maps any EJB references to JNDI names and is discussed in "EJB Environment References" on page 19-2.

<resource-ref-mapping>

The `<resource-ref-mapping>` element maps any resource manager references to JNDI names and is discussed in "Resource Manager Connection Factory Environment References" on page 19-2.

<resource-env-ref-mapping>

The `<resource-env-ref-mapping>` element is used to map an administered object for a resource. For example, to use JMS, the bean must obtain both a JMS factory object and a destination object. These objects are retrieved at the same time from JNDI. The `<resource-ref>` element declares the JMS factory and the `<resource-env-ref>` element is used to declare the destination. Thus, the `<resource-env-ref-mapping>` element maps the destination object. See "Configuring an Environment Reference to a JMS Destination or Connection Resource Manager Connection Factory (JMS 1.0)" on page 19-14 for more information.

<message-destination-ref-mapping>

The <message-destination-ref-mapping> element is only used if you are using JMS 1.1. Use this element to map the message-destination-ref-name in the client deployment descriptor to another location that is available in the OC4J environment. It provides means of linking message consumers and producers to one or more common logical destinations. For more information, see "Configuring an Environment Reference to a JMS Destination Resource Manager Connection Factory (JMS 1.1)" on page 19-13.

<config-property>

The <config-property> element is only used if you are using a J2CA message service provider. Use this element to set J2CA resource adapter configuration properties. When OC4J deploys an MDB configured to use a J2CA message service provider, OC4J provides the MDB's activation specification to the resource adapter. This specification includes the properties you set in the <config-property> element.

Alternatively, for an EJB 3.0 message-driven bean, you can set J2CA resource adapter configuration properties using @MessageDriven attribute configProperty and @ActivationConfig annotation.

You can use the orion-ejb-jar.xml file <config-property> configuration to override @MessageDriven configuration.

For more information, see the following:

- "Configuring an EJB 3.0 MDB to Access a Message Service Provider Using J2CA" on page 10-1
- "Configuring an EJB 2.1 MDB to Access a Message Service Provider Using J2CA" on page 18-1

<assembly-descriptor>

In addition to specifying deployment information for individual beans, you can also specify addition deployment mapping information for security in the <assembly-descriptor> section. The <assembly-descriptor> section contains the following structure:

```
<assembly-descriptor>
  <security-role-mapping impliesAll=... name=...>
   <group name=... />
   <user name=... />
  </security-role-mapping>
  <message-destination-mapping location=... name=...>
  </message-destination-mapping>
  <default-method-access>
   <security-role-mapping impliesAll=... name=...>
    <group name=... />
    <user name=... />
   </security-role-mapping>
  </default-method-access>
</assembly-descriptor>
```

For information on each of these elements and subelements, see: the following

- <security-role-mapping>
- <message-destination-mapping>

- <default-method-access>
- <method>

Examples

For examples of `<assembly-descriptor>` element configuration, see the following:

- "Specifying Logical Roles in the EJB Deployment Descriptor" on page 22-3
- "Implementing an EJB 2.1 MDB" on page 17-1

<security-role-mapping>

The `<security-role-mapping>` element is described in "Mapping Logical Roles to Users and Groups" on page 22-8.

<message-destination-mapping>

<default-method-access>

The `<default-method-access>` element is described in "Specifying a Default Role Mapping for Undefined Methods" on page 22-9.

<method>

The `<method>` element is used to specify the methods (and possibly their parameters) of an enterprise bean:

```
<method>
    <description></description>
    <ejb-name></ejb-name>
    <method-intf></method-intf>
    <method-name></method-name>
    <method-params>
        <method-param></method-param>
    </method-params>
</method>
```

You can configure a `<method>` element using any of the following styles:

- When referring to all the methods of the specified enterprise bean's home and remote interfaces, specify the methods as follows:

```
<method>
    <ejb-name>EJBNAME</ejb-name>
    <method-name>*</method-name>
</method>
```

- When referring to multiple methods with the same overloaded name, specify the methods as follows:

```
<method>
    <ejb-name>EJBNAME</ejb-name>
    <method-name>METHOD</method-name>
</method>
```

- When referring to a single method within a set of methods with an overloaded name, you can specify each parameter within the method as follows:

```
<method>
  <ejb-name>EJBNAME</ejb-name>
  <method-name>METHOD</method-name>
  <method-params>
    <method-param>PARAM-1</method-param>
    <method-param>PARAM-2</method-param>
    ...
    <method-param>PARAM-n</method-param>
  </method-params>
</method>
```

B

J2CA Activation Configuration Properties

This appendix describes the J2EE Connector Architecture (J2CA) activation configuration properties that you can use to specify message service options when you access a JMS message service provider using a J2CA connector such as the Oracle JMS Connector.

Table B–1 lists the mandatory J2CA activation configuration properties you must set.

Table B–2 lists the optional J2CA activation configuration properties you may set.

For a complete list of all activation configuration properties, download and unzip one of the `how-to-gjra-with-<RESOURCE-PROVIDER-NAME>.zip` files from `http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`, where `<RESOURCE-PROVIDER-NAME>` is the name of the relevant resource provider. The `orion-ejb-jar.xml` demo file contains comments describing all activation configuration properties.

For more information, see the following:

- "Message Service Configuration Options: Annotations or XML? Attributes or Activation Configuration Properties?" on page 2-26
- "Configuring an EJB 3.0 MDB to Access a Message Service Provider Using J2CA" on page 10-1
- "Configuring an EJB 2.1 MDB to Access a Message Service Provider Using J2CA" on page 18-1
- "JMS Resource Adapter" in the *Oracle Containers for J2EE Services Guide*

Table B–1 Mandatory J2CA @ActivationConfigProperty Attributes

Property Name	Value
`ConnectionFactoryJndiName`	The JNDI name of the message service provider connection factory. You define this name when you configure your message service provider.
`DestinationName`	The JNDI name of the message service provider destination name. You define this name when you configure your message service provider
`DestinationType`	The fully qualified `String` class name of the destination type for your message service provider. For a JMS MDB, either `javax.jms.Queue` or `javax.jms.Topic`.

Table B–2 Optional J2CA @ActivationConfigProperty Attributes

Property Name	Value
`AcknowledgeMode`	How listener threads, which consume messages and call a message-driven bean's message listener method (for example, the `onMessage` method for a JMS message listener), acknowledge the delivery of a message. The following are valid values: ▪ `Auto-acknowledge` (default): the listener thread sends an acknowledgment as soon as a message is received by the MDB. ▪ `Dups-ok-acknowledge`: the listener thread sends an acknowledgment lazily; this can improve performance. It is possible for the MDB to receive duplicates of the message until the acknowledgment is actually sent. If you select this option, your MDB must be able to handle duplicate messages.
`ClientId`	The `String` name that a listener thread will set on connections it acquires on behalf of its message-driven bean. The default is no name.
`EndpointFailureRetryInterval`	The number of milliseconds that OC4J will wait before attempting to start a new listener thread and reconnect with the JMS provider after any failure that causes the number of listener threads to drop to zero. Listener threads are automatically terminated on JMS provider failure: for example, if a listener thread calling `MessageConsumer` method `receive` results in an exception. If all listener threads terminate while the endpoint is still running (for example, due to a sustained network failure or if the JMS server is not up), the MDB cannot receive messages. In this situation, the Oracle JMS Connector attempts to start a new listener thread and reconnect with the JMS provider every `EndpointFailureRetryInterval` milliseconds until it is either successful or the endpoint is shut down. Once a listener thread is successfully created, normal listener thread management resumes (see `ReceiverThreads`, `ListenerThreadMaxIdleDuration`, and `ListenerThreadMinBusyDuration`). Consider the following consequences of connection failure recovery: ▪ message ordering: since recovery from a connection failure requires the creation of a new JMS session, and JMS message ordering guarantees only apply to messages received within a single session, messages received after the reconnect may not be ordered with respect to messages received before the reconnect; ▪ lost messages: if the endpoint is a nondurable subscriber, messages may be lost or duplicated. This problem will not happen when using queues or when using topics with durable subscribers. Whether or not you experience these problems may be subject to the specific behavior of your JMS provider The default is 60000 milliseconds.
`ExceptionQueueName`	The JNDI name of the of the `javax.jms.Queue` object to use as the exception queue. This property is required when `UseExceptionQueue` is `true`, and ignored when `false`.
`IncludeBodiesInExceptionQueue`	Determines whether or not messages sent to the exception queue will include a message body. The following are valid values: ▪ `true` (default): OC4J includes the message body in messages sent to the exception queue; ▪ `false`: OC4J does not include the message body in messages sent to the exception queue. If many messages are sent to the exception queue during normal operation and the message body is of no use in the exception queue, then this property may be set false to improve performance. This property does not apply: ▪ If `UseExceptionQueue` is `false`. ▪ If the original message did not have a message body, then the message sent to the exception queue will not have one either. ▪ If a copy of the original message cannot be created for any reason, then the original may be sent to the exception queue instead. This may result in a message body being sent to the exception queue.
`ListenerThreadMaxIdleDuration`	The number of milliseconds that OC4J will keep a listener thread that is not receiving any messages. At least one listener thread will remain as long as the endpoint is active. The default is 300000 milliseconds.

Table B–2 (Cont.) Optional J2CA @ActivationConfigProperty Attributes

Property Name	Value
`ListenerThreadMaxPollInterval`	The upper limit (in milliseconds) on the polling interval of the Oracle JMS Connector adaptive polling interval algorithm. The Oracle JMS Connector uses an adaptive algorithm to determine the actual polling interval: during periods of activity, it uses shorter polling intervals (higher polling rates) and during periods of inactivity, it uses longer polling intervals (lower polling rates) that will not exceed this property. Listener threads poll to see if there is a message waiting to be processed. The more frequently this polling is performed, the faster (on average) a given listener thread can respond to a new message. The price for frequent polling is overhead–the resource provider must process a receive request each time it is polled. The default is 5000 milliseconds.
`ListenerThreadMinBusyDuration`	If a listener thread has just received a message, has not been idle (had to wait for a new message to arrive) at any point during the past `ListenerThreadMinBusyDuration` milliseconds, and the current number of listener threads for this endpoint is less than `ReceiverThreads`, then OC4J will create an additional listener thread if possible. The default is 10000 milliseconds.
`LoggerName`	The logger name for Oracle JMS Connector log messages. The `LoggerName` property is required when setting the `LogLevel` property.
`LogLevel`	Determines the level of detail of Oracle JMS Connector log messages. Although primarily intended for debugging the Oracle JMS Connector itself, these messages may also be useful when debugging issues related to its use. Oracle recommends that you set this property temporarily for debugging purposes; this property should not be set in production code. Note that specific log messages and log levels may be added, removed, or modified in future versions of the Oracle JMS Connector. The `LoggerName` property is required when setting the `LogLevel` property. The following are valid values: ■ `ConnectionPool`: log connection pool related messages only. ■ `ConnectionOps`: log connection related messages only. ■ `TransactionalOps`: log transaction related messages only. ■ `ListenerThreads`: log listener thread related messages only. ■ `INFO`: logs the login/logout for each server session, including the user name. After acquiring the session, detailed information is logged. ■ `CONFIG`: logs only login, JDBC connection, and database information. ■ `FINE`: logs some internal information. ■ `FINER`: similar to warning. Includes stack trace. ■ `FINEST`: includes additional low level information. ■ `SEVERE`: logs exceptions indicating TopLink cannot continue, as well as any exceptions generated during login. This includes a stack trace. ■ `WARNING`: logs exceptions that do not force TopLink to stop, including all exceptions not logged with severe level. This does not include a stack trace. ■ `OFF`: disables logging.
`MaxDeliveryCnt`	The maximum number of times a listener thread will attempt to deliver a message to a message-driven bean. A value of 0 means never discard a message. If a message has the `JMSXDeliveryCount` property, whose value is greater than `MaxDeliveryCnt`, then the message will be discarded: that is, the listener thread will not call the message-driven bean's message listener method (for example, the `onMessage` method for a JMS message listener). If the exception queue is enabled (see `UseExceptionQueue`), a copy of the message will be sent to the exception queue. Use a value of `0` with caution. If `MaxDeliveryCnt` is set to `0` to prevent a message from ever being discarded, and a message-driven bean always responds to a given message by throwing an exception, then the message-driven bean may endlessly fail to process the same message as it is redelivered over and over again. The default value is `5`.

Table B–2 (Cont.) Optional J2CA @ActivationConfigProperty Attributes

Property Name	Value
`MessageSelector`	The `String` selector expression of message properties that match the type of message your MDB should receive. Messages that do not match the expression are filtered (not delivered to the MDB). This is used as the `messageSelector` for the JMS sessions created for the listener threads. The default is no filtering.
`ReceiverThreads`	The maximum number of listener threads to create for this endpoint. For queues, using more than one thread may help increase the rate at which messages can be consumed. For topics this value must always be `1`. Each listener thread gets its own session and topic subscriber. For durable subscribers, it would be an error to have more than one subscriber with the same subscription name. For nondurable, subscribers having more than one thread will not help because more threads translates into more subscribers which translates into more copies of each message. See also `ListenerThreadMinBusyDuration`. The default is 1.
`ResPassword`	The String password that OC4J passes to the resource provider. The `ResPassword` property supports standard password indirection options (for example, you can use `->joeuser` to represent the password of `joeuser`). When set, OC4J passes this value to the `create*Connection` method as the password argument. When only one of `ResUser` or `ResPassword` is set, OC4J passes null for the unset property. When neither `ResUser` nor `ResPassword` are set, connections used for this MDB's inbound message handling and exception queue handling (see `UseExceptionQueue`) are created using the no-argument version of the `create*Connection` method. The default is `null`.
`ResUser`	The `String` user name that OC4J passes to the resource provider. When set, OC4J passes this value to the `create*Connection` method as the `user` argument. When only one of `ResUser` or `ResPassword` is set, OC4J passes null for the unset property. When neither `ResUser` nor `ResPassword` are set, connections used for this MDB's inbound message handling and exception queue handling (see `UseExceptionQueue`) are created using the no-argument version of the `create*Connection` method. The default is `null`.
`SubscriptionDurability`	Determines the durability of the topic consumer used by a listener thread. This is applicable only for topics (do not set this property for queues). The following are valid values: ▪ `Durable`: messages are not missed even if the OC4J is not running. Reliable applications will typically make use of durable topic subscriptions rather than non-durable topic subscriptions. When this property is set to `Durable` (and `DestinationType` is `javax.jms.Topic` or `javax.jms.Destination`), the `SubscriptionName` property is required. ▪ `NonDurable` (default): OC4J ensures that a message-drive bean is available to service a message as long as OC4J is running. Messages may be missed if OC4J is not running for any period of time.

Table B–2 (Cont.) Optional J2CA @ActivationConfigProperty Attributes

Property Name	Value
`SubscriptionName`	The `String` name that a listener thread uses when it creates a durable subscriber. In a given JMS server, you should assign a given subscription name to at most one MDB (which must have at most one listener thread). This property is required when `SubscriptionDurability` is `Durable` (and `DestinationType` is `javax.jms.Topic` or `javax.jms.Destination`). In all other cases, this property is ignored.
`TransactionTimeout`	The upper limit (in milliseconds) that the Oracle JMS Connector will wait for a message to arrive before exiting the current transaction. The OC4J transaction manager limits the amount of time a transaction can last (see `transaction-timeout` in `transaction-manager.xml`). Set this property so that the transaction manager will not timeout the transaction during a call to a message-driven bean's message listener method (for example, the `onMessage` method for a JMS message listener) unless something is wrong. For example, If the transaction manager timeout is set to 30 seconds, and the `onMessage` routine will never take more than 10 seconds unless something is wrong, then you could set `TransactionTimeout` to 20 seconds (20000 milliseconds). The default is 300000 milliseconds.

Table B–2 (Cont.) Optional J2CA @ActivationConfigProperty Attributes

Property Name	Value
`UseExceptionQueue`	Determines how OC4J handles messages that it cannot deliver because the message-driven bean's message listener method (for example, the `onMessage` method for a JMS message listener) throws an exception or the listener thread exceeds the `MaxDeliveryCnt` threshold. The following are valid values: ▪ `true`: OC4J sends undeliverable messages to the exception queue as described later. In this case, the `ExceptionQueueName` property is required. ▪ `false` (default): OC4J discards undeliverable messages. The default is `false`. When `UseExceptionQueue` is set to `true`, OC4J sends undeliverable messages to the exception queue as follows: 1. Create a new message of the same type. 2. Copy the properties and body from the original message to the new message. 3. Translate headers in the original to properties in the copy, assigning each header obtained through `getJMS{Header}` to property `GJRA_CopyOfJMS{Header}`. Since `javax.jms.Destination` is not a valid property type, translate destination headers into descriptive messages. This service is not provided for `JMSX*` properties, most notably the `JMSXDeliveryCount` property. OC4J translates headers this way because if the headers were copied, sending the message to the exception queue would cause most of them to be lost (over-written by the resource-provider). 4. Add a string property called `GJRA_DeliveryFailureReason` which indicates why the message was not delivered. 5. If the message-driven bean's message listener method generated an exception immediately prior to the delivery failure, add a string property called `GJRA_onMessageExceptions` which contains exception information. 6. Validate the new message: If the copy and augmentation process is successful, add a `boolean` property called `GJRA_CopySuccessful` with the value `true`. If some part of the copy or augmentation process fails, OC4J does not stop. It attempts to complete the rest of the procedure. For `Bytes`, `Map`, and `Stream` message types, this can mean that part of the body is copied and the rest is not. In this case, add a `boolean` property called `GJRA_CopySuccessful` with the value `false`. 7. Use the connection factory specified by the `ConnectionFactoryJndiName` property to send the resulting message to the exception queue. Only one attempt is made to send the message to the exception queue. Should this attempt fail, the message will be discarded without being placed in the exception queue. Because OC4J uses the connection factory specified by the `ConnectionFactoryJndiName` property to send the message to the exception queue (in addition to being used for the primary destination), if the primary destination (specified by the `DestinationName` property) is a topic, then the connection factory must support both queues and topics (that is, the connection factory must be either `javax.jms.ConnectionFactory` or `javax.jms.XAConnectionFactory`). For potential variations of the previous procedure, see `IncludeBodiesInExceptionQueue`.

Glossary

This glossary defines terms frequently used in this guide. For additional Java EE terminology, see http://java.sun.com/javaee/reference/glossary/index.jsp.

Annotation

A simple, expressive means of decorating Java source code with metadata that is compiled into the corresponding Java class files for interpretation at run time by a **JPA persistence provider** to manage JPA behavior. Annotations allow you to declaratively define how to map Java objects to relational database tables in a standard, portable way that works both inside a Java EE 5 application server and outside an EJB container in a Java Standard Edition (Java SE) 5 application. JPA annotations are specified in the `javax.persistence` package.

Entity

A Java object whose nontransient fields should be persisted to a relational database using the services of a **JPA entity manager** obtained from a JPA **persistence provider** (either within a Java EE EJB container or outside of an EJB container in a Java SE application). Using JPA, you can designate any **POJO** as an entity using the `@Entity` **annotation**.

Entity Manager

The interface that you use to access a **persistence context**. You use an entity manager to create, read, update, and delete **entity** instances.

J2CA

J2EE Connector Architecture: the standard way to integrate **JMS** providers with J2EE application servers by wrapping a JMS provider client library in a resource adapter.

JMS

Java Message Service.

JPA

The EJB 3.0 Java Persistence API.

OEMS

Oracle Enterprise Messaging Service: a suite of **JMS** providers that OC4J supports, including: OEMS JMS Connector (a **J2CA**-based provider), OEMS JMS (an in-memory or file-based provider), and OEMS JMS Database (**Oracle AQ**-based provider).

Oracle AQ

AQ is a unique database-integrated message queuing feature, built on the Oracle Streams information integration infrastructure. It allows diverse applications to communicate asynchronously through messages. Integration with the database provides unique message management functionality, such as auditing, tracking, and message persistence for security, scheduling, and message metadata analysis.

You can access AQ through PL/SQL, Java (using the `oracle.AQ` package), Java Message Service (JMS), or over the Internet using transport protocols such as HTTP, HTTPS, and SMTP. For Internet access, the client - a user or Internet application - and the Oracle server exchange structured XML messages.

AQ also provides transformations that are useful for enterprise application integration and a messaging gateway to automatically propagate messages to and from OracleAQ queues.

For more information, see `http://otn.oracle.com/products/aq/index.html`.

Persistence Context

The set of **entity** instances in which for any persistent entity identity there is a unique entity instance. A persistence context is associated with an **entity manager** instance. It is within this persistence context that the entity manager manages the entity instances and their life cycle.

Persistence Provider

An EJB 3.0 Java Persistence API implementation of

Persistence Unit

The set of entities that an **entity manager** instance manages. A persistence unit defines the set of all classes that are related by your application and which must be mapped to a single database.

POJI

Plain Old Java Interface: an interface that you define; one that need not extend an interface that Java EE specifies. In EJB 3.0, a business interface may be a POJI.

POJO

Plain Old Java Object: a Java class that you define; one that need not extend a class or implement an interface that Java EE specifies. In EJB 3.0, an **entity** may be a POJO.

RA

Resource Adapter: specifically, one that complies with **J2CA**.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X