

Oracle® Containers for J2EE

Enterprise JavaBeans Developer's Guide,
10g Release 3 (10.1.3)

B14428-01

January 2006

Primary Author: Peter Purich

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

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

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

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

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

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

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

Contents

Preface	xvii
Audience	xvii
Documentation Accessibility	xvii
Related Documents	xviii
Conventions	xviii

Part I EJB Overview

1 Understanding Enterprise JavaBeans

What are Enterprise JavaBeans?	1-1
What is the Anatomy of an EJB 3.0 EJB?	1-2
What is the Anatomy of an EJB 2.1 EJB?	1-4
What is the Lifecycle of an EJB?	1-5
Callback Methods	1-6
What is EJB Context?	1-6
How Do Annotations and Resource Injection Work?	1-7
What is a Session Bean?	1-8
What is a Stateless Session Bean?	1-8
What is the Stateless Session Bean Lifecycle?	1-9
What is a Stateful Session Bean?	1-10
What is the Stateful Session Bean Lifecycle?	1-10
What is Session Context?	1-13
What is an EJB 3.0 Entity?	1-14
What are Container-Managed Persistence Fields?	1-14
What are Container-Managed Relationship Fields?	1-14
What is the EJB 3.0 Entity Lifecycle?	1-15
What is an EJB 3.0 Entity Primary Key?	1-16
How Do You Query for an EJB 3.0 Entity?	1-16
Understanding EJB Query Syntax	1-16
Understanding the EJB 3.0 EntityManager Query API	1-18
What is an EJB 2.1 Entity Bean?	1-18
What is an EJB 2.1 CMP Entity Bean?	1-19
What are Container-Managed Persistence Fields?	1-20
What are Container-Managed Relationship Fields?	1-20
What is the CMP Entity Bean Lifecycle?	1-20

What is a CMP Entity Bean Primary Key?	1-21
What is an EJB 2.1 BMP Entity Bean?	1-22
What are Bean-Managed Persistence Fields?	1-22
What are Bean-Managed Relationship Fields?	1-22
What is the BMP Entity Bean Lifecycle?	1-22
What is a BMP Entity Bean Primary Key?	1-23
What is Entity Context?	1-24
How do You Avoid Database Resource Contention?	1-24
Transaction Isolation	1-24
Concurrency (Locking) Mode	1-25
When Does Entity Bean Passivation Occur?	1-25
What are Entity Bean Commit Options?	1-26
Commit Options and CMP Applications	1-26
Commit Options and BMP Applications	1-27
How Do You Query for an EJB 2.1 Entity Bean?	1-27
Understanding EJB Query Syntax	1-27
Understanding Finder Methods	1-30
Understanding Select Methods	1-32
What is a Message-Driven Bean?	1-33
What is the Message-Driven Bean Lifecycle?	1-34
What is Message Driven Context?	1-35
Which Type of EJB Should You Use?	1-35
Which Type of Session Bean Should You Use?	1-35
When do you use Bean-Managed versus Container-Managed Persistence?	1-35
What is the Difference Between Session and Entity Beans?	1-36

2 Understanding EJB Application Development

How Should You Develop EJB Applications?	2-1
Understanding the EJB Application Directory Structure	2-1
Using EJB Development Tools	2-2
Using JDeveloper	2-2
Using Eclipse	2-3
Using TopLink Workbench	2-3
What OC4J Services Can You Use with an EJB?	2-3
How do You Package and Deploy an EJB Application?	2-4
General Packaging and Deployment Procedure	2-4
Understanding EJB Deployment Descriptor Files	2-7
What is the ejb-jar.xml File?	2-7
What is the orion-ejb-jar.xml File?	2-8
What is the toplink-ejb-jar.xml File?	2-8
What is the ejb3-toplink-sessions.xml File?	2-9
Understanding Packaging	2-9
Understanding Deployment	2-9
How Do Specify Vendor-Specific Configuration in an EJB 3.0 Application?	2-10
In What Order does OC4J Deploy EJB Modules?	2-10
How Do You Use an EJB in Your Application?	2-11
Understanding Client Access	2-11

Understanding EJB 3.0 Interceptors	2-11
Interceptor Restrictions	2-11
Interceptors and Invocation Context.....	2-12
Understanding EJB Administration	2-12
Understanding EJB JNDI Services	2-13
Understanding EJB Data Source Services	2-13
What Types of Data Source does OC4J Support?	2-13
Where Do You Configure Data Source Information in OC4J?	2-14
What is a Default Data Source?	2-14
How Does OC4J Handle Multiple Data Sources?	2-14
Understanding EJB Transaction Services	2-14
Who Manages a Transaction?.....	2-15
Container-Managed Transaction (CMT)	2-15
Bean-Managed Transaction (BMT).....	2-15
Who Begins and Commits a Transaction?	2-15
How do I Participate in a Global or Two-Phase Commit Transaction?	2-16
Understanding EJB Security Services	2-16
Understanding Message Services	2-16
What Message Providers Can I use with My MDB?	2-16
Oracle Application Server JMS (OracleAS JMS) Provider: File-Based	2-17
Oracle JMS (OJMS) Provider: Advanced Queueing (AQ)-Based.....	2-17
J2EE Connector Architecture (J2CA) Adapter Message Provider	2-18
Understanding OC4J EJB Application Clustering Services	2-20
State Replication	2-20
State Replication Trigger.....	2-21
State Replication Scope	2-21
State Replication Mode	2-21
Load Balancing	2-22
Replication-Based Load Balancing	2-22
Static Retrieval Load Balancing	2-22
DNS Load Balancing	2-22
Failover	2-22
Transactions	2-22
Performance	2-22
Understanding EJB Timer Services	2-23
Understanding J2EE Timer Services.....	2-23
Understanding OC4J Cron Timer Services.....	2-23

3 Understanding EJB Support in OC4J

EJB 3.0 Support	3-1
What JDK is Required?	3-1
How do You Define an EJB 3.0 Application?	3-2
How does OC4J Manage Persistence in an EJB 3.0 Application?	3-2
TopLink Entity Manager.....	3-2
Customizing the TopLink Entity Manager	3-2
EJB 2.1 Support	3-4
What JDK is Required?	3-4

How do You Define an EJB 2.1 Application?	3-4
How does OC4J Manage Persistence in an EJB 2.1 Application?	3-4
TopLink Persistence Manager	3-4
Customizing the TopLink Persistence Manager	3-5
Migrating to the TopLink Persistence Manager	3-5
Configuration Changes in this Release	3-5
New Package Names for RMI and Application Client Initial Context Factories	3-5
Unsupported orion-ejb-jar.xml Attributes	3-6

Part II EJB 3.0 Session Beans

4 Implementing an EJB 3.0 Session Bean

Implementing an EJB 3.0 Stateless Session Bean	4-1
Implementing an EJB 3.0 Stateful Session Bean	4-2

5 Using EJB 3.0 Session Bean API

Configuring Passivation	5-1
Configuring Passivation Criteria	5-2
Configuring Passivation Location.....	5-2
Configuring a Lifecycle Callback Method for an EJB 3.0 Session Bean.....	5-2
Using Annotations	5-2
Configuring an Interceptor on an EJB 3.0 Session Bean.....	5-3
Using Annotations	5-3

Part III EJB 3.0 Entities

6 Implementing an EJB 3.0 Entity

Implementing an EJB 3.0 Entity	6-1
--------------------------------------	-----

7 Using EJB 3.0 Persistence API

Configuring an EJB 3.0 Entity Primary Key	7-2
Configuring an EJB 3.0 Entity Primary Key Field	7-2
Using Annotations	7-2
Configuring EJB 3.0 Entity Automatic Primary Key Generation.....	7-3
Using Annotations	7-3
Configuring Table and Column Information	7-5
Configuring the Primary Table	7-5
Using Annotations	7-5
Configuring a Secondary Table.....	7-5
Using Annotations	7-6
Configuring a Column	7-6
Using Annotations	7-6
Configuring a Join Column	7-7
Using Annotations	7-7
Configuring an EJB 3.0 Entity Container-Managed Relationship Field	7-8

Configuring a Basic Mapping	7-8
Using Annotations	7-8
Configuring a Large Object Mapping	7-9
Using Annotations	7-9
Configuring a Serialized Object Mapping	7-9
Using Annotations	7-9
Configuring a One-to-One Mapping	7-10
Using Annotations	7-10
Configuring a Many-to-One Mapping	7-10
Using Annotations	7-11
Configuring a One-to-Many Mapping	7-11
Using Annotations	7-11
Configuring a Many-to-Many Mapping	7-11
Using Annotations	7-12
Configuring an Aggregate Mapping	7-12
Using Annotations	7-12
Configuring Optimistic Lock Version Field	7-14
Using Annotations	7-14
Configuring Lazy Loading on Finder Methods	7-14
Configuring a Lifecycle Callback Method for an EJB 3.0 Entity	7-14
Using Annotations	7-15
Configuring Inheritance for an EJB 3.0 Entity	7-16
Joined Subclass	7-16
Single Table per Class Hierarchy	7-16
Using Annotations	7-17
Configuring Joined Subclass Inheritance with Annotations	7-17
Configuring Single Table Inheritance with Annotations	7-18

8 Using EJB 3.0 Query API

Implementing an EJB 3.0 Named Query	8-1
Using Annotations	8-1
Implementing an EJB 3.0 Dynamic Query	8-2
Using Java.....	8-3

Part IV EJB 3.0 Message-Driven Beans

9 Implementing an EJB 3.0 MDB

Implementing an EJB 3.0 MDB	9-1
--	-----

10 Using EJB 3.0 MDB API

Configuring an EJB 3.0 MDB to Use a Non-J2CA Message Service Provider	10-2
Using Annotations	10-2
Using Deployment XML	10-3
Configuring an EJB 3.0 MDB to Use a J2CA Message Service Provider	10-3
Using Annotations	10-3
Using Deployment XML	10-5

Configuring Listener Threads	10-5
Configuring Maximum Delivery Count	10-5
Configuring Dequeue Retry Count and Interval	10-5
Configuring an Interceptor on an EJB 3.0 MDB Message Listener Method	10-5
Using Annotations	10-6
Configuring a Lifecycle Callback Method for an EJB 3.0 MDB	10-6
Using Annotations	10-6

Part V EJB 2.1 Session Beans

11 Implementing an EJB 2.1 Session Bean

Implementing an EJB 2.1 Stateless Session Bean	11-1
Using Java	11-2
Using Deployment XML	11-3
Implementing an EJB 2.1 Stateful Session Bean	11-4
Using Java	11-5
Using Deployment XML	11-6
Implementing the Home Interfaces	11-7
Implementing the Remote Home Interface	11-7
Implementing the Local Home Interface	11-8
Implementing the Component Interfaces	11-9
Implementing the Remote Component Interface	11-9
Implementing the Local Component Interface	11-9
Implementing the setSessionContext Method	11-10

12 Using EJB 2.1 Session Bean API

Configuring Passivation	12-1
Using Deployment XML	12-2
Configuring Passivation Criteria	12-2
Using Deployment XML	12-2
Configuring Passivation Location	12-3
Using Deployment XML	12-3
Configuring a Lifecycle Callback Method for an EJB 2.1 Session Bean	12-3
Using Java	12-4

Part VI EJB 2.1 Entity Beans

13 Implementing an EJB 2.1 Entity Bean

Implementing an EJB 2.1 CMP Entity Bean	13-1
Using Java	13-3
Using Deployment XML	13-5
Implementing an EJB 2.1 BMP Entity Bean	13-6
Using Java	13-8
Using Deployment XML	13-14
Implementing an EJB 2.1 BMP ejbCreate Method	13-15
Implementing the EJB 2.1 Home Interfaces	13-18

Implementing the Remote Home Interface	13-18
Implementing the Local Home Interface	13-19
Implementing the EJB 2.1 Component Interfaces	13-19
Implementing the Remote Component Interface	13-19
Implementing the Local Component Interface	13-20
Implementing the setEntityContext and unsetEntityContext Methods	13-20
14 Using EJB 2.1 CMP Entity Bean API	
Configuring an EJB 2.1 CMP Entity Bean Primary Key	14-1
Configuring an EJB 2.1 CMP Entity Bean Primary Key Field	14-1
Using Deployment XML	14-2
Configuring an EJB 2.1 CMP Entity Bean Composite Primary Key Class	14-3
Using Java	14-3
Using Deployment XML	14-4
Configuring EJB 2.1 CMP Entity Bean Automatic Primary Key Generation	14-4
Using Deployment XML	14-5
Configuring Automatic Database Table Creation	14-5
Using Deployment XML	14-5
Configuring an EJB 2.1 CMP Entity Bean Container-Managed Persistence Field	14-6
Using Java	14-7
Using Deployment XML	14-7
Configuring an EJB 2.1 CMP Entity Bean Container-Managed Relationship Field	14-8
Using Java	14-8
Using Deployment XML	14-9
Configuring Default Mappings	14-9
Using Deployment XML	14-9
Configuring Exclusive Write Access to the Database	14-10
Using Deployment XML	14-11
Configuring Lazy Loading on Finder Methods	14-11
Using Deployment XML	14-11
15 Using EJB 2.1 BMP Entity Bean API	
Configuring a Read-Only BMP Entity Bean	15-1
Using Deployment XML	15-2
Configuring BMP Commit Options	15-2
Using Deployment XML	15-2
Configuring an EJB 2.1 BMP Entity Bean Query	15-3
Implementing an EJB 2.1 BMP the ejbFindByPrimaryKey Method	15-3
Implementing Other EJB 2.1 BMP Finder Methods	15-3
Configuring a Lifecycle Callback Method for an EJB 2.1 BMP Entity Bean	15-4
Implementing an EJB 2.1 BMP ejbStore Method	15-4
Implementing an EJB 2.1 BMP ejbLoad Method	15-5
Implementing an EJB 2.1 BMP ejbPassivate Method	15-5
Implementing an EJB 2.1 BMP ejbActivate Method	15-6
Implementing an EJB 2.1 BMP ejbRemove Method	15-6

16 Using EJB 2.1 Query API

Implementing an EJB 2.1 EJB QL Finder Method	16-1
Using Java.....	16-2
Using Deployment XML	16-3
Using TopLink Workbench	16-4
Implementing an EJB 2.1 EJB QL Select Method	16-5
Using Java.....	16-5
Using Deployment XML	16-7
Using TopLink Workbench	16-7
OC4J EJB 2.1 EJB QL Extensions	16-8

Part VII EJB 2.1 Message-Driven Beans

17 Implementing an EJB 2.1 MDB

Implementing an EJB 2.1 MDB	17-1
Using Java.....	17-2
Using Deployment XML	17-4
Implementing the setMessageDrivenContext Method.....	17-6

18 Using EJB 2.1 MDB API

Configuring an EJB 2.1 MDB to Use a Non-J2CA Message Service Provider	18-1
Using Deployment XML	18-1
Configuring an EJB 2.1 MDB to Use a J2CA Message Service Provider	18-2
Using Deployment XML	18-3
Configuring an MDB for Fast Undeploy on Windows	18-4
Using System Properties	18-4
Configuring an MDB for Oracle RAC Failover	18-5
Using Deployment XML	18-5
Using Java.....	18-5
Configuring Listener Threads	18-6
Using Deployment XML	18-6
Configuring Maximum Delivery Count	18-7
Using Deployment XML	18-7
Configuring Dequeue Retry Count and Interval	18-8
Using Deployment XML	18-8

Part VIII OC4J EJB Services

19 Configuring JNDI Services

Configuring Environment References	19-1
EJB Environment References	19-2
Resource Manager Connection Factory Environment References.....	19-2
Environment Variable Environment References	19-2
Web Service Environment References	19-2
Where Do You Configure an EJB Environment Reference?	19-3

Should You Use Logical Names?	19-3
Configuring an Environment Reference to an EJB	19-3
Configuring an Environment Reference to a Remote EJB.....	19-3
Configuring an Environment Reference to a Local EJB.....	19-5
Configuring an Environment Reference to a JDBC Data Source Resource Manager Connection Factory	19-6
Configuring an Environment Reference to a JMS Destination or Connection Resource Manager Connection Factory	19-7
Configuring an Environment Reference to a Java Mail Resource Manager Connection Factory	19-10
Configuring an Environment Reference to a URL Resource Manager Connection Factory	19-12
Configuring an Environment Reference to an Environment Variable	19-13
Configuring an Environment Reference to a Web Service	19-14
Configuring the Initial Context Factory	19-15
Configuring the Default Initial Context Factory	19-16
Configuring an Oracle Initial Context Factory	19-16
Configuring the Naming Provider URL for OC4J and Oracle Application Server.....	19-17
Configuring the Naming Provider URL for OC4J Standalone.....	19-18
Setting JNDI Properties in an EJB	19-18
Setting JNDI Properties with the JNDI Properties File.....	19-18
Setting JNDI Properties with System Properties	19-19
Setting JNDI Properties in the Initial Context.....	19-19
Looking up an EJB 3.0 EJB	19-19
Using Annotations	19-19
Using Initial Context.....	19-20
Looking Up the Remote Interface of an EJB 3.0 EJB Using ejb-ref.....	19-20
Looking Up the Remote Interface of an EJB 3.0 EJB Using location.....	19-20
Looking up the Local Interface of an EJB 3.0 EJB Using local-ref	19-21
Looking up the Local Interface of an EJB 3.0 EJB Using local-location.....	19-21
Looking Up an EJB 3.0 Resource Manager Connection Factory	19-22
Using Annotations	19-22
Using Initial Context.....	19-22
Looking Up an EJB 3.0 Environment Variable	19-23
Using Resource Injection.....	19-23
Using Initial Context.....	19-24
Looking Up an EJB 2.1 EJB	19-24
Using Initial Context.....	19-24
Looking Up the Remote Interface of an EJB 2.1 EJB Using ejb-ref.....	19-24
Looking Up the Remote Interface of an EJB 2.1 EJB Using location.....	19-25
Looking up the Local Interface of an EJB 2.1 EJB Using local-ref	19-25
Looking up the Local Interface of an EJB 2.1 EJB Using local-location.....	19-26
Looking Up an EJB 2.1 Resource Manager Connection Factory	19-26
Using Initial Context.....	19-27
Looking Up an EJB 2.1 Environment Variable	19-27
Using Initial Context.....	19-27

20 Configuring Data Sources

Configuring a Data Source for an Oracle Database	20-1
Using Application Server Control Console	20-1
Using Deployment XML	20-2
Configuring a Data Source for a Third-Party Database.....	20-2
Using Application Server Control Console	20-2
Using Deployment XML	20-2
Configuring a Default Data Source for an EJB 3.0 Application.....	20-3
Using Deployment XML	20-3
Configuring a Default Data Source for an EJB 2.1 Application.....	20-3
Using Deployment XML	20-4

21 Configuring Transaction Services

Configuring Transaction Timeouts	21-1
Configuring a Global Transaction Timeout	21-1
Using Application Server Control Console.....	21-1
Using Deployment XML	21-2
Configuring a Transaction Timeout for a Session Bean	21-2
Using Deployment XML	21-2
Configuring a Transaction Timeout for a Message-Driven Bean	21-2
Using Deployment XML	21-3
Transaction Best Practices.....	21-3
Using Container Managed Transactions with Datasource Connections	21-3
Using a Rollback Strategy	21-5

22 Configuring Security Services

Granting Permissions in Browser	22-1
Authenticating and Authorizing EJB Applications.....	22-1
Specifying Users and Groups.....	22-2
Specifying Logical Roles in the EJB Deployment Descriptor	22-3
Specifying a Role for an EJB Method.....	22-4
Using Annotations	22-4
Using Deployment XML	22-5
Specifying Unchecked Security for EJB Methods.....	22-6
Using Annotations	22-6
Using Deployment XML	22-6
Specifying the runAs Security Identity	22-7
Using Annotations	22-7
Using Deployment XML	22-7
Mapping Logical Roles to Users and Groups	22-8
Specifying a Default Role Mapping for Undefined Methods.....	22-9
Specifying Users and Groups by the Client	22-9
Specifying Credentials in EJB Clients	22-10
Specifying Credentials in JNDI Properties	22-10
Specifying Credentials in the Initial Context	22-11
Retrieving Credentials from an EJB Using the JAAS API	22-12

Configuring EJB 3.0 Security Options	22-12
Using Annotations	22-13
23 Configuring Message Services	
Configuring an OracleAS JMS Message Service Provider	23-1
OracleAS JMS Destination and Connection Factory Names	23-2
Configuring jms.xml	23-2
Configuring an OJMS Message Service Provider	23-3
OJMS Destination and Connection Factory Names	23-3
Installing and Configuring the OJMS Provider	23-4
Configuring data-sources.xml	23-5
Transactional Functionality	23-5
Identify the JNDI Name of the Oracle AQ JMS Data Source.....	23-6
Configuring a Message Service Provider Using J2CA	23-7
J2CA Message Service Provider Connection Factory Names.....	23-8
Installing and Configuring a J2CA Adapter	23-8
Configuring OC4J Deployment XML Files	23-8
24 Configuring OC4J EJB Application Clustering Services	
Configuring EJB 3.0 and EJB 2.1 Stateful Session Bean Replication Policy	24-1
Using Deployment XML	24-1
Configuring Global Replication Policy in the application.xml File for Web and EJB Components	24-2
Configuring Application-Level Replication Policy in the orion-application.xml File for Web and EJB Components	24-2
Overriding Application-Level Replication Policy in the orion-ejb-jar.xml File for EJB Components	24-2
Configuring Replication-Based Load Balancing	24-3
Using System Properties	24-3
Configuring Static Retrieval Load Balancing	24-3
Using JNDI Properties.....	24-3
Configuring DNS Load Balancing	24-4
Using JNDI Properties.....	24-4
25 Configuring Timer Services	
Configuring an EJB 3.0 EJB with a J2EE Timer	25-1
Using Annotations	25-1
Using Initial Context.....	25-2
Configuring an EJB 2.1 EJB with a J2EE Timer	25-2
Configuring an EJB with an OC4J Cron Timer	25-3
Troubleshooting Timers	25-4
How to Retrieve Information About the Timer	25-5
How to Retrieve a Persisted Timer.....	25-5
Executing the Timer Within the Scope of a Transaction	25-5
What Does a NoSuchObjectLocalException Mean with Timers?	25-5

Part IX Packaging and Deploying an EJB Application

26 Configuring Deployment Descriptor Files

Configuring the ejb-jar.xml File.....	26-1
Creating ejb-jar.xml During Migration	26-1
Creating the ejb-jar.xml File at Deployment Time	26-1
Creating ejb-jar.xml with JDeveloper.....	26-1
Configuring the topink-ejb-jar.xml File.....	26-2
Creating topink-ejb-jar.xml During Migration	26-2
Creating topink-ejb-jar.xml with TopLink Workbench.....	26-2
Configuring the orion-ejb-jar.xml File.....	26-2
Configuring the ejb3-toplink-sessions.xml File.....	26-3
Creating ejb3-toplink-sessions.xml with TopLink Workbench.....	26-3

27 Packaging an EJB Application

Packaging an Application with Both EJB 3.0 and EJB 2.1 EJBs	27-1
Sharing Classes Between EJB Applications.....	27-1
Handling Out of Memory Exceptions at Runtime	27-2
Handling Class Cast Exceptions at Runtime.....	27-2

28 Deploying an EJB Application to OC4J

Deploying a Large EJB Application.....	28-1
Tuning the VM to Avoid Out Of Memory Errors During Deployment	28-1
Disabling Batch Compilation to Avoid Out Of Memory Errors During Deployment	28-2
Deploying Incrementally.....	28-2
Troubleshooting Application Deployment.....	28-3

Part X Using an EJB in Your Application

29 Accessing an EJB from a Client

What Type of Client Do You Have?	29-1
EJB Client.....	29-2
Stand-alone Java Client	29-2
Servlet or JSP Client	29-2
Configuring the Client.....	29-2
Configuring the Client Classpath for OC4J	29-3
Selecting an Initial Context Factory Class	29-3
Specifying Security Credentials	29-3
Selecting an EJB Reference.....	29-3
Accessing an EJB 3.0 EJB.....	29-4
Accessing an EJB 3.0 EJB in Another Application	29-5
Accessing an EJB 3.0 Entity Using an EntityManager.....	29-5
Acquiring an EntityManager.....	29-5
Using Annotations in an EJB 3.0 EJB Client	29-6
Using the InitialContext in an EJB Client	29-6

Using the InitialContext in a Web Client.....	29-6
Creating a New CMP Entity Instance	29-7
Querying for an EJB 3.0 Entity Using the EntityManager.....	29-8
Finding an Entity by Primary Key with the Entity Manager	29-8
Creating a Named Query with the EntityManager	29-9
Creating a Dynamic EJB QL Query with the Entity Manager.....	29-9
Creating a Dynamic TopLink Expression Query with the EntityManager.....	29-9
Creating a Dynamic Native SQL Query with the EntityManager.....	29-10
Configuring Query Hints.....	29-10
Executing a Query.....	29-11
Modifying a CMP Entity Instance	29-12
Using an Updating Query	29-12
Using the Entity's Public API.....	29-12
Refreshing from the Database	29-12
Removing an Entity	29-13
Using Flush	29-13
Detaching and Merging a CMP Entity Bean Instance	29-13
Accessing an EJB 3.0 EJBContext	29-14
Using Resource Injection.....	29-14
Accessing an EJB 2.1 EJB	29-14
Accessing an EJB 2.1 EJB Remotely	29-14
Accessing an EJB 2.1 EJB Locally	29-16
Accessing an EJB 2.1 EJB Using RMI from a Stand-Alone Java Client.....	29-16
Accessing an EJB 2.1 EJB in Another Application	29-17
Accessing an EJB 2.1 MDB	29-17
Sending a Message to a JMS Destination Using EJB 2.1	29-17
Sending a Message to a J2CA Destination Using EJB 2.1	29-21
Accessing an EJB 2.1 EJBContext	29-21
Handling Parameters	29-21
Passing Parameters Into an EJB	29-21
Handling Parameters Returned by an EJB	29-22
Handling Exceptions	29-22
Recovering From a NamingException While Accessing a Remote EJB.....	29-22
Recovering From a NullPointerException While Accessing a Remote EJB.....	29-23
Recovering From Deadlock Conditions.....	29-23
 30 Using a Stateless Session Bean with a Web Service	
Calling Out to a Web Service	30-1
Using Initial Context.....	30-1
Exposing a Stateless Session Bean to a Web Service	30-2
 31 Administrating an EJB Application	
Using Oracle Enterprise Manager 10g Application Server Control	31-1
Configuring EJB Logging	31-1
Logging Namespaces.....	31-2
Logging Levels	31-2

Configuring Logging with Application Server Control Logging MBean.....	31-2
Configuring Logging Using the j2ee-logging.xml File	31-2
Configuring Logging Using System Properties.....	31-2
Managing the Bean Instance Pool.....	31-3
Configuring Bean Instance Pool Size	31-3
Using Deployment XML	31-3
Configuring Bean Instance Pool Timeouts for Session Beans	31-4
Using Deployment XML	31-4
Configuring Bean Instance Pool Timeouts for Entity Beans.....	31-4
Using Deployment XML	31-4
Starting and Stopping an EJB Application.....	31-5
Troubleshooting an EJB Application	31-5
Validating XML Files	31-5
Debugging the ejb-jar.xml File	31-5
Debugging Generated Code	31-6
Preserving Generated Code in the Default Directory	31-6
Preserving Generated Code in a Directory You Specify	31-6
Disabling Generated Code Preservation	31-6

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

OC4J and the orion-ejb-jar.xml File.....	A-1
TopLink Persistence Manager Support	A-2
OC4J-Specific Deployment Descriptor for EJBs	A-4
Enterprise Beans Section	A-4
Persistence Manager Section (persistence-manager).....	A-5
Session Bean Section (session-deployment).....	A-5
Entity Bean Section (entity-deployment).....	A-9
Message Driven Bean Section (message-driven-deployment).....	A-14
EJB 1.1 CMP Field Mapping Section (cmp-field-mapping)	A-17
Method Definition.....	A-17
Assembly Descriptor Section.....	A-18
Element Description	A-19

Glossary

Index

Preface

This guide gets you started building EJB 3.0 and 2.1 Enterprise JavaBeans for Oracle Containers for J2EE (OC4J) using the TopLink 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 persistence manager for new development. Using the migration tool (see "[Migrating to the TopLink Persistence Manager](#)" on page 3-5), 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 J2EE and the EJB 3.0 and EJB 2.1 specifications.

Documentation Accessibility

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

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

Accessibility of Code Examples in Documentation

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

Accessibility of Links to External Web Sites in Documentation

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

TTY Access to Oracle Support Services

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

Related Documents

For more information, see the following documents in the Oracle Containers for J2EE 10g Release 3 (10.1.3) documentation set:

- *Oracle Application Server Release Notes*
- *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 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.
<i>italic</i>	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.

Convention	Meaning
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:

- [Chapter 1, "Understanding Enterprise JavaBeans"](#)
- [Chapter 2, "Understanding EJB Application Development"](#)
- [Chapter 3, "Understanding EJB Support in OC4J"](#)

Understanding Enterprise JavaBeans

The Java 2 Enterprise Edition (J2EE) 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 Enterprise JavaBeans 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?](#)
- [What is a Session Bean?](#)
- [What is an EJB 3.0 Entity?](#)
- [What is an EJB 2.1 Entity Bean?](#)
- [What is a Message-Driven Bean?](#)
- [Which Type of EJB Should You Use?](#)

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 EJB 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 1-8
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 1-8
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 1-10
Entity	An EJB 3.0 compliant light-weight entity object that represents persistent data stored in a relational database using container-managed persistence. Because it is not a remotely accessible component, an entity can represent a fine-grained persistent object.	"What is an EJB 3.0 Entity?" on page 1-14
Entity Bean	An EJB 2.1 EJB component that represents persistent data stored in a relational database.	"What is an EJB 2.1 Entity Bean?" on page 1-18
CMP	A Container-Managed Persistence (CMP) entity bean is an entity bean that delegates persistence management to the container that hosts it.	"What is an EJB 2.1 CMP Entity Bean?" on page 1-19
BMP	A Bean-Managed Persistence (BMP) entity bean is an entity bean that manages its own persistence.	"What is an EJB 2.1 BMP Entity Bean?" on page 1-22
MDB	A Message-Driven Bean (MDB) is an EJB 3.0 or EJB 2.1 EJB component that functions as an asynchronous consumer of Java Message Service (JMS) messages.	"What is a Message-Driven Bean?" on page 1-33

For more information, see:

- [What is the Anatomy of an EJB 3.0 EJB?](#)
- [What is the Anatomy of an EJB 2.1 EJB?](#)
- [What is the Lifecycle of an EJB?](#)
- [What is EJB Context?](#)
- [How Do Annotations and Resource Injection Work?](#)
- [Which Type of EJB Should You Use?](#)

What is the Anatomy of an EJB 3.0 EJB?

Using EJB 3.0, the interfaces for your EJB implementation are not restricted by EJB type. For example, in your EJB 3.0 entity bean 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 like `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 an EJB 3.0 Entity?"](#) on page 1-16). 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 EJB.

Table 1-2 *Parts of an EJB 3.0 EJB*

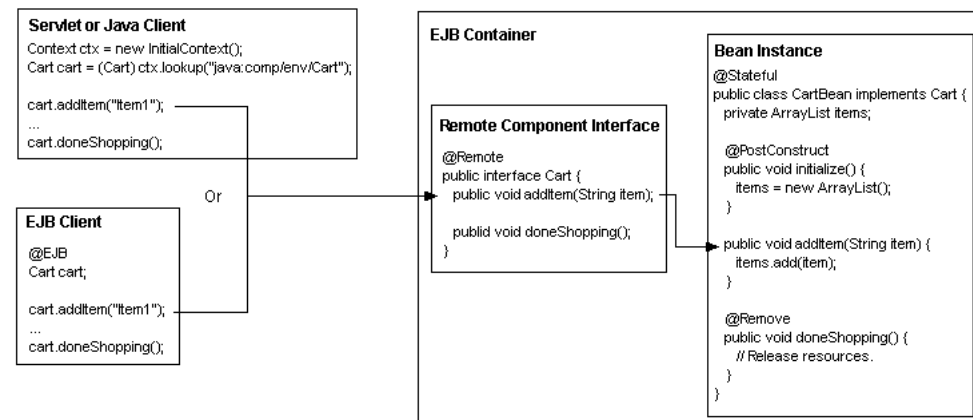
Part	Type	Description
Home interface	POJI	An optional POJI annotated with <code>@Home</code> that specifies an object that the container itself implements: the <i>home object</i> . The <code>@Home</code> is only provided to help EJB 3.0 beans interoperate with EJB 2.1 clients, if necessary. Most EJB 3.0 bean instances will not need to provide a home interface.
Component interface	POJI	An mandatory POJI annotated with <code>@Remote</code> or <code>@Local</code> (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 lifecycle callback function.

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

Part	Type	Description
Deployment descriptor	ejb-jar.xml orion-ejb-jar.xml toplink-ejb-jar.xml ejb3-toplink-sessions.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 2-7.

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 an EJB 3.0 entity (see ["How Do You Query for an EJB 3.0 Entity?"](#) on page 1-16).

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. Entity beans do not require a `remove` method; you use the EJB 3.0 `EntityManager` to create and destroy entity beans.

What is the Anatomy of an EJB 2.1 EJB?

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 EJB instance only with a `create` method that your EJB home interface provides. For more information, see ["EJB 2.1 Support"](#) on page 3-4.

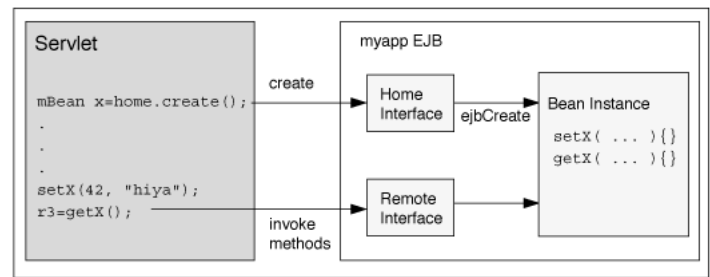
[Table 1-3](#) lists the parts you create when developing an EJB 2.1 EJB.

Table 1-3 Parts of an EJB 2.1 EJB

Part	Type	Description
Home interface	<code>javax.ejb.EJBHome</code> (remote) <code>javax.ejb.EJBLocalHome</code>	Specifies the interface to an object that the container itself implements: the <i>home object</i> . The home interface contains the life cycle methods, such as the <code>create</code> methods that specify how a bean is created.
Component interface	<code>javax.ejb.EJBObject</code> (remote) <code>javax.ejb.EJBLocalObject</code>	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	<code>javax.ejb.SessionBean</code> <code>javax.ejb.EntityBean</code> <code>javax.ejb.MessageDrivenBean</code>	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	<code>ejb-jar.xml</code> <code>toplink-ejb-jar.xml</code> <code>orion-ejb-jar.xml</code>	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 EJB 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—normally 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 Lifecycle of an EJB?

The lifecycle of an EJB involves important events such as creation, passivation, activation, and removal. Each such event is associated with a callback defined on the EJB class (see ["Callback Methods"](#) on page 1-6). The container invokes the callback prior to or immediately after the lifecycle event (depending on the event type).

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

For an EJB 3.0 bean, when the container is responsible for the lifecycle 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 bean, even when the container is responsible for the lifecycle callback, and even if you do not want to perform additional logic, you must at least provide an empty implementation of the lifecycle methods to satisfy the requirements of the applicable EJB interface.

For more information, see:

- ["What is the Stateless Session Bean Lifecycle?"](#) on page 1-9
- ["What is the Stateful Session Bean Lifecycle?"](#) on page 1-10
- ["What is the CMP Entity Bean Lifecycle?"](#) on page 1-20
- ["What is the BMP Entity Bean Lifecycle?"](#) on page 1-22
- ["What is the Message-Driven Bean Lifecycle?"](#) on page 1-34

Callback Methods

A lifecycle callback method is a method of an EJB class that you define to handle a lifecycle event (see ["What is the Lifecycle of an EJB?"](#) on page 1-5). The container invokes the callback associated with a given lifecycle event immediately prior to or immediately after the lifecycle event (depending on the event type).

You implement a lifecycle callback method to change the default behaviour of an EJB.

For an EJB 3.0 bean, you can annotate any EJB class method as a lifecycle method.

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

What is EJB Context?

The `EJBContext` interface provides an instance with access to the container-provided runtime context of an EJB 2.1 EJB 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
<code>getEnvironment</code>	Get the values of properties for the bean.
<code>getUserTransaction</code>	Get a transaction context, which allows you to demarcate transactions programmatically when using bean managed transactions (BMT). This is valid only for beans that have been designated transactional.
<code>setRollbackOnly</code>	Set the current transaction so that it cannot be committed. Applicable only to container-managed transactions.
<code>getRollbackOnly</code>	Check whether the current transaction is marked for rollback only. Applicable only to container-managed transactions.
<code>getEJBHome</code>	Retrieve the object reference to the corresponding <code>EJBHome</code> (home interface) of the bean.
<code>lookup</code>	Use JNDI to retrieve the bean by environment reference name. When using this method, you do <i>not</i> prefix the bean reference with <code>"java:comp/env"</code> .

Do not confuse `EJBContext` with `InitialContext` (see ["Configuring the Initial Context Factory"](#) on page 19-15).

For more information, see:

- ["What is Session Context?"](#) on page 1-13
- ["What is Entity Context?"](#) on page 1-24
- ["What is Message Driven Context?"](#) on page 1-35
- ["Accessing an EJB 2.1 EJBContext"](#) on page 29-21

How Do Annotations and Resource Injection Work?

Using the `@Resource` or `@EJB` annotation, an EJB 3.0 bean may use dependency injection mechanisms to acquire references to resources or other objects in its environment. You use `@Resource` to inject non-EJB resources and `@EJB` to inject EJBs.

If an EJB 3.0 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.

Note: In this release, OC4J does not support resource injection in the web container. For more information, see ["Servlet or JSP Client"](#) on page 29-2).

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 normally 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.

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>
```

What is a Session Bean?

A session bean is an EJB 3.0 or EJB 2.1 EJB 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 crash occur. Session bean objects are either stateless (see ["What is a Stateless Session Bean?"](#) on page 1-8) or stateful: maintaining conversational state across method calls and transactions (see ["What is a Stateful Session Bean?"](#) on page 1-10). 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 1-11). However, the session bean object itself must manage its own persistent data.

From a client's perspective, a session bean is a non-persistent 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 methods like `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 11-7) and component (["Implementing the Component Interfaces"](#) on page 11-9) interfaces. The client of a session bean may be a local client, a remote client, or a web service client, 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 1-13) that you use to make callback requests to the container.

This section describes:

- [What is a Stateless Session Bean?](#)
- [What is a Stateful Session Bean?](#)
- [What is Session Context?](#)

For more information, see:

- ["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.

For more information, see:

["Implementing an EJB 3.0 Stateless Session Bean"](#) on page 4-1

["Implementing an EJB 2.1 Stateless Session Bean"](#)

What is the Stateless Session Bean Lifecycle?

The lifecycle for EJB 3.0 (see [Table 1-5](#)) and EJB 2.1 (see [Table 1-6](#)) stateless session beans are identical. The difference is in how you register lifecycle callback methods.

[Table 1-5](#) lists the optional EJB 3.0 stateless session bean lifecycle callback methods you can define using annotations. For EJB 3.0 stateless session beans, you do not need to implement these methods.

Table 1-5 Lifecycle 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 1-6](#) lists the EJB 2.1 lifecycle 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 1-6 Lifecycle Methods for an EJB 2.1 Stateless Session Bean

EJB Method	Description
<code>ejbCreate</code>	The container invokes this method right before it creates the bean. Use this method to initialize non-client specific information such as retrieving a data source.
<code>ejbActivate</code>	This method is never called for a stateless session bean. Provide an empty implementation only.
<code>ejbPassivate</code>	This method is never called for a stateless session bean. Provide an empty implementation only.
<code>ejbRemove</code>	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.
<code>setSessionContext</code>	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 11-10.

For more information, see:

- ["Configuring a Lifecycle Callback Method for an EJB 3.0 Session Bean"](#) on page 5-2

- ["Configuring a Lifecycle 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 (to free up resources), the container maintains its 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 crashes. 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 recreated 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

- ["Implementing an EJB 3.0 Stateful Session Bean"](#) on page 4-2
- ["Implementing an EJB 2.1 Stateful Session Bean"](#) on page 11-4

What is the Stateful Session Bean Lifecycle?

The lifecycle for EJB 3.0 (see [Table 1-7](#)) and EJB 2.1 (see [Table 1-8](#)) stateful session beans are identical. The difference is in how you register lifecycle callback methods.

[Table 1-7](#) lists the optional EJB 3.0 stateful session bean lifecycle callback methods you can define using annotations. For EJB 3.0 stateful session beans, you do not need to implement these methods.

Table 1–7 Lifecycle 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: <ul style="list-style-type: none"> ■ "When does Stateful Session Bean Passivation Occur?" on page 1-11 ■ "What Object Types Can Be Passivated?" on page 1-12 ■ "Where is a Passivated Stateful Session Bean Stored?" on page 1-13
@PostActivate	The container invokes this method right after it reactivates a formerly passivated stateful session bean.

[Table 1–8](#) lists the EJB 2.1 lifecycle 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 Lifecycle Methods for an EJB 2.1 Stateful Session Bean

EJB Method	Description
<code>ejbCreate</code>	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.
<code>ejbActivate</code>	The container invokes this method right after it reactivates the bean.
<code>ejbPassivate</code>	The container invokes this method right before it passivates the bean. For more information, see: <ul style="list-style-type: none"> ■ "When does Stateful Session Bean Passivation Occur?" on page 1-11 ■ "What Object Types Can Be Passivated?" on page 1-12 ■ "Where is a Passivated Stateful Session Bean Stored?" on page 1-13
<code>ejbRemove</code>	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.
<code>setSessionContext</code>	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 11-10.

For more information, see:

- ["Configuring a Lifecycle Callback Method for an EJB 3.0 Session Bean"](#) on page 5-2
- ["Configuring a Lifecycle 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 1-12).

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

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

For information on configuring this criteria, 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 de-serializing the bean from secondary storage, and brought back into memory. Before activation, the container invokes the `ejbActivate` method so that the bean developer 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 1-12. If you do not prepare your stateful session beans for passivation by releasing all resources and only allowing 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 (note that you do not necessarily 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, an EJB home and component interface, 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` and/or `EntityManagerFactory` by serializable objects during the object's serialization.

You are responsible for ensuring that all non-transient fields are of these types after the `PrePassivate` method (see ["Configuring a Lifecycle Callback Method for an EJB 3.0 Session Bean"](#) on page 5-2) or `ejbPassivate` method (see ["Configuring a Lifecycle Callback Method for an EJB 2.1 Session Bean"](#) on page 12-3) completes. Within this method, you must set all transient or non-serializable 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:

- ["Accessing an EJB 3.0 EJBContext"](#) on page 29-14
- ["Accessing an EJB 2.1 EJBContext"](#) on page 29-21

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.

What is an EJB 3.0 Entity?

An EJB 3.0 entity is an EJB 3.0 compliant light-weight entity object that manages persistent data, performs complex business logic, potentially uses several dependent Java objects, and can be uniquely identified by a primary key.

EJB 3.0 entities represent persistent data stored in a relational database automatically using container-managed persistence.

EJB 3.0 entities are persistent because their data is stored persistently in some form of data storage system, such as a database: they do survive a server crash, failover, or a network failure. When an entity is re-instantiated, the state of the previous instance is automatically restored.

An entity models a business entity or models multiple actions within a business process. Entity beans are often used to facilitate business services that involve data and computations on that data. For example, an application developer 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.

EJB 3.0 entities can represent fine-grained persistent objects because they are not remotely accessible components.

An EJB 3.0 entity can aggregate objects together and effectively persist data and related objects using container transactional, security, and concurrency services.

This section describes:

- [What are Container-Managed Persistence Fields?](#)
- [What are Container-Managed Relationship Fields?](#)
- [What is the EJB 3.0 Entity Lifecycle?](#)
- [What is an EJB 3.0 Entity Primary Key?](#)
- [How Do You Query for an EJB 3.0 Entity?](#)

For more information, see "[Implementing an EJB 3.0 Entity](#)" on page 6-1.

What are Container-Managed Persistence Fields?

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

By specifying a CMP field, you are instructing OC4J to take responsibility for ensuring that the field's value is persisted to the database.

Using EJB 3.0, all data members are by default considered container-managed persistence fields unless annotated with `@Transient`.

What are 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 CMP 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.

By specifying a CMR field, you are instructing OC4J to take responsibility for ensuring that a reference to one or more related EJB 3.0 entities or EJB 2.1 CMP entity beans is persisted to the database. For this reason, this relationship is often referred to as a CMR or a mapping from one EJB 3.0 entity or EJB 2.1 CMP entity bean 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 we often say that it "knows" about its related object. For example, if an `ProjectEJB` bean knows what `TaskEJB` beans it has and if each `TaskEJB` bean knows what `ProjectEJB` bean it belongs to, 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 EJBs.
- **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 CMP Relationships with any type of multiplicity and with both unidirectional and bi-directional relationships.

For more information, see:

- ["Configuring an EJB 3.0 Entity Container-Managed Relationship Field"](#) on page 7-8
- ["Using EJB 3.0 Query API"](#) on page 8-1

What is the EJB 3.0 Entity Lifecycle?

Table 1–9 lists the optional EJB 3.0 entity lifecycle callback methods you can define using annotations. For EJB 3.0 entities, you do not need to implement these methods.

Table 1–9 Lifecycle Methods for an EJB 3.0 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.
@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.
@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.

Table 1–9 (Cont.) Lifecycle Methods for an EJB 3.0 Entity

Annotation	Description
@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 ["Configuring a Lifecycle Callback Method for an EJB 3.0 Entity"](#) on page 7-14.

What is an EJB 3.0 Entity Primary Key?

Each EJB 3.0 entity instance has a primary key that uniquely identifies it from other instances. The primary key (or the fields contained within a complex primary key) must be a container-managed persistent fields.

All fields within the primary key are restricted to::

- 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 an EJB 3.0 Entity 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 EJB 3.0 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 an EJB 3.0 Entity Primary Key"](#) on page 7-2

How Do You Query for an EJB 3.0 Entity?

In EJB 3.0, you use a `javax.persistence.EntityManager` to create, find, merge, and persist your EJB 3.0 entities (see ["Accessing an EJB 3.0 Entity Using an EntityManager"](#) on page 29-5).

You can express your selection criteria using an appropriate query syntax (see ["Understanding EJB Query Syntax"](#) on page 1-27).

Understanding EJB Query Syntax

[Table 1–15](#) summarizes the types of query syntax you can use to define EJB queries.

Table 1–10 OC4J EJB Query Syntax Support

Query Syntax	See Also
EJB QL	"Understanding EJB Query Syntax" on page 1-27
Native SQL	"Understanding Native SQL Query Syntax" on page 1-29

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

Understanding EJB QL Query Syntax EJB QL is a specification language used to define query semantics in a portable and optimizable format.

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 CMP entity beans, using the abstract schema name and the CMP and CMR 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, CMP field names, and CMR field names to the appropriate database tables and column names. EJB QL is portable to all databases supported by OC4J.

In EJB 3.0, EJB QL syntax includes everything in EJB 2.1 plus additional features such as bulk update and delete, JOIN operations, GROUP BY, HAVING, projection, subqueries, and the use of EJB QL in dynamic queries using the EJB 3.0 `EntityManager` API (see ["Understanding the EJB 3.0 EntityManager Query API"](#) on page 1-18). For complete details, see the EJB 3.0 persistence specification.

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 EJBs 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 - Oracle, MS SQL-Server, IBM DB/2, Informix, and Sybase.
- **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 3.0, OC4J supports all of the enhanced EJB QL features defined in the EJB 3.0 persistence specification, including SQRT and date, time, and timestamp options.

Understanding Native SQL Query Syntax In this release, the TopLink persistence manager takes the query syntax you specify (see ["Understanding EJB Query Syntax"](#) on page 1-27) 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 3.0, you can use `EntityManager` method `createNativeQuery(String sqlString, Class resultType)` to create a native SQL query (see ["Creating a Dynamic Native SQL Query with the EntityManager"](#) on page 29-10).

Note: OC4J does not support `EntityManager` method `createNativeQuery(String sqlString)` nor does it support native SQL named queries (see ["Implementing an EJB 3.0 Named Query"](#) on page 8-1).

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

Understanding the EJB 3.0 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 (see ["What is an EJB 3.0 Named \(Predefined\) Query?"](#) on page 1-18) or dynamic queries (see ["What is an EJB 3.0 Dynamic \(Ad-Hoc\) Query?"](#) on page 1-18).

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

For more information, see:

- ["How Do You Query for an EJB 3.0 Entity?"](#) on page 1-16
- ["Querying for an EJB 3.0 Entity Using the EntityManager"](#) on page 29-8

What is an EJB 3.0 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 an EJB 3.0 Named Query"](#) on page 8-1) and then create and execute the query by name at runtime (see ["Creating a Named Query with the EntityManager"](#) on page 29-9).

In this release OC4J supports only EJB QL named queries.

What is an EJB 3.0 Dynamic (Ad-Hoc) Query? A dynamic query is a query that you can compose, configure, and execute at runtime. You can use dynamic queries in addition to named queries.

OC4J supports both EJB QL (["Creating a Dynamic EJB QL Query with the EntityManager"](#) on page 29-9) and native SQL (["Creating a Dynamic Native SQL Query with the EntityManager"](#) on page 29-10) dynamic 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 29-9).

What is an EJB 2.1 Entity Bean?

An entity bean is an EJB 2.1 EJB component that manages persistent data, performs complex business logic, potentially uses several dependent Java objects, and can be uniquely identified by a primary key (["What is a CMP Entity Bean Primary Key?"](#) on page 1-21 or ["What is a BMP Entity Bean Primary Key?"](#) on page 1-23).

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

- Automatically by the container using a container-managed persistent (CMP) entity bean (see ["What is an EJB 2.1 CMP Entity Bean?"](#) on page 1-19)
- Programmatically through methods implemented in a bean-managed persistent (BMP) entity bean (see ["What is an EJB 2.1 BMP Entity Bean?"](#) on page 1-22). These methods use JDBC, SQLJ, or a persistence framework (such as TopLink) to manage persistence.

For information on choosing between CMP and BMP architectures, see ["When do you use Bean-Managed versus Container-Managed Persistence?"](#) on page 1-35.

Entity beans are persistent because their data is stored persistently in some form of data storage system, such as a database: they do survive a server crash, failover, or a network failure. When an entity bean is re-instantiated, 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 1-25).

An entity bean models a business entity or models multiple actions within a business process. Entity beans are often used to facilitate business services that involve data and computations on that data. For example, an application developer 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:

- [What is an EJB 2.1 CMP Entity Bean?](#)
- [What is an EJB 2.1 BMP Entity Bean?](#)
- [What is Entity Context?](#)
- [How do You Avoid Database Resource Contention?](#)
- [How Do You Query for an EJB 2.1 Entity Bean?](#)
- [When Does Entity Bean Passivation Occur?](#)
- [What are Entity Bean Commit Options?](#)

For more information, see ["Implementing an EJB 2.1 Entity Bean"](#) on page 13-1.

What is an EJB 2.1 CMP Entity Bean?

When you choose to have the container manage your persistent data for an entity bean, you define a container-managed persistence (CMP) entity bean. A CMP entity bean class is an abstract class (the container provides the implementation class that is used at runtime) whose persistent data is specified as container-managed persistence fields (see ["What are Container-Managed Persistence Fields?"](#) on page 1-20) for simple data or as container-managed relationship fields (see ["What are Container-Managed Relationship Fields?"](#) on page 1-20) for relationships with other CMP entity beans. 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 CMP Entity Bean Lifecycle?"](#) on page 1-20), 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 CMP Entity Bean Primary Key?"](#) on page 1-21): the container provides this key for the bean.

For more information, see ["Implementing an EJB 2.1 CMP Entity Bean"](#) on page 13-1.

What are Container-Managed Persistence Fields?

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

By specifying a CMP 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 CMP entity bean are considered non-persistent (transient).

Using EJB 2.1, you must explicitly specify CMP fields (see ["Configuring an EJB 2.1 CMP Entity Bean Container-Managed Persistence Field"](#) on page 14-6).

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 CMP 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.

By specifying a CMR field, you are instructing OC4J to take responsibility for ensuring that a reference to one or more related CMP entity beans is persisted to the database. For this reason, a relationship between CMP entity beans is often referred to as container-managed relationship (CMR) or a mapping from one CMP entity bean 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 we often say that it "knows" about its related object. For example, if an `ProjectEJB` bean knows what `TaskEJB` beans it has and if each `TaskEJB` bean knows what `ProjectEJB` bean it belongs to, 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 EJBs.
- **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 CMP Relationships with any type of multiplicity and with both unidirectional and bi-directional relationships.

For more information, see:

- ["Configuring an EJB 2.1 CMP Entity Bean Container-Managed Relationship Field"](#) on page 14-8
- ["Using EJB 2.1 Query API"](#) on page 16-1

What is the CMP Entity Bean Lifecycle?

[Table 1-11](#) lists the EJB 2.1 lifecycle methods, as specified in the `javax.ejb.EntityBean` interface, that a CMP entity bean must implement. For EJB

2.1 CMP entity beans, you must at the least provide an empty implementation for all callback methods.

Table 1–11 Lifecycle Methods for an EJB 2.1 CMP Entity Bean

EJB Method	Description
<code>ejbCreate</code>	<p>You must implement an <code>ejbCreate</code> method corresponding to each <code>create</code> method declared in the home interface. When the client invokes the <code>create</code> method, the container first invokes the constructor to instantiate the object, then it invokes the corresponding <code>ejbCreate</code> method.</p> <p>For a CMP entity bean, use this method to initialize container-managed persistent fields.</p> <p>The return type of all <code>ejbCreate</code> methods is the type of the bean's primary key.</p> <p>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 <code>null</code>.</p>
<code>ejbPostCreate</code>	<p>The container invokes this method after the environment is set. For each <code>ejbCreate</code> method, an <code>ejbPostCreate</code> method must exist with the same arguments.</p> <p>For a CMP entity bean, you can leave this implementation empty or use your implementation to initialize parameters within or from the entity context.</p>
<code>ejbRemove</code>	<p>The container invokes this method before it ends the life of the entity bean.</p> <p>For a CMP entity bean, 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.</p>
<code>ejbStore</code>	<p>The container invokes this method right before a transaction commits. It saves the persistent data to an outside resource, such as a database.</p> <p>For a CMP entity bean, you can leave this implementation empty.</p>
<code>ejbLoad</code>	<p>The container invokes this method when the data should be reinitialized from the database. This normally occurs after activation of an entity bean.</p> <p>For a CMP entity bean, you can leave this implementation empty.</p>
<code>ejbActivate</code>	<p>The container calls this method directly before it activates an object that was previously passivated. Perform any necessary reacquisition of resources in this method.</p>
<code>ejbPassivate</code>	<p>The container calls this method before it passivates the object. Release any resources that can be easily re-created in <code>ejbActivate</code>, and save storage space. Normally, you want to free resources that cannot be passivated, such as sockets or database connections. Retrieve these resources in the <code>ejbActivate</code> method.</p>

What is a CMP Entity Bean Primary Key?

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:

- 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 an EJB 2.1 CMP Entity Bean Primary Key Field"](#) on page 14-1).
- Define a composite primary key class made up of one or more well-known serializable Java primitive and/or object types within a `<name>PK` class that is

serializable (see ["Configuring an EJB 2.1 CMP Entity Bean Composite Primary Key Class"](#)).

You can assign primary key values yourself, or more typically, you can create an auto-generated primary key (see ["Configuring EJB 2.1 CMP Entity Bean Automatic Primary Key Generation"](#) on page 14-4).

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 an EJB 2.1 CMP Entity Bean Primary Key"](#) on page 14-1.

What is an EJB 2.1 BMP Entity Bean?

When you choose to manage your persistent data for an entity bean yourself, you define a bean-managed persistence (BMP) entity bean. A BMP entity bean class is a concrete class (you provide the implementation that is used at runtime) whose persistent data is specified as bean-managed persistence fields (see ["What are Bean-Managed Persistence Fields?"](#) on page 1-22) for simple data or as bean-managed relationship fields (see ["What are Bean-Managed Relationship Fields?"](#) on page 1-22) for relationships with other BMP entity beans. 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 BMP Entity Bean Lifecycle?"](#) on page 1-22). 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 BMP Entity Bean Primary Key?"](#) on page 1-23).

You can specify a BMP entity bean as read-only (see ["Configuring a Read-Only BMP Entity Bean"](#) on page 15-1) and take advantage of the optimizations that OC4J provides read-only BMP entity beans depending on the commit option you choose (see ["What are Entity Bean Commit Options?"](#) on page 1-26).

For more information, see ["Implementing an EJB 2.1 BMP Entity Bean"](#) on page 13-6.

What are Bean-Managed Persistence Fields?

With bean-managed persistence, the code that you write determines what BMP entity bean fields are persistent.

What are Bean-Managed Relationship Fields?

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

What is the BMP Entity Bean Lifecycle?

[Table 1-12](#) lists the lifecycle methods, as specified in the `javax.ejb.EntityBean` interface, that a BMP entity bean must implement.

For a BMP entity bean, you must provide a complete implementation of all lifecycle methods.

Table 1–12 EJB Lifecycle Methods for a BMP Entity Bean

EJB Method	Description
<code>ejbCreate</code>	You must implement an <code>ejbCreate</code> method corresponding to each <code>create</code> method declared in the home interface. When the client invokes the <code>create</code> method, the container first invokes the constructor to instantiate the object, then it invokes the corresponding <code>ejbCreate</code> method. The <code>ejbCreate</code> method performs the following: <ul style="list-style-type: none"> ■ creates any persistent storage for its data, such as database rows ■ initializes a unique primary key and returns it
<code>ejbPostCreate</code>	The container invokes this method after the environment is set. For each <code>ejbCreate</code> method, an <code>ejbPostCreate</code> method must exist with the same arguments. This method can be used to initialize parameters within or from the entity context.
<code>ejbRemove</code>	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.
<code>ejbStore</code>	The container invokes this method right before a transaction commits. It saves the persistent data to an outside resource, such as a database.
<code>ejbLoad</code>	The container invokes this method when the data should be reinitialized from the database. This normally occurs after activation of an entity bean.
<code>ejbActivate</code>	The container calls this method directly before it activates an object that was previously passivated. Perform any necessary reacquisition of resources in this method.
<code>ejbPassivate</code>	The container calls this method before it passivates the object. Release any resources that can be easily re-created in <code>ejbActivate</code> , and save storage space. Normally, you want to free resources that cannot be passivated, such as sockets or database connections. Retrieve these resources in the <code>ejbActivate</code> method.

For more information, see ["Configuring a Lifecycle Callback Method for an EJB 2.1 BMP Entity Bean"](#) on page 15-4.

What is a BMP Entity Bean Primary Key?

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:

- 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 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`.
- Define the type of the primary key as a serializable object within a `<name>PK` class that is serializable.

In either case, for a BMP entity bean, you create the primary key in the `ejbCreate` method.

What is Entity Context?

OC4J maintains a `javax.ejb.EntityContext` for each EJB 2.1 CMP or BMP entity bean 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:

- ["Implementing the setEntityContext and unsetEntityContext Methods"](#) on page 13-20
- ["Accessing an EJB 2.1 EJBContext"](#) on page 29-21.

How do You Avoid Database Resource Contention?

OC4J and the TopLink persistence manager use a combination of transaction isolation (see ["Transaction Isolation"](#) on page 1-24) and concurrency mode (see ["Concurrency \(Locking\) Mode"](#) on page 1-25) 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 1–13](#). 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.
- Non repeatable 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 1–13 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 persistence manager provide read-committed transaction isolation.

Using a TopLink persistence manager customization file (see ["Customizing the TopLink Persistence Manager"](#) on page 3-5), you can configure the transaction isolation mode. For more information, see:

- "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 container-managed persistence (CMP) entity beans.

Bean-managed persistence entity beans manage the resource locking within the bean implementation themselves.

CMP entity bean concurrency modes include:

- **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

By default, the TopLink persistence manager enforces optimistic locking by using a numeric version field (also known as a write-lock field) that TopLink updates each time an object change is committed.

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 per bean and apply on the transaction boundaries.

By default, OC4J and the TopLink persistence manager use optimistic locking and all CMP entity beans are not read-only.

Using a TopLink persistence manager customization file (see ["Customizing the TopLink Persistence Manager"](#) on page 3-5), you can configure the concurrency mode on a per-CMP EJB basis. For more information, see:

- "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*

When Does Entity Bean Passivation Occur?

Entity bean passivation applies only to EJB 2.1 CMP entity beans.

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 1–14 lists the commit options as defined by the EJB 2.1 specification and indicates which are supported by OC4J.

Table 1–14 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 BMP entity beans only (see "Commit Options and BMP Applications" on page 1-27).

² CMP entity beans only (see "Commit Options and CMP Applications" on page 1-26).

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 lifecycle 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:

- ["Customizing the TopLink Persistence Manager"](#) on page 3-5
- ["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 (see ["Configuring BMP Commit Options"](#) on page 15-2).

When you configure a BMP entity bean 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 BMP Entity Bean"](#) on page 15-1.

You can use BMP commit option A and read-only BMP entity beans independently (that is, you can configure a BMP entity bean with commit option A without using read-only and you can use read-only without configuring a BMP entity bean 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 1-30 and ["Understanding Select Methods"](#) on page 1-32).

In either case, you express your selection criteria using an appropriate query syntax (see ["Understanding EJB Query Syntax"](#) on page 1-27).

For more information, see ["Using EJB 2.1 Query API"](#) on page 16-1.

Understanding EJB Query Syntax

[Table 1–15](#) summarizes the types of query syntax you can use to define EJB queries.

Table 1–15 *OC4J EJB Query Syntax Support*

Query Syntax	See Also
EJB QL	"Understanding EJB Query Syntax" on page 1-27
TopLink	"Understanding TopLink Query Syntax" on page 1-28
Predefined Finder	"Predefined TopLink Finders" on page 1-30
Default Finder	"Default TopLink Finders" on page 1-31
Custom Finder	"Custom TopLink Finders" on page 1-32
Custom Select	"Custom TopLink Select Methods" on page 1-33
Native SQL	"Understanding Native SQL Query Syntax" on page 1-29

Oracle recommends 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 1-30 and ["Understanding Select Methods"](#) on page 1-32) 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 CMP entity beans, using the abstract schema name and the CMP and CMR 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, CMP field names, and CMR 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 1-30 and ["Custom TopLink Select Methods"](#) on page 1-33). 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 EJBs 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 - 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 Persistence Manager"](#) on page 3-4), 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 1-29).

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 1-30 and ["Custom TopLink Select Methods"](#) on page 1-33).

For more information, see:

- "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 like this:

```
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 this release, the TopLink persistence manager takes the query syntax you specify ("[Understanding EJB QL Query Syntax](#)" on page 1-27 or "[Understanding TopLink Query Syntax](#)" on page 1-28) 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:

- default finders that take a native SQL string (see ["Default TopLink Finders"](#) on page 1-31)
- custom finder or select methods that use native SQL calls (see ["TopLink Finders"](#) on page 1-30 and ["Custom TopLink Select Methods"](#) on page 1-33)

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 CMP entity beans. Using EJB 2.1, you can:

- Expose any of the predefined and default finders that OC4J and the TopLink persistence manager provide to all CMP entity beans (see ["Predefined TopLink Finders"](#) on page 1-30 and ["Default TopLink Finders"](#) on page 1-31).
- 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 1-32).

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 1-30) and default (see ["Default TopLink Finders"](#) on page 1-31) 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 1-32).

Predefined TopLink Finders

[Table 1–16](#) lists the predefined finders you can expose for EJB 2.1 CMP entity beans. The TopLink persistence manager reserves the method names listed in [Table 1–16](#).

Table 1–16 *Predefined TopLink CMP Finders*

Method	Arguments	Return
<code>findAll</code>	<code>()</code>	<code>Collection</code>
<code>findManyByEJBQL</code>	<code>(String ejbql)</code> <code>(String ejbql, Vector args)</code>	<code>Collection</code>
<code>findManyByQuery</code>	<code>(DatabaseQuery query)</code> <code>(DatabaseQuery query, Vector args)</code>	<code>Collection</code>
<code>findManyBySQL</code>	<code>(String sql)</code> <code>(String sql, Vector args)</code>	<code>Collection</code>

Table 1–16 (Cont.) Predefined TopLink CMP Finders

Method	Arguments	Return
findByPrimaryKey	(Object primaryKeyObject)	EJBObject or EJBLocalObject ¹
findOneByEJBQL	(String ejbql)	Component interface
	(String ejbql, Vector args)	EJBObject or EJBLocalObject ¹
findOneByQuery	(DatabaseQuery query)	Component interface
	(DatabaseQuery query, Vector args)	EJBObject or EJBLocalObject ¹
findOneBySQL	(String sql)	Component interface
	(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. We 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 a `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. We 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 CMP entity bean 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 CMP field or entity bean references of container-managed relationship (CMR) 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<MethodName>(...) 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 My Select Method Return?"](#) on page 1-32).
- 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 1-33).

What Type Can My 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 CMP or CMR field.

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

- All values must be returned as objects; 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.

- CMP values: If you return multiple CMP values, the container returns a `Collection` of objects whose type it determines from the EJB QL select statement.
- CMR values: If you return multiple CMR 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 take advantage of 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 EJB 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 Providers Can I use with My MDB?"](#) on page 2-16). You associate the MDB with an existing message provider and the container handles much of the setup required, as follows:

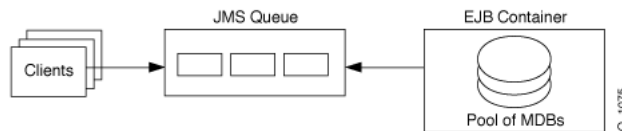
- The EJB container creates a consumer of type `QueueReceiver` or `TopicSubscriber` for the listener.
- At deployment time, the EJB container registers the MDB with the consumer, which is either a `QueueReceiver` or `TopicSubscriber`, and its factory.
- The EJB container specifies the message acknowledgment 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 EJBs 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 1–3](#) 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 1–3 Message Driven Beans



This section describes:

- [What is the Message-Driven Bean Lifecycle?](#)
- [What is Message Driven Context?](#)

For more information, see:

- ["Implementing an EJB 3.0 MDB" on page 9-1](#)
- ["Implementing an EJB 2.1 MDB" on page 17-1](#)

What is the Message-Driven Bean Lifecycle?

The lifecycle for EJB 3.0 (see [Table 1–17](#)) and EJB 2.1 (see [Table 1–18](#)) message-driven beans are identical. The difference is in how you register lifecycle callback methods.

[Table 1–17](#) lists the optional EJB 3.0 message-driven bean lifecycle callback methods you can define using annotations. For EJB 3.0 message-driven beans, you do not need to implement these methods.

Table 1–17 Lifecycle 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 1–18](#) lists the EJB 2.1 lifecycle 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 1–18 Lifecycle Methods for an EJB 2.1 Message-Driven Bean

EJB Method	Description
<code>ejbCreate</code>	The container invokes this method right before it creates the bean. A message-driven bean must do nothing in this method.
<code>ejbRemove</code>	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:

- ["Configuring a Lifecycle Callback Method for an EJB 3.0 MDB"](#) on page 10-6

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:

- ["Accessing an EJB 3.0 EJBContext"](#) on page 29-14
- ["Accessing an EJB 2.1 EJBContext"](#) on page 29-21

Which Type of EJB Should You Use?

This section describes:

- [Which Type of Session Bean Should You Use?](#)
- [When do you use Bean-Managed versus Container-Managed Persistence?](#)
- [What is the Difference Between Session and Entity Beans?](#)

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 1–19](#) provides a definition for both BMP and CMP, and a summary of the programmatic and declarative differences between them.

Table 1–19 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 <code>ejbStore</code> , <code>ejbLoad</code> , <code>ejbCreate</code> , and <code>ejbRemove</code> <code>EntityBean</code> methods. These methods must contain logic for saving and restoring the persistent data. For example, the <code>ejbStore</code> 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 <code>ejbStore</code> and <code>ejbLoad</code> for preparing the data before the commit or for manipulating the data after it is refreshed from the database. The container always invokes the <code>ejbStore</code> method right before the commit. In addition, it always invokes the <code>ejbLoad</code> method right after reinstating CMP data from the database.
Finder methods allowed	The <code>findByPrimaryKey</code> method and other finder methods are allowed.	The <code>findByPrimaryKey</code> method and other finder methods clause are allowed.
Defining CMP fields	N/A	Required within the EJB deployment descriptor. The primary key must also be declared as a CMP field.
Mapping CMP 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 <code>TopLink</code> persistence manager.

With CMP, you can build components to the EJB 2.0 specification that can save the state of your EJB to any J2EE 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:

- ["What is an EJB 2.1 CMP Entity Bean?"](#) on page 1-19
- ["What is an EJB 2.1 BMP Entity Bean?"](#) on page 1-22

What is the Difference Between Session and Entity Beans?

The major differences between session and entity beans are that entity beans involve a framework for persistent data management, a persistent identity, and complex business logic. [Table 1–20](#) illustrates the different interfaces for session and entity beans. Notice that the difference between the two types of EJBs exists within the bean class and the primary key. All of the persistent data management is done within the bean class methods.

Table 1–20 Session and Entity Bean Differences

J2EE Subject	Entity Bean	Session Bean
Local interface	Extends <code>javax.ejb.EJBLocalObject</code>	Extends <code>javax.ejb.EJBLocalObject</code>
Remote interface	Extends <code>javax.ejb.EJBObject</code>	Extends <code>javax.ejb.EJBObject</code>

Table 1–20 (Cont.) Session and Entity Bean Differences

J2EE Subject	Entity Bean	Session Bean
Local Home interface	Extends <code>javax.ejb.EJBLocalHome</code>	Extends <code>javax.ejb.EJBLocalHome</code>
Remote Home interface	Extends <code>javax.ejb.EJBHome</code>	Extends <code>javax.ejb.EJBHome</code>
Bean class	Extends <code>javax.ejb.EntityBean</code>	Extends <code>javax.ejb.SessionBean</code>
Primary key	Used to identify and retrieve specific bean instances	Not used for session beans. Stateful session beans do have an identity, but it is not externalized.

Understanding EJB Application Development

This chapter describes:

- [How Should You Develop EJB Applications?](#)
- [What OC4J Services Can You Use with an EJB?](#)
- [How do You Package and Deploy an EJB Application?](#)
- [How Do You Use an EJB in Your Application?](#)
- [Understanding EJB JNDI Services](#)
- [Understanding EJB Data Source Services](#)
- [Understanding EJB Transaction Services](#)
- [Understanding EJB Security Services](#)
- [Understanding Message Services](#)
- [Understanding OC4J EJB Application Clustering Services](#)
- [Understanding EJB Timer Services](#)

How Should You Develop EJB Applications?

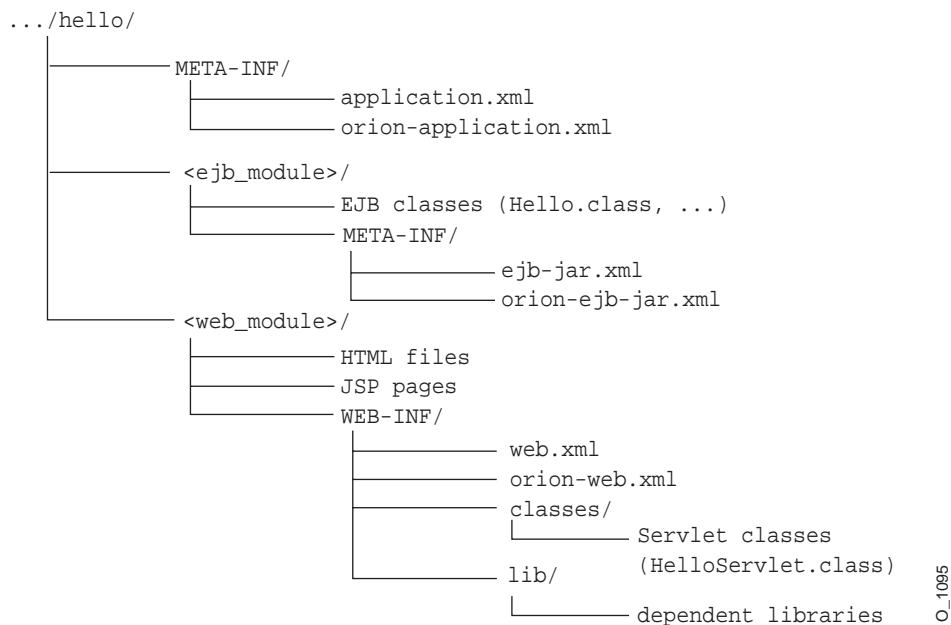
This section describes:

- [Understanding the EJB Application Directory Structure](#)
- [Using EJB Development Tools](#)

Understanding the EJB Application Directory Structure

Although you can develop your application in any manner, we encourage you to use consistent naming to locate your application easily. One method would be to implement your enterprise Java application under a single parent directory structure, separating each module of the application into its own subdirectory.

Notice in [Figure 2-1](#) that the EJB and Web modules exist under the `hello` application parent directory and are developed separately in their own directory.

Figure 2–1 Hello Directory Structure

Note: For EJB modules, the top of the module (`ejb_module`) represents the start of a search path for classes. As a result, classes belonging to packages are expected to be located in a nested directory structure beneath this point. For example, a reference to a package class `myapp.Hello.class` is expected to be located in `...hello/ejb_module/myapp/Hello.class`.

Using EJB Development Tools

This section describes developing EJB applications:

- [Using JDeveloper](#)
- [Using Eclipse](#)
- [Using TopLink Workbench](#)

Using JDeveloper

Oracle JDeveloper greatly simplifies J2EE application development 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

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:

- 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:

- "Understanding the TopLink Workbench" in the *Oracle TopLink Developer's Guide*
- ["Understanding EJB Deployment Descriptor Files"](#) on page 2-7

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 you can use them with.

Table 2–1 OC4J Services and EJB Support

OC4J Service	Stateful Session Bean	Stateless Session Bean	CMP Entity Bean	BMP Entity Bean	Message-Driven Bean
"Understanding EJB JNDI Services" on page 2-13	✓	✓	✓	✓	✓
"Understanding EJB Data Source Services" on page 2-13	✓	✓	✓	✓	✓
"Understanding EJB Transaction Services" on page 2-14	✓	✓	✓	✓	✓
"Understanding EJB Security Services" on page 2-16	✓	✓	✓	✓	✓
"Understanding Message Services" on page 2-16					✓
"Understanding OC4J EJB Application Clustering Services" on page 2-20	✓				
"Understanding J2EE Timer Services" on page 2-23		✓	✓	✓	✓
"Understanding OC4J Cron Timer Services" on page 2-23		✓	✓ ¹	✓ ²	✓

¹ EJB 2.1 only.

² EJB 2.1 only.

For more information on OC4J services, see the appropriate OC4J guide as shown in [Table 2–2](#):

Table 2–2 Location of Information for J2EE Subjects

J2EE Subject	The Subject is Documented in this OC4J Documentation Book
JNDI	<i>Oracle Containers for J2EE Services Guide</i>
Data Source	<i>Oracle Containers for J2EE Services Guide</i>
RMI and RMI/IIOP	<i>Oracle Containers for J2EE Services Guide</i>
Transactions (JTA)	<i>Oracle Containers for J2EE Services Guide</i>
Security	<i>Oracle Application Server Security Guide</i>
CSIv2	<i>Oracle Containers for J2EE Services Guide</i>
JMS	<i>Oracle Containers for J2EE Services Guide</i>
Clustering	<i>Oracle Containers for J2EE Services Guide</i>
Timers	<i>Oracle Containers for J2EE Services Guide</i>
J2CA	<i>Oracle Containers for J2EE Services Guide</i>
Java Object Cache	<i>Oracle Containers for J2EE Services Guide</i>
Web Services	<i>Oracle Application Server Web Services Developer's Guide</i>
HTTPS	<i>Oracle Containers for J2EE Services Guide</i>
Optimization	<i>Oracle Application Server Performance Guide</i>
Default Persistence	<i>Oracle TopLink Developer's Guide</i>

How do You Package and Deploy an EJB Application?

This section describes the following:

- [General Packaging and Deployment Procedure](#)
- [Understanding EJB Deployment Descriptor Files](#)
- [Understanding Packaging](#)
- [Understanding Deployment](#)

General Packaging and Deployment Procedure

In general, to package and deploy an EJB application:

1. Create the Deployment Descriptor

After implementing and compiling your classes, you must create the standard J2EE EJB deployment descriptor for all beans in the module. The XML deployment descriptor (defined in the `ejb-jar.xml` file) describes the EJB module of the application. It describes the types of beans, their names, and attributes. The structure for this file is defined by the XML schema document (XSD) at http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd.

Any EJB container services that you want to configure are designated in the deployment descriptor. For information, see ["What OC4J Services Can You Use with an EJB?"](#) on page 2-3.

You can also configure OC4J-specific options in the `orion-ejb-jar.xml` file (or using Application Server Control after deployment). For more information, see:

- ["Understanding EJB Deployment Descriptor Files"](#) on page 2-7
- ["How Do Specify Vendor-Specific Configuration in an EJB 3.0 Application?"](#) on page 2-10

After creation, place the deployment descriptors for the EJB application in the META-INF directory that is located in the same directory as the EJB classes (see [Figure 2-1](#)).

The following example shows the sections that are necessary for the `Hello` example, which implements both a remote and a local interface.

[Example 2-1](#) shows the deployment descriptor for a version of the `Hello` example that uses a stateless session bean. This example defines both the local and remote interfaces. You do not have to define both interface types; you may define only one of them.

Example 2-1 *ejb-jar.xml Deployment Descriptor for Hello Bean Application*

```
<?xml version="1.0" encoding="UTF-8" ?>
<ejb-jar
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                      http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd"
  version="2.1">
  <enterprise-beans>
    <session>
      <description>no description</description>
      <display-name>HelloBean</display-name>
      <ejb-name>HelloBean</ejb-name>
      <home>hello.HelloHome</home>
      <remote>hello.Hello</remote>
      <local-home>hello.HelloLocalHome</local-home>
      <local>hello.HelloLocal</local>
      <ejb-class>hello.HelloBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>

  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>HelloBean</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Supports</trans-attribute>
    </container-transaction>
    <security-role>
      <role-name>users</role-name>
    </security-role>
  </assembly-descriptor>
</ejb-jar>
```

2. Archive the EJB Application

After you have finalized your implementation and created the deployment descriptors, archive your EJB application into a JAR file. The JAR file should include all EJB application files and the deployment descriptor.

Note: If you have included a Web application as part of this enterprise Java application, follow the instructions for building the Web application in the *Oracle Containers for J2EE Developer's Guide*.

For example, to archive your compiled EJB class files and XML files for the Hello example into a JAR file, perform the following in the `.../hello/ejb_module` directory:

```
% jar cvf helloworld-ejb.jar .
```

This archives all files contained within the `ejb_module` subdirectory within the JAR file.

3. Prepare the EJB Application for Assembly

To prepare the application for deployment, you do the following:

- a. Modify the `application.xml` file with the modules of the enterprise Java application.

The `application.xml` file acts as the manifest file for the application and contains a list of the modules that are included within your enterprise application. You use each `<module>` element defined in the `application.xml` file to designate what comprises your enterprise application as [Table 2–3](#) shows.

Table 2–3 *Module Elements in the application.xml File*

Element	Contents
<code><ejb></code>	EJB JAR filename
<code><web></code>	Web WAR filename in the <code><web-uri></code> sub-element, and its context in the <code><context></code> sub-element
<code><java></code>	Client JAR filename, if any

As [Figure 2–1](#) shows, the `application.xml` file is located under a `META-INF` directory under the parent directory for the application. The JAR, WAR, and client JAR files should be contained within this directory. Because of this proximity, the `application.xml` file refers to the JAR and WAR files only by name and relative path—not by full directory path. If these files were located in subdirectories under the parent directory, then these subdirectories must be specified in addition to the filename.

[Example 2–2](#) modifies the `<ejb>`, `<web>`, and `<java>` module elements within `application.xml` for the Hello EJB application that also contains a Java client that interacts with the EJB.

Example 2–2 *application.xml Deployment Descriptor for Hello Bean*

```
<?xml version="1.0"?>
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application 1.2//EN"
"http://java.sun.com/j2ee/dtds/application_1_2.dtd">
<application>
  <display-name>helloworld j2ee application</display-name>
  <description>
    A sample J2EE application that uses a Helloworld Session Bean
    on the server and calls from java/servlet/JSP clients.
  </description>
  <module>
    <ejb>helloworld-ejb.jar</ejb>
  </module>
  <module>
    <web>
      <web-uri>helloworld-web.war</web-uri>
      <context-root>/helloworld</context-root>
    </web>
```

```

</module>
<module>
  <java>helloworld-client.jar</java>
</module>
</application>

```

- b. Archive all elements of the application into an EAR file.

Create the EAR file that contains the JAR, WAR, and XML files for the application. Note that the `application.xml` file serves as the EAR manifest file.

To create the `helloworld.ear` file, execute the following in the `hello` directory contained in [Figure 2-1](#):

```
% jar cvf helloworld.ear .
```

This step archives the `application.xml`, the `helloworld-ejb.jar`, the `helloworld-web.war`, and the `helloworld-client.jar` files into the `helloworld.ear` file.

Understanding EJB Deployment Descriptor Files

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

[Table 2-4](#) 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 the deployment descriptor applies to and whether or not the deployment descriptor is optional, required, or not applicable to the EJB specification you are using.

Table 2-4 OC4J EJB Deployment Descriptor Files

Deployment Descriptor File	Session Bean	Entity	Entity Bean	Message-Driven Bean	EJB 3.0	EJB 2.1
What is the ejb-jar.xml File?	✓	✓	✓	✓	Optional	Required
What is the orion-ejb-jar.xml File?		✓	✓	✓	Optional	Optional
What is the toplink-ejb-jar.xml File?		✓	✓		Optional	Required
What is the ejb3-toplink-sessions.xml File?		✓			Optional	Not Applicable

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 J2EE 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 2-8 and ["What is the toplink-ejb-jar.xml File?"](#) on page 2-8).

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 except object-relational entity mapping configuration (see ["Implementing an EJB 3.0 Entity"](#) on page 6-1): all object-relational entity mapping configuration must be done using annotations only. Configuration in the `ejb-jar.xml` file overrides annotations.

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.

Note: In this release (10.1.3.0.0), OC4J support for EJB 3.0 features is based on a pre-release version of EJB 3.0. Consequently, OC4J support for some EJB 3.0 features may differ from what is specified in the `ejb-jar_3_0.xsd` (for example, interceptors and lifecycle listeners). For a list of options that OC4J does not support, see the *Oracle Application Server Release Notes*.

You may need to make code changes to your EJB 3.0 OC4J application after the EJB 3.0 specification is finalized and OC4J is updated to full EJB 3.0 compliance.

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 2-7).

For more information, see ["Configuring the orion-ejb-jar.xml File"](#) on page 26-2.

EJB 3.0 If you are using EJB 3.0, this file is mandatory for all OC4J-proprietary options because there are no OC4J-proprietary annotations. Alternatively, you could deploy without an `orion-ejb-jar.xml` file and configure OC4J-proprietary options with Application Server Control (see ["How Do Specify Vendor-Specific Configuration in an EJB 3.0 Application?"](#) on page 2-10).

EJB 2.1 If you are using EJB 2.1, this file is mandatory for all OC4J-proprietary options.

XML Reference This deployment descriptor file conforms to the XML schema document at http://www.oracle.com/technology/oracleas/schema/orion-ejb-jar-10_0.xsd.

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 persistence configuration descriptor file, and, when used, it describes TopLink project-level options (see "" in the *Oracle TopLink Developer's Guide*) such as TopLink descriptors and mappings.

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, this file is only used to customize TopLink persistence manager configuration (see ["Customizing the TopLink Entity Manager"](#) on page 3-2). If you use this file to customize the TopLink persistence manager, you must also use an `ejb3-toplink-sessions.xml` file (see ["What is the ejb3-toplink-sessions.xml File?"](#) on page 2-9).

EJB 2.1 If you are using EJB 2.1, this file is optional. If you omit this file from your application, you can configure OC4J to automatically construct it for your (see ["Configuring Default Mappings"](#) on page 14-9). Alternatively, you can use this file to configure TopLink persistence options yourself (see ["Customizing the TopLink Persistence Manager"](#) on page 3-5).

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 persistence configuration descriptor file, and, when used, 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.

This file provides a reference to the primary project (see ["What is the toplink-ejb-jar.xml File?"](#) on page 2-8), 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, this file is only used to customize TopLink persistence manager configuration (see ["Customizing the TopLink Entity Manager"](#) on page 3-2). If you use this file to customize the TopLink persistence manager, you may also use a `toplink-ejb-jar.xml` file (see ["What is the toplink-ejb-jar.xml File?"](#) on page 2-8).

EJB 2.1 If you are using EJB 2.1, this 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*).

Understanding Packaging

The J2EE architecture provides a variety of ways to package (or assemble) your application and its various J2EE components.

For more information, see ["Packaging an EJB Application"](#) on page 27-1.

Understanding Deployment

After you package your J2EE application, to execute the application and make it available to end users, you deploy it to OC4J.

This section describes:

- [How Do Specify Vendor-Specific Configuration in an EJB 3.0 Application?](#)
- [In What Order does OC4J Deploy EJB Modules?](#)

For more information, see ["Deploying an EJB Application to OC4J"](#) on page 28-1.

How Do Specify Vendor-Specific Configuration in an EJB 3.0 Application?

When you deploy an EJB 3.0 application, you must specify any vendor-specific configuration in one of the following ways:

- Package an `orion-ejb-jar.xml` with the desired vendor-specific configuration and deploy.
- After deployment, use the Application Server Control deployment profile to make the vendor-specific configuration.

For example, you can use annotations to define security roles but defining role-to-group mappings requires vendor-specific configuration. In this case, you can either define the role-to-group mappings in an `orion-ejb-jar.xml` file and package that in your application or, you can deploy your application without an `orion-ejb-jar.xml` and use Application Server Control to make this vendor-specific configuration after deployment.

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 base don natural ordering per component type.

For example, consider the `application.xml` file shown in [Example 2-3](#).

Example 2-3 *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

How Do You Use an EJB in Your Application?

In general, you use an EJB from a client (see ["Understanding Client Access"](#) on page 2-11).

You can also use EJBs to implement fine-grained control over method invocation flow (see ["Understanding EJB 3.0 Interceptors"](#) on page 2-11).

In a deployed EJB application, you can exploit the component nature of a J2EE 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 EJB 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 EJB 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 or message-driven bean message listener method. When a client invokes a session bean business method or message-driven bean message listener method, OC4J intercepts the client invocation and invokes your interceptor method before allowing the client invocation to proceed.

This section describes:

- [Interceptor Restrictions](#)
- [Interceptors and Invocation Context](#)

For more information, see:

- ["Configuring an Interceptor on an EJB 3.0 Session Bean"](#) on page 5-3
- ["Configuring an Interceptor on an EJB 3.0 MDB Message Listener Method"](#) on page 10-5

Note: In this release, OC4J interceptor support does not comply with the functionality specified in the EJB 3.0 public review draft. If you use interceptors, you may need to make code changes to your EJB 3.0 OC4J application after the EJB 3.0 specification is finalized and OC4J is updated to full EJB 3.0 compliance. For more information, see ["Understanding EJB Support in OC4J"](#) on page 3-1.

Interceptor Restrictions

You can use interceptors with:

- stateless session beans
- stateful session beans
- message driven beans

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

In this release, you must define an interceptor as a method of the bean class to which it applies. You may define only one interceptor method per class.

An interceptor method may not be a business method.

An interceptor method has the following signature:

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

Within an interceptor, you can use the `InvocationContext` to access client invocation metadata (see ["Interceptors and Invocation Context"](#) on page 2-12).

An interceptor must observe the following transaction restrictions:

- 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 runtime 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 21-5.

- When using container-managed transactions (see ["Container-Managed Transaction \(CMT\)"](#) on page 2-15), 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.

Interceptors and Invocation Context

Using the `InvocationContext` API, you can access (and modify) all the metadata relevant to the client invocation and implement fine grained control over method invocation flow by selectively choosing whether or not to allow the client invocation to proceed. Using EJB 3.0 interceptors, you can:

- Modify parameters before they're passed to the bean
- Modify the value returned from the bean
- Catch and swallow method exceptions
- Interrupt the call completely (for example, to implement your own security framework)
- Provide method profiling

Understanding EJB Administration

After you deploy your J2EE application, you can use J2EE administration features to monitor and optimize your application at runtime.

For more information, see:

- ["Administering an EJB Application"](#) on page 31-1

- ["Optimizing EJB Performance"](#) on page 32-1

Understanding EJB JNDI Services

The Java Naming and Directory Interface (JNDI) provides your J2EE application with a unified interface to multiple naming and directory services. You use JNDI to organize and locate components in a distributed J2EE environment.

You can define environment references for J2EE components and associated JNDI properties.

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

- JNDI initial context
- EJB context
- `@Resource` injection (EJB 3.0 only)

For more information, see ["Configuring JNDI Services"](#) on page 19-1.

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:

- [What Types of Data Source does OC4J 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:

- ["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 source:

- **Managed:** a managed data source is an OC4J-provided implementation of the `java.sql.DataSource` interface that acts as a wrapper for a JDBC driver or data source.
- **Native:** a native data source is a JDBC vendor-provided implementation of the `java.sql.DataSource` interface.

[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

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.

For more information, see:

- ["How Does OC4J Handle Multiple Data Sources?"](#) on page 2-14
- ["What is a Default Data Source?"](#) on page 2-14

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 application you want to access the default data source from:

- ["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 20-3

How Does OC4J Handle Multiple Data Sources?

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

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` for all entity beans.

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

For more information, see:

- ["In What Order does OC4J Deploy EJB Modules?"](#) on page 2-10
- ["What is a Default Data Source?"](#) on page 2-14

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 EJBs during design or deployment and then let the OC4J take over the responsibility of transaction management.

All entity beans with CMP and CMR relationships must be involved in a transaction. As such, you cannot define any entity bean with a transaction attribute of `NEVER`, `SUPPORTS`, or `NOT_REQUIRED` as this would put the entity outside of a transaction.

This section describes:

- [Who Manages a Transaction?](#)
- [Who Begins and Commits a Transaction?](#)
- [How do I Participate in a Global or Two-Phase Commit Transaction?](#)

For more information, see:

- ["Configuring Transaction Services"](#) on page 21-1

- ["Transaction Best Practices"](#) on page 21-3
- "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 ["Container-Managed Transaction \(CMT\)"](#) on page 2-15) or the bean (["Bean-Managed Transaction \(BMT\)"](#) on page 2-15).

Container-Managed Transaction (CMT)

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

Using EJB 3.0, you can use either CMT or BMT (see ["Bean-Managed Transaction \(BMT\)"](#) on page 2-15).

Using EJB 2.1, you can only use CMT.

Bean-Managed Transaction (BMT)

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

Using EJB 3.0, you can use either BMT or CMT (see ["Container-Managed Transaction \(CMT\)"](#) on page 2-15).

Using EJB 2.1, you can only use CMT.

Who Begins and Commits a Transaction?

Client-controlled transactions are started explicitly by your application by way of the `javax.transaction.UserTransaction` interface.

Container-controlled transactions are started implicitly by the container to satisfy the transaction attribute configuration when a bean method is invoked in the absence of a client-controlled transaction.

[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.

Table 2-6 EJB Transaction Support by Transaction Attribute

Transaction Attribute	Client-Controlled Transaction Exists	Client-Controlled Transaction Does Not Exist
NotSupported	Use no transaction	Use no transaction
Supports	Use client-controlled transaction	Use no transaction
Required	Use client-controlled transaction	Use container-controlled transaction
RequiresNew	Use client-controlled transaction	Use container-controlled transaction
Mandatory	Use client-controlled transaction	Exception raised
Never	Exception raised	Use no transaction

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.

This applies to both EJB 3.0 and EJB 2.1.

How do I Participate in a Global or Two-Phase Commit 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, transaction coordination functionality is now located in OC4J, replacing in-database coordination, which is now deprecated. Also, the middle-tier coordinator is now "heterogeneous", meaning that it supports all XA-compatible resources, not just those from Oracle.

The middle-tier coordinator provides the following features:

- Supports any XA compliant resource
- Supports interpositioning and transaction inflow
- Last resource commit optimization
- Recovery logging

For more information, see "Middle-Tier Two-Phase Commit (2PC) Coordinator" in the *Oracle Containers for J2EE Services Guide*.

Understanding EJB Security Services

You can configure your EJBs to use the J2EE security services that OC4J provides.

For more information, see:

- ["Configuring Security Services"](#) on page 22-1
- *Oracle Application Server 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 (see ["What is a Message-Driven Bean?"](#) on page 1-33) can receive messages for processing.

Using EJB 3.0, message-driven beans can only use a JMS message service provider.

Using EJB 2.1, message-driven beans can use a JMS or non-JMS service provider.

In either case, you can configure a message service provider by specifying message service provider classes or by using a J2EE Connector Architecture (J2CA) adapter.

For more information, see:

- ["What Message Providers Can I use with My MDB?"](#) on page 2-16
- ["Configuring an EJB 3.0 MDB to Use a Non-J2CA Message Service Provider"](#) on page 10-2
- ["Configuring an EJB 2.1 MDB to Use a Non-J2CA Message Service Provider"](#) on page 18-1
- ["Configuring Message Services"](#) on page 23-1

What Message Providers Can I use with My MDB?

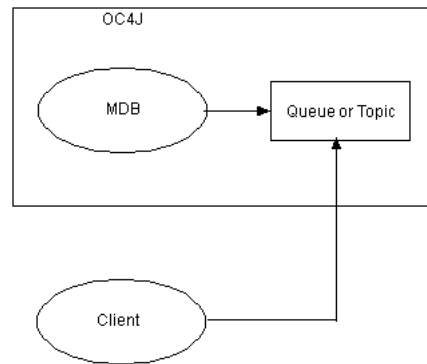
Using OC4J, you can use an MDB with the following types of message provider:

- [Oracle Application Server JMS \(OracleAS JMS\) Provider: File-Based](#)
- [Oracle JMS \(OJMS\) Provider: Advanced Queueing \(AQ\)-Based](#)
- [J2EE Connector Architecture \(J2CA\) Adapter Message Provider](#)

Oracle Application Server JMS (OracleAS JMS) Provider: File-Based

OracleAS JMS is a native Java JMS provider implementation that provides 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 OracleAS JMS queue or topic that is located internally within OC4J. The MDB receives the message directly from OracleAS JMS.

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



If you do not access OracleAS JMS using the Oracle JMS Connector (see ["J2EE Connector Architecture \(J2CA\) Adapter Message Provider"](#) on page 2-18), be aware of the following restrictions:

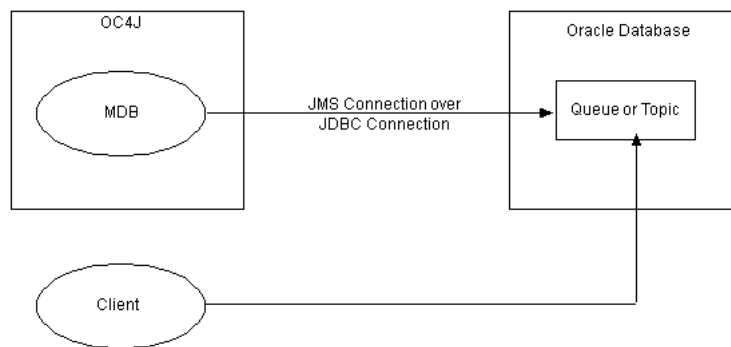
- no support for two-phase commit (2PC)

For more information, see

- ["Configuring an OracleAS JMS Message Service Provider"](#) on page 23-1
- ["Configuring an EJB 3.0 MDB to Use a Non-J2CA Message Service Provider"](#) on page 10-2
- ["Configuring an EJB 2.1 MDB to Use a Non-J2CA Message Service Provider"](#) on page 18-1
- "Java Message Service (JMS)" in the *Oracle Containers for J2EE Services Guide*.

Oracle JMS (OJMS) Provider: Advanced Queueing (AQ)-Based

Oracle JMS (OJMS) 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 OJMS Destination

An MDB uses OJMS 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.

For more information, see:

- ["Configuring an OJMS Message Service Provider"](#) on page 23-3
- ["Configuring an EJB 3.0 MDB to Use a Non-J2CA Message Service Provider"](#) on page 10-2
- ["Configuring an EJB 2.1 MDB to Use a Non-J2CA Message Service Provider"](#) on page 18-1
- *Oracle Streams Advanced Queuing User's Guide and Reference*
- "Java Message Service (JMS)" in the *Oracle Containers for J2EE Services Guide*

J2EE Connector Architecture (J2CA) Adapter Message Provider

OC4J provides a JMS Connector: a generic Java Message Service (JMS) J2CA resource adapter that integrates OC4J with OracleAS JMS and OJMS message service providers, as well as non-Oracle JMS providers as [Table 2–7](#) shows.

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

JMS Provider	Version
OracleAS JMS	all
OJMS	all
IBM WebSphere MQ-based JMS	Server Version 5.3 and 6.0

Table 2–7 (Cont.) Oracle JMS Connector Support for JMS Message Service Providers

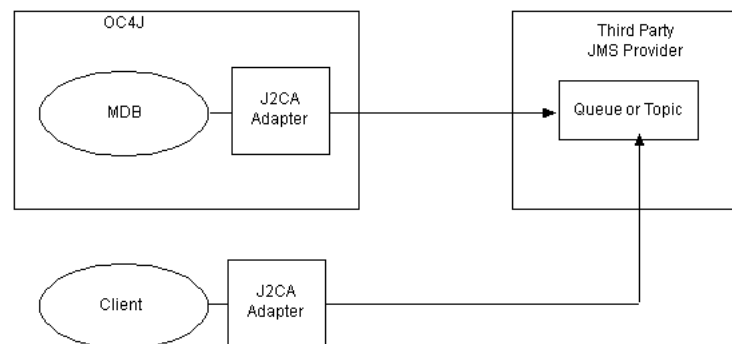
JMS Provider	Version
TIBCO Enterprise for JMS	3.1.0
SonicMQ	6.0

You can use other J2CA-compliant adapters with OC4J to integrate with other types of enterprise information systems (EIS).

Note: Oracle recommends that newer JMS applications be deployed using the J2CA 1.5 Resource Adapter mandated by the J2EE 1.4 standard.

Using a J2CA adapter, you can access other JMS and non-JMS message service providers as [Figure 2–4](#) shows. In this architecture, the client does not access a queue or topic directly. Instead, to send a message to an EIS by way of a J2CA adapter, you:

1. Obtain a `javax.resource.cci.ConnectionFactory`.
If the EIS is a JMS message service provider, there will likely be connection factory choices for queue or topic. For example, the Oracle JMS Connector offers a `QueueConnectionFactory` and a `TopicConnectionFactory`.
2. Use the factory to obtain a `javax.resource.cci.Connection`.
3. Use the connection to obtain a `javax.resource.cci.Interaction`.
4. Configure the interaction and use `Interaction` method `execute` to send the message.

Figure 2–4 Demonstration of an MDB Interacting with a J2CA JMS Destination

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

For more information, see:

- ["Configuring a Message Service Provider Using J2CA"](#) on page 23-7
- ["Configuring an EJB 3.0 MDB to Use a J2CA Message Service Provider"](#) on page 10-3
- ["Configuring an EJB 2.1 MDB to Use a J2CA Message Service Provider"](#) on page 18-2
- "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 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.

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.

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:

- [State Replication](#)
- [Load Balancing](#)
- [Failover](#)
- [Transactions](#)
- [Performance](#)

For more information, see:

- ["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 (see ["Failover"](#) on page 2-22). If you only want to take advantage of load balancing, replication is not required (see ["Load Balancing"](#) on page 2-22).

A replication policy determines the conditions (see ["State Replication Trigger"](#) on page 2-21) under which bean state (see ["State Replication Scope"](#) on page 2-21) is broadcast (see ["State Replication Mode"](#) on page 2-21) to all other OC4J processes in the cluster.

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.

You can configure a replication policy globally for all applications deployed to an OC4J instance or at the application level. You can configure a replication policy for all Web and EJB components and you can configure a replication policy for EJB components only.

For more information on configuring a replication policy for a stateful session bean, see ["Configuring EJB 3.0 and EJB 2.1 Stateful Session Bean Replication Policy"](#) on page 24-1.

State Replication Trigger

You can choose the condition that triggers replication as 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.
- **on request end** — 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 is less performant than the JVM termination replication mode, because the state is sent out more often. However, the guarantee for reliance is higher.
- **on shutdown** — 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 is the most performant option, 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.

State Replication Scope

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).

State Replication Mode

You can configure OC4J EJB application clustering in-memory replication by way of: multicast communication, peer-to-peer communication, or persistence of state data to a database. For more information on configuring replication type, see ["Application Clustering in OC4J"](#) in the *Oracle Containers for J2EE Configuration and Administration Guide*.

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 Load Balancing](#)
- [Static Retrieval Load Balancing](#)
- [DNS Load Balancing](#)

Replication-Based Load Balancing

When you configure a replication policy for a clustered OC4J EJB application (see ["State Replication"](#) on page 2-20), OC4J can automatically select an OC4J instance at random from the pool of OC4J instances in the cluster when the first client request is serviced. You can configure how subsequent requests will be load balanced.

For more information, see ["Configuring Replication-Based Load Balancing"](#) on page 24-3.

Static Retrieval Load Balancing

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 Load Balancing

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 24-4.

Failover

Failover requires that the state of the bean is replicated, so that when the original bean terminates unexpectedly, the request can be transparently forwarded to another OC4J process in the cluster.

For more information, see ["State Replication"](#) on page 2-20.

Transactions

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 2-14.

Performance

The performance for clustering stateful session beans is dependent on the type of replication (see ["State Replication"](#) on page 2-20) and load balancing (see ["Load Balancing"](#) on page 2-22) 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.

Understanding EJB Timer Services

You can set up a timer that invokes an EJB at a specified time, after a specified elapsed time, or at specified intervals. Timers are for use in modeling of application-level processes, not for real-time events.

[Table 2–8](#) summarizes the timers you can use with enterprise JavaBeans.

Table 2–8 EJB Timers

Timer	Stateful Session Bean	Stateless Session Bean	CMP Entity Bean	BMP Entity Bean	Message-Driven Bean
"Understanding J2EE Timer Services" on page 2-23					
EJB 3.0		✓			✓
EJB 2.1		✓	✓	✓	✓
"Understanding OC4J Cron Timer Services" on page 2-23					
EJB 3.0		✓			✓
EJB 2.1	✓	✓	✓	✓	✓

For more information, see ["Configuring Timer Services"](#) on page 25-1.

Understanding J2EE Timer Services

The EJB timer service is a container-managed service that provides methods to allow callbacks to be scheduled for time-based events. The container 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.

The J2EE timer service is implemented by OC4J. An enterprise bean accesses this service by means of dependency injection, through the `EJBContext` interface, or through lookup in the JNDI namespace.

For more information, see:

- ["Configuring an EJB 3.0 EJB with a J2EE Timer"](#) on page 25-1
- ["Configuring an EJB 2.1 EJB with a J2EE Timer"](#) on page 25-2

Understanding OC4J Cron Timer Services

In the UNIX world, you can schedule a timer, known as a cron timer, to execute regularly at specified intervals. Oracle has extended OC4J to support cron timers with EJBs. You can use cron expressions for scheduling timer events with EJBs deployed to OC4J.

For more information, see ["Configuring an EJB with an OC4J Cron Timer"](#) on page 25-3.

Understanding EJB Support in OC4J

This chapter describes:

- [EJB 3.0 Support](#)
- [EJB 2.1 Support](#)
- [Configuration Changes in this Release](#)

EJB 3.0 Support

In this release, OC4J supports a subset of the functionality specified in the EJB 3.0 public review draft (<http://jcp.org/aboutJava/communityprocess/pr/jsr220/index.html>).

You may need to make code changes to your EJB 3.0 OC4J application after the EJB 3.0 specification is finalized and OC4J is updated to full EJB 3.0 compliance.

Oracle cannot guarantee backwards compatibility in all cases. For example, if the meaning or purpose of currently supported annotation changes when the EJB 3.0 specification is finalized, then you must make code changes to your EJB 3.0 OC4J application.

There are no OC4J-proprietary EJB 3.0 annotations. For all OC4J-specific configuration, you must still use the `orion-ejb-jar.xml` file. For more information, see [Appendix A, "XML Reference for orion-ejb-jar.xml Elements"](#).

In this release, OC4J supports the use of `ejb-jar.xml` except for object-relational entity mapping configuration (see ["Implementing an EJB 3.0 Entity"](#) on page 6-1): all such configuration must be done using annotations only. For more information, see ["What is the ejb-jar.xml File?"](#) on page 2-7.

In this release, OC4J does not support resource injection in the web container. For more information, see ["How Do Annotations and Resource Injection Work?"](#) on page 1-7.

This section describes:

- [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?

If you are using EJB 3.0 and annotations, then you must use JDK 5.0.

If you are using EJB 3.0 without annotations, then you may use JDK 1.4. However, in this case, all EJB configuration must be done using the `ejb-jar.xml` and `orion-ejb-jar.xml` deployment descriptor. For an example of how to use EJB 3.0 with JDK 1.4, see the "Using EJB 3.0 EntityManager API in JDK 1.4" in <http://www.oracle.com/technology/tech/java/ejb30.html>.

How do You Define an EJB 3.0 Application?

For entities, an `ejb-jar.xml` file is not used. OC4J assumes that an application deployed without an `ejb-jar.xml` file is an EJB 3.0 application.

For session beans and message-driven beans, an `ejb-jar.xml` file is used. In this case, OC4J checks the `ejb-jar` element version attribute. If it is set to "3.0", OC4J assumes that the application is an EJB 3.0 application.

How does OC4J Manage Persistence in an EJB 3.0 Application?

In an EJB 3.0 application, OC4J delegates persistence operations to an entity manager. In this release, OC4J uses the TopLink persistence framework to implement its entity manager (see "[TopLink Entity Manager](#)" on page 3-2).

TopLink Entity 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 entity manager for EJB 3.0 entities.

For more information about the TopLink, see "What is TopLink?" in the *Oracle TopLink Developer's Guide*.

For EJB 3.0 projects, you configure persistence properties through annotations. OC4J translates these annotations into TopLink configuration.

You can customize this configuration using an `ejb3-toplink-sessions.xml` file. For more information, see:

- "[What is the ejb3-toplink-sessions.xml File?](#)" on page 2-9
- "[Customizing the TopLink Entity Manager](#)" on page 3-2

Customizing the TopLink Entity Manager

At runtime, you can access TopLink API to take advantage of advanced TopLink features.

Typically, you use object-relational annotations (see "[Configuring an EJB 3.0 Entity Container-Managed Relationship Field](#)" on page 7-8) to specify how you want OC4J to store a persistent field in the database and rely on the default TopLink configuration for each such annotation.

Alternatively, you can access the TopLink API in an EJB 3.0 entity application at run time by creating an `ejb3-toplink-sessions.xml` (see "[What is the ejb3-toplink-sessions.xml File?](#)" on page 2-9) and `toplink-ejb-jar.xml` (see "[What is the toplink-ejb-jar.xml File?](#)" on page 2-8) file and packaging them 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 persistence manager, do the following:

1. Create a relational TopLink Workbench project.
"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.
"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.
"Configuring Project Deployment XML Options" in the *Oracle TopLink Developer's Guide*
4. Optionally, configure other TopLink project-level options.
"Configuring a Relational Project" in the *Oracle TopLink Developer's Guide*
5. Configure TopLink relational descriptors for the entity classes you want to customize.
"Creating a Relational Descriptor" in the *Oracle TopLink Developer's Guide*
"Configuring a Relational Descriptor" in the *Oracle TopLink Developer's Guide*
6. Configure TopLink relational mappings for the persistent fields you want to customize.
"Creating a Mapping" in the *Oracle TopLink Developer's Guide*
"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.
"Exporting Deployment XML Information" in the *Oracle TopLink Developer's Guide*
8. Create a TopLink sessions configuration file named `ejb3-toplink-sessions.xml`.
"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.
"Configuring a Primary Mapping Project" in the *Oracle TopLink Developer's Guide*
10. Optionally, configure any other TopLink session-level options.
"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 2-2).

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:

- [What JDK is Required?](#)
- [How do You Define an EJB 2.1 Application?](#)
- [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 Application?

OC4J assumes that the application is an EJB 2.1 application if element `cmp-version` is set to `2.x`.

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 Persistence Manager"](#) on page 3-4).

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 Persistence Manager"](#) on page 3-5), 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 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 CMP entity beans.

For more information about the TopLink persistence manager, see "What is TopLink?" in the *Oracle TopLink Developer's Guide*.

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 2-8). When you migrate an Orion CMP application to TopLink persistence (see "[Migrating to the TopLink Persistence Manager](#)" on page 3-5), the TopLink migration tool automatically creates a TopLink Workbench project for you.

You can customize this configuration at runtime using a TopLink customization class (see "[Customizing the TopLink Persistence Manager](#)" on page 3-5).

Customizing the TopLink Persistence Manager

At runtime, 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 run time creates a new instance of this class and invokes its methods `beforeLoginCustomization` (before the TopLink run time logs into the session) and `afterLoginCustomization` (after the TopLink runtime logs into 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:

- "[Configuring pm-properties](#)" in the *Oracle TopLink Developer's Guide*
- *Oracle TopLink API Reference*

Migrating to the TopLink 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*.

Configuration Changes in this Release

This section lists the following configuration changes introduced in this release:

- [New Package Names for RMI and Application Client Initial Context Factories](#)
- [Unsupported orion-ejb-jar.xml Attributes](#)

For a complete list of new and deprecated OC4J features, see the 10g Release 3 (10.1.3) *Oracle Application Server Release Notes*.

New Package Names for RMI and Application Client Initial Context Factories

In this release, note the new package names for the following initial context factories:

- `oracle.j2ee.rmi.RMIInitialContextFactory`

- `oracle.j2ee.naming.ApplicationClientInitialContextFactory`

For more information, see ["Configuring the Initial Context Factory"](#) on page 19-15.

Unsupported orion-ejb-jar.xml Attributes

The following `orion-ejb-jar.xml` file attributes are unsupported:

- `max-instances-per-pk`
- `min-instances-per-pk`
- `disable-wrapper-cache`
- `instance-cache-timeout`
- `locking-mode="old_pessimistic"`

Note: Do not use these attributes in this release. Doing so will lead to deployment failure.

For more information, see [Appendix A, "XML Reference for orion-ejb-jar.xml Elements"](#).

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 EJB 3.0 Session Bean API"](#)

Implementing an EJB 3.0 Session Bean

This chapter explains how to implement an EJB 3.0 session bean, including:

- [Implementing an EJB 3.0 Stateless Session Bean](#)
- [Implementing an EJB 3.0 Stateful Session Bean](#)

Note: In this release, OC4J supports a subset of the functionality specified in the EJB 3.0 public review draft. You may need to make code changes to your EJB 3.0 OC4J application after the EJB 3.0 specification is finalized and OC4J is updated to full EJB 3.0 compliance. For more information, see ["Understanding EJB Support in OC4J"](#) on page 3-1.

There are no OC4J-proprietary EJB 3.0 annotations. For all OC4J-specific configuration, you must still use the EJB 2.1 `orion-ejb-jar.xml` file

For more information, see:

- ["What is a Session Bean?"](#) on page 1-8
- ["Using EJB 3.0 Session Bean API"](#) 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`.
- Home (`javax.ejb.EJBHome` and `javax.ejb.EJBLocalHome`) and component (`javax.ejb.EJBObject` and `javax.ejb.EJBLocalObject`) business interfaces are not required.

If you choose to use a remote component interface, you can use a plain old Java interface (POJI); it does not need to extend `javax.ejb.EJBObject`.

- 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 ["What is a Stateless Session Bean?"](#) on page 1-8.

Note: You can download an EJB 3.0 stateless session bean code example from:

<http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30slsb/doc/how-to-ejb30-stateless-ejb.html>.

To implement an EJB 3.0 stateless session bean:

1. Optionally, create one or more remote component interfaces and/or local interfaces for your stateless session bean.

You can create a plain old Java interface (POJI) and define it as a remote interface with the `@Remote` annotation or as a local interface using the `@Local` annotation.

You can specify multiple interfaces using the annotation's `value` attribute:

```
@Stateless
@Local (value={Local1.class})
@Remote (value={Remote1.class, Remote2.class})
```

2. 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.

Implement your remote interface, if used.

3. Implement your business methods.

Note: A stateless session bean does not need a remove method.

4. Optionally, define lifecycle callback methods using the appropriate annotations.

You do not need to define lifecycle methods: OC4J provides an implementation for all such methods. Define a method of your stateless session bean class as a lifecycle callback method only if you want to take some action of your own at a particular point in the stateless session bean's lifecycle.

For more information, see "[Configuring a Lifecycle Callback Method for an EJB 3.0 Session Bean](#)" on page 5-2.

5. Complete the configuration of your session bean (see "[Using EJB 3.0 Session Bean API](#)" 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 plain old Java object (POJO); it does not need to implement `javax.ejb.SessionBean`.
- Home (`javax.ejb.EJBHome` and `javax.ejb.EJBLocalHome`) and component (`javax.ejb.EJBObject` and `javax.ejb.EJBLocalObject`) business interfaces are not required.

If you choose to use a remote component interface, you can use a plain old Java interface (POJI); it does not need to extend `javax.ejb.EJBObject`.

- 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 "[What is a Stateful Session Bean?](#)" on page 1-10.

Note: You can download an EJB 3.0 stateful session bean code example from:

<http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30sfsb/doc/how-to-ejb30-stateful-ejb.html>.

To implement an EJB 3.0 stateful session bean:

1. Optionally, create one or more remote component interfaces and/or local interfaces for your stateless session bean.

You can create a plain old Java interface (POJI) and define it as a remote interface with the `@Remote` annotation or as a local interface using the `@Local` annotation.

You can specify multiple interfaces using the annotation's `value` attribute:

```
@Stateless
@Local (value={Local1.class})
@Remote (value={Remote1.class, Remote2.class})
```

2. Create the stateful session bean class.

You can create a plain old Java object (POJO) and define it as a stateful session bean with the `@Stateful` annotation.

Extend your remote interface, if used.

3. Implement your business methods.

To define a method of your stateful session bean class as a remove method using the `@Remove` annotation.

4. Optionally, define lifecycle callback methods using the appropriate annotations.

You do not need to define lifecycle methods: OC4J provides an implementation for all such methods. Define a method of your stateful session bean class as a lifecycle callback method only if you want to take some action of your own at a particular point in the stateful session bean's lifecycle.

For more information, see "[Configuring a Lifecycle Callback Method for an EJB 3.0 Session Bean](#)" on page 5-2.

5. Complete the configuration of your session bean (see "[Using EJB 3.0 Session Bean API](#)" on page 5-1).

Using EJB 3.0 Session Bean API

This chapter describes the various options that you must configure in order to use an EJB 3.0 session bean.

Note: In this release, OC4J supports a subset of the functionality specified in the EJB 3.0 public review draft. You may need to make code changes to your EJB 3.0 OC4J application after the EJB 3.0 specification is finalized and OC4J is updated to full EJB 3.0 compliance. For more information, see ["Understanding EJB Support in OC4J"](#) on page 3-1.

There are no OC4J-proprietary EJB 3.0 annotations. For all OC4J-specific configuration, you must still use the EJB 2.1 `orion-ejb-jar.xml` file.

[Table 5–1](#) 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:

- ["What is a Session Bean?"](#) on page 1-8
- ["Implementing an EJB 3.0 Session Bean"](#) on page 4-1

Table 5–1 Configurable Options for an EJB 3.0 Session Bean

Options	Type
"Configuring Passivation" on page 5-1	Advanced
"Configuring Passivation Criteria" on page 5-2	Advanced
"Configuring Passivation Location" on page 5-2	Advanced
"Configuring Bean Instance Pool Size" on page 31-3	Basic
"Configuring Bean Instance Pool Timeouts for Session Beans" on page 31-4	Advanced
"Configuring a Transaction Timeout for a Session Bean" on page 21-2	Advanced
"Configuring a Lifecycle Callback Method for an EJB 3.0 Session Bean" on page 5-2	Basic
"Configuring an Interceptor on an EJB 3.0 Session Bean" on page 5-3	Advanced

Configuring Passivation

Passivation is an Oracle-specific option that you configure using the EJB 2.1 `orion-ejb-jar.xml` file.

For more information, see ["Configuring Passivation"](#) on page 12-1.

Configuring Passivation Criteria

Passivation criteria is an Oracle-specific option that you configure using the EJB 2.1 `orion-ejb-jar.xml` file.

For more information, see ["Configuring Passivation Criteria"](#) on page 12-2.

Configuring Passivation Location

Passivation location is an Oracle-specific option that you configure using the EJB 2.1 `orion-ejb-jar.xml` file.

For more information, see ["Configuring Passivation Location"](#) on page 12-3.

Configuring a Lifecycle Callback Method for an EJB 3.0 Session Bean

You can specify an EJB 3.0 session bean class method as a callback method for any of the following lifecycle events (see ["Using Annotations"](#) on page 5-2):

- **Post-construct:** a method called before the first business method invocation on the session bean and after OC4J performs any dependency injection.
- **Pre-destroy:** a method called before OC4J removes the session bean. Typically, you use this method to release any resources that your session bean is holding.
- **Pre-passivate:** a method called before OC4J passivates the session bean. Applicable to stateful session beans only. Use only if you need to close resources prior to passivation.
- **Post-activate:** a method called immediately after OC4J reactivates a formerly passivated session bean. Applicable to stateful session beans only. Use only if you need to reopen resources after reactivation.

Note: Do not specify pre-passivate or post-activate lifecycle callback methods on a stateless session bean.

The session bean class lifecycle callback method must have the following signature:

```
public void <MethodName>()
```

For more information, see ["Callback Methods"](#) on page 1-6.

Using Annotations

You can specify an EJB 3.0 session bean class method as a lifecycle callback method using any of the following annotations:

- `@PostConstruct`
- `@PreDestroy`
- `@PrePassivate` (stateful session beans only)
- `@PostActivate` (stateful session beans only)

[Example 5-1](#) shows how to use the `@PostConstruct` annotation to specify EJB 3.0 stateful session bean class method `initialize` as a lifecycle callback method.

Example 5-1 @PostConstruct

```

@Stateful
public class CartBean implements Cart
{
    private ArrayList items;

    @PostConstruct
    public void initialize()
    {
        items = new ArrayList();
    }
    ...
}

```

Configuring an Interceptor on an EJB 3.0 Session Bean

You can designate one non-business method as the interceptor method for a stateless or stateful session bean (see ["Using Annotations"](#) on page 5-3). The method must have a signature of:

```
public Object <MethodName>(InvocationContext) throws Exception
```

For more information, see ["Understanding EJB 3.0 Interceptors"](#) on page 2-11.

Using Annotations

[Example 5-2](#) 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 5-2 @AroundInvoke in an EJB 3.0 Session Bean

```

@Stateless
public class HelloWorldBean implements HelloWorld
{
    public void sayHello()
    {
        System.out.println("Hello!");
    }

    @AroundInvoke
    public 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();
    }
}

```


Part III

EJB 3.0 Entities

This part provides procedural information on implementing and configuring EJB 3.0 entity beans and entity bean queries. For conceptual information, see [Part I, "EJB Overview"](#).

This part contains the following chapters:

- [Chapter 6, "Implementing an EJB 3.0 Entity"](#)
- [Chapter 7, "Using EJB 3.0 Persistence API"](#)
- [Chapter 8, "Using EJB 3.0 Query API"](#)

Implementing an EJB 3.0 Entity

This chapter explains how to implement an EJB 3.0 entity.

Note: In this release, OC4J supports a subset of the functionality specified in the EJB 3.0 public review draft. You may need to make code changes to your EJB 3.0 OC4J application after the EJB 3.0 specification is finalized and OC4J is updated to full EJB 3.0 compliance. For more information, see ["Understanding EJB Support in OC4J"](#) on page 3-1.

There are no OC4J-proprietary EJB 3.0 annotations. For all OC4J-specific configuration, you must still use the EJB 2.1 `orion-ejb-jar.xml` file.

For more information, see:

- ["What is an EJB 3.0 Entity?"](#) on page 1-14
- ["Using EJB 3.0 Persistence API"](#) on page 7-1

Implementing an EJB 3.0 Entity

EJB 3.0 greatly simplifies the development of EJBs, 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.EntityBean`.
- Home (`javax.ejb.EJBHome`) and component (`javax.ejb.EJBObject`) interfaces are not required.

If you choose to use a remote component interface, you can use a plain old Java interface (POJI); it does not need to extend `javax.ejb.EJBObject`.

- 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 an EJB 3.0 Entity?"](#) on page 1-14.

Note: You can download an EJB 3.0 CMP entity code example from:
<http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30entity/doc/how-to-ejb30-entity-ejb.html>.

To implement an EJB 3.0 entity:

1. Create the entity bean class.

You can create a plain old Java object (POJO) and define it as a CMP entity bean with the `@Entity` annotation.

All data members are by default considered container-managed persistence 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 7-5.

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 an EJB 3.0 Entity Primary Key](#)" on page 7-2.

4. Define container-managed relationships using the appropriate object-relational mapping annotations, such as `@OneToOne`.

For more information, see "[Configuring an EJB 3.0 Entity Container-Managed Relationship Field](#)" on page 7-8.

5. Optionally, define finders and queries using the `@NamedQuery` annotation.

At runtime, you can use the predefined finders (see "[Predefined TopLink Finders](#)" on page 1-30) and default finders (see "[Default TopLink Finders](#)" on page 1-31) that the TopLink persistence manager provides.

For more information, see "[Using EJB 3.0 Query API](#)" on page 8-1.

6. Optionally, define lifecycle callback methods using the appropriate annotations.

You do not need to define lifecycle methods: OC4J provides an implementation for all such methods. Define a method of your entity bean class as a lifecycle callback method only if you want to take some action of your own at a particular point in the entity bean's lifecycle.

For more information, see "[Configuring a Lifecycle Callback Method for an EJB 3.0 Entity](#)" on page 7-14.

7. Complete the configuration of your entity bean (see "[Using EJB 3.0 Persistence API](#)" on page 7-1).

Using EJB 3.0 Persistence API

This chapter describes the various options that you can configure in order to use an EJB 3.0 entity.

Note: In this release, OC4J supports a subset of the functionality specified in the EJB 3.0 public review draft. You may need to make code changes to your EJB 3.0 OC4J application after the EJB 3.0 specification is finalized and OC4J is updated to full EJB 3.0 compliance. For more information, see ["Understanding EJB Support in OC4J"](#) on page 3-1.

There are no OC4J-proprietary EJB 3.0 annotations. For all OC4J-specific configuration, you must still use the EJB 2.1 `orion-ejb-jar.xml` file.

[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:

- ["What is an EJB 3.0 Entity?"](#) on page 1-14
- ["Implementing an EJB 3.0 Entity"](#) on page 6-1

Table 7–1 Configurable Options for an EJB 3.0 Entity

Options	Type
"Configuring an EJB 3.0 Entity Primary Key" on page 7-2	Basic
"Configuring Table and Column Information" on page 7-5	Basic
"Configuring an EJB 3.0 Entity Container-Managed Relationship Field" on page 7-8	Basic
"Configuring a Basic Mapping" on page 7-8	Basic
"Configuring a Large Object Mapping" on page 7-9	Advanced
"Configuring a Serialized Object Mapping" on page 7-9	Advanced
"Configuring a One-to-One Mapping" on page 7-10	Basic
"Configuring a Many-to-One Mapping" on page 7-10	Basic
"Configuring a One-to-Many Mapping" on page 7-11	Basic
"Configuring a Many-to-Many Mapping" on page 7-11	Basic
"Configuring an Aggregate Mapping" on page 7-12	Advanced
"Configuring Optimistic Lock Version Field" on page 7-14	Advanced
"Using EJB 3.0 Query API" on page 8-1	Basic

Table 7–1 (Cont.) Configurable Options for an EJB 3.0 Entity

Options	Type
"Configuring Inheritance for an EJB 3.0 Entity" on page 7-16	Advanced
"Configuring Lazy Loading on Finder Methods" on page 7-14	Basic
"Configuring Bean Instance Pool Size" on page 31-3	Basic
"Configuring Bean Instance Pool Timeouts for Entity Beans" on page 31-4	Advanced
"Configuring a Lifecycle Callback Method for an EJB 3.0 Entity" on page 7-14	Advanced

Configuring an EJB 3.0 Entity Primary Key

Every EJB 3.0 entity must have a primary key field (see ["Configuring an EJB 3.0 Entity Primary Key Field" on page 7-2](#)).

You can specify a primary key as a single primitive or JDK object type.

You can either assign primary key values yourself, or, more typically, you can associate a primary key field with a primary key value generator (see ["Configuring EJB 3.0 Entity Automatic Primary Key Generation" on page 7-3](#)).

Configuring an EJB 3.0 Entity Primary Key Field

You must specify one entity field as the primary key. You can specify a primary key as a single primitive or JDK object type or as a composite primary key class made up of one or more primitive or JDK object types

Typically, you associate the primary key field with a primary key value generator (see ["Configuring EJB 3.0 Entity Automatic Primary Key Generation" on page 7-3](#)).

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#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 EJB 3.0 Entity Automatic Primary Key Generation" on page 7-3](#)).

Example 7–1 @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 EJB 3.0 Entity Automatic Primary Key Generation

Typically, you associate a primary key field (see "[Configuring an EJB 3.0 Entity 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 EJB 3.0 Entity Primary Key Value Generators

Type	Description	For more information, see ...
Generated Id Table	A database table 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 <i>Oracle TopLink Developer's Guide</i>
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 <i>Oracle TopLink Developer's Guide</i>
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 <i>Oracle TopLink Developer's Guide</i> "Native Sequencing With a Non-Oracle Database Platform" in the <i>Oracle TopLink Developer's Guide</i>

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 7-2](#) shows how to use the `@GeneratedIdTable` annotation to specify a primary key value generator based on a database table. When a new instance of `Employee` is created, a new value for entity field `id` is obtained from `EMPLOYEE_GENERATOR_TABLE`.

Example 7-2 @GeneratedIdTable

```
@Entity
@Table(name="EJB_EMPLOYEE")
@GeneratedIdTable(
    name="EMPLOYEE_GENERATOR_TABLE",
    table=@Table(name="EJB_EMPLOYEE_SEQ"),
    pkColumnName="SEQ_NAME", valueColumnName="SEQ_COUNT")
```

```
)
public class Employee implements Serializable
{
    ...
    @Id(generate=TABLE, generator="EMPLOYEE_GENERATOR_TABLE")
    @Column(name="EMPLOYEE_ID", primaryKey=true)
    public Integer getId()
    {
        return id;
    }
    ...
}
```

[Example 7-3](#) shows how to use the `@TableGenerator` annotation to specify a primary key value generator based on a database table. When a new instance of `Employee` is created, a new value for entity field `id` is obtained from `EMPLOYEE_GENERATOR_TABLE`. You must set the `@Id` annotation attribute `generate` to `TABLE` in this case.

Example 7-3 @TableGenerator

```
@Entity
@Table(name="EJB_ADDRESS")
public class Address implements Serializable
{
    ...
    @Id(generate=TABLE, generator="ADDRESS_TABLE_GENERATOR")
    @TableGenerator(
        name="ADDRESS_TABLE_GENERATOR",
        tableName="EMPLOYEE_GENERATOR_TABLE",
        pkColumnName="ADDRESS_SEQ"
    )
    @Column(name="ADDRESS_ID")
    public Integer getId()
    {
        return id;
    }
    ...
}
```

[Example 7-3](#) shows how to use the `@SequenceGenerator` annotation to specify a primary key value generator based on a sequence object provided by the database. When a new instance of `Employee` is created, a new value for entity field `id` is obtained from database sequence object `ADDRESS_SEQ`. You must set the `@Id` annotation attribute `generate` to `SEQUENCE` in this case.

Example 7-4 @SequenceGenerator

```
@Entity
@Table(name="EJB_ADDRESS")
public class Address implements Serializable
{
    ...
    @Id(generate=SEQUENCE, generator="ADDRESS_TABLE_GENERATOR")
    @SequenceGenerator(
        name="ADDRESS_TABLE_GENERATOR",
        sequenceName="ADDRESS_SEQ"
    )
    @Column(name="ADDRESS_ID")
    public Integer getId()
    {
        return id;
    }
    ...
}
```

```
}
```

Configuring Table and Column Information

You can define the characteristics of the database table into which the TopLink persistence manager persists your entity, including:

- [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 an EJB 3.0 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 persistence manager persists your entity: in particular, it is the table that stores the entity's primary key (see "[Configuring an EJB 3.0 Entity Primary Key](#)" on page 7-2). Optionally, you can also specify one or more secondary tables (see "[Configuring a Secondary Table](#)" on page 7-5) if the entity's persistent data is stored across multiple tables.

You define the primary table at the entity class level.

Using Annotations

[Example 7-5](#) shows how to use the `@Table` annotation to define the primary table for the `Employee` class. The TopLink persistence manager will persist instances of this entity to a table named `EJB_EMPLOYEE`.

Example 7-5 `@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 7-5) 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 Join Column"](#) on page 7-7) for persistent fields that are stored in that table. This allows you to distribute the persistent fields of an entity across multiple tables.

Using Annotations

[Example 7-6](#) 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 7-6 `@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 7-5) into which the TopLink persistence 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 7-5), you can specify the secondary table name in the column definition. This allows you to distribute the persistent fields of an entity across multiple tables.

Using Annotations

[Example 7-7](#) shows how to use the `@Column` annotation to specify column `F_NAME` in the primary table for field `firstName`.

Example 7-7 `@Column` for the Primary Table

```
@Column(name="F_NAME")
public String getFirstName()
{
    return firstName;
}
```

[Example 7-8](#) shows how to use the `@Column` annotation to specify column `SALARY` in secondary table `EMP_SALARY` for field `salary`.

Example 7-8 `@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 a:

- secondary table (see [Example 7-9](#))
- one-to-one mapping (see [Example 7-10](#))
- many-to-one mapping (see [Example 7-11](#))
- one-to-many mapping (see [Example 7-12](#))

Using Annotations

[Example 7-9](#) 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 7-5.

Example 7-9 @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 7-10](#) shows how to use the `@JoinColumn` annotation to specify a join column with a one-to-one mapping. For more information, see ["Configuring a One-to-One Mapping"](#) on page 7-10.

Example 7-10 @JoinColumn with a One-to-One Mapping

```
@OneToOne(cascade=ALL, fetch=LAZY)
@JoinColumn(name="ADDR_ID")
public Address getAddress()
{
    return address;
}
```

[Example 7-11](#) shows how to use the `@JoinColumn` annotation to specify a join column with a many-to-one mapping. For more information, see ["Configuring a Many-to-One Mapping"](#) on page 7-10.

Example 7-11 @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 7-12](#) shows how to use the `@JoinColumn` annotation to specify a join column with a one-to-many mapping. For more information, see ["Configuring a One-to-Many Mapping"](#) on page 7-11.

Example 7-12 @JoinColumn with a One-to-Many Mapping

```
@OneToMany(cascade=PERSIST)
@JoinColumn(name="MANAGER_ID", referencedColumnName="EMP_ID")
public Collection getManagedEmployees()
{
    return managedEmployees;
}
```

Configuring an EJB 3.0 Entity Container-Managed Relationship Field

In an EJB 3.0 entity, you define container-managed relationship (CMR) fields (see ["What are Container-Managed Relationship Fields?"](#) on page 1-20) as follows:

- ["Configuring a Basic Mapping"](#) on page 7-8
- ["Configuring a Large Object Mapping"](#) on page 7-9
- ["Configuring a Serialized Object Mapping"](#) on page 7-9
- ["Configuring a One-to-One Mapping"](#) on page 7-10
- ["Configuring a Many-to-One Mapping"](#) on page 7-10
- ["Configuring a One-to-Many Mapping"](#) on page 7-11
- ["Configuring a Many-to-Many Mapping"](#) on page 7-11

Note: You can download an EJB 3.0 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.

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 7-13](#) shows how to use the `@Basic` annotation to specify a basic mapping for field `firstName`.

Example 7-13 @Basic

```
@Basic(fetch=EAGER)
@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:

- "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 7-14](#) shows how to use the `@Lob` annotation to specify a large object mapping for field `image`.

Example 7-14 `@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:

- "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 7-15](#) shows how to use the `@Serialized` annotation to specify a serialized object mapping for field `picture`.

Example 7-15 `@Serialized`

```
@Serialized(fetch=EAGER)
@Column(name="PICTURE")
```

```
public Byte[] getPicture()
{
    return picture;
}
```

Configuring a One-to-One Mapping

Use a 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 a 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 7-16](#) shows how to use the `@OneToOne` annotation to specify a one-to-one mapping for field address.

Example 7-16 `@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 7-17](#) shows how to use the `@ManyToOne` annotation to specify a many-to-one mapping for field `manager`.

Example 7-17 `@ManyToOne`

```
@ManyToOne(cascade=PERSIST, fetch=LAZY)
@JoinColumn(name="MANAGER_ID", referencedColumnName="EMP_ID")
public Employee getManager()
{
    return manager;
}
```

Configuring a One-to-Many Mapping

Use a 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 7-18](#) shows how to use the `@OneToMany` annotation to specify a one-to-many mapping for field `managedEmployees`.

Example 7-18 `@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 7–19](#) shows how to use the `@ManyToMany` annotation to specify a many-to-many mapping for field projects and how to use the `@AssociationTable` annotation to specify an association table.

Example 7–19 `@ManyToMany`

```
@ManyToMany(cascade=PERSIST)
@AssociationTable(
    table=@Table(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 allows you to 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 7-6) 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 7–20](#) shows how to use the `@Embedded` annotation to specify an aggregate mapping for field period. This field contains an instance of `EmploymentPeriod`.

[Example 7–21](#) 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 7-5) to associate this class with the owning entity's table.

Example 7-20 @Embedded

```
@Entity
@Table(name="EJB_EMPLOYEE")
public class Employee implements Serializable
{
    ...
    @Embedded
    public EmploymentPeriod getPeriod()
    {
        return period;
    }
    ...
}
```

Example 7-21 @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 7-22](#)) to override the column definitions made in the owned entity (see [Example 7-23](#)).

Example 7-22 @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 7-23 @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;
    ...
}
```

```
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 a CMP Entity Bean Instance"](#) on page 29-13) 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 7-24](#) shows how to use the `@Version` annotation to define an optimistic version locking policy using column `VERSION`.

Example 7-24 `@Version`

```
@Version  
@Column(name="VERSION")  
public int getVersion()  
{  
    return version;  
}
```

Configuring Lazy Loading on Finder Methods

Lazy loading 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 14-11.

Configuring a Lifecycle Callback Method for an EJB 3.0 Entity

You can specify an EJB 3.0 entity class method as a callback method for any of the following lifecycle events:

- Pre-persist: a method called on an object when that object has the create operation applied to it.
- Post-persist: a method called on an object that has just been inserted into the database. You can use this method to notify any of the object's dependents or to update information not accessible until the object has been inserted.
- Pre-remove: a method called on an object when that object has the remove operation applied to it.
- Post-remove: a method called on an object that has just been deleted from the database. You can use this method to notify or remove any of the object's dependents.

- Pre-update: a method called when an object's row is about to be updated.

The TopLink persistence manager 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 whether or not an actual change was required.

You can use this method to modify the row before insert.

- Post-update: a method called on an object that has just been updated into the database.

Use this callback to update any dependent objects.

The TopLink persistence manager 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.

- Post-load: a method called on an object that has just been built from the database. This event can be used to correctly initialize an object's non-persistent attributes or to perform complex optimizations or mappings. This event is called whenever an object is built.

The entity class method must have the following signature:

```
public int <MethodName>()
```

For more information, see:

- ["Callback Methods"](#) 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 lifecycle 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 an EJB 3.0 entity class method as a lifecycle callback method using any of the following annotations:

- `@PrePersist`
- `@PostPersist`
- `@PreRemove`
- `@PostRemove`
- `@PreUpdate`
- `@PostUpdate`
- `@PostLoad`

[Example 7-25](#) shows how to use the `@PrePersist` annotation to specify EJB 3.0 entity class method `initialize` as a lifecycle callback method.

Example 7-25 @PrePersist

```
@PrePersist
public int initialize()
{
    ...
}
```

Configuring Inheritance for an EJB 3.0 Entity

OC4J supports the following inheritance strategies for mapping a class or class hierarchy to a relational database schema:

- [Joined Subclass](#)
- [Single Table per Class Hierarchy](#)

You can configure either approach using annotations (see "[Using Annotations](#)" on page 7-17).

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 separate 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 7-17.

Single Table per 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 7-18.

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 7-16): [Example 7-26](#) shows how to use the `@Inheritance` annotation in the base class `Project`. [Example 7-27](#) and [Example 7-28](#) 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 7-27](#), 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 7-26 @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 7-27 @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 7–28 @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 per class hierarchy approach (see ["Single Table per Class Hierarchy"](#) on page 7-16): [Example 7–26](#) shows how to use the `@Inheritance` annotation in the base class `Project`. [Example 7–27](#) and [Example 7–28](#) 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 7–29 @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 7–30 @Inheritance: Derived Class LargeProject in Single Table Inheritance

```
@Entity
@Inheritance(discriminatorValue="L")
public class LargeProject extends Project
{
    ...
}
```

Example 7–31 @Inheritance: Derived Class SmallProject in Single Table Inheritance

```
@Entity
@Inheritance(discriminatorValue="S")
public class SmallProject extends Project
{
    ...
}
```

Using EJB 3.0 Query API

This section describes how to create pre-defined, static queries that you can access at runtime, including:

- [Implementing an EJB 3.0 Named Query](#)
- [Implementing an EJB 3.0 Dynamic Query](#)

Note: In this release, OC4J supports a subset of the functionality specified in the EJB 3.0 public review draft. You may need to make code changes to your EJB 3.0 OC4J application after the EJB 3.0 specification is finalized and OC4J is updated to full EJB 3.0 compliance. For more information, see "[Understanding EJB Support in OC4J](#)" on page 3-1.

There are no OC4J-proprietary EJB 3.0 annotations. For all OC4J-specific configuration, you must still use the EJB 2.1 `orion-ejb-jar.xml` file.

For more information, see "[How Do You Query for an EJB 3.0 Entity?](#)" on page 1-16.

Implementing an EJB 3.0 Named Query

A named query is a pre-defined query that you create and associate with a CMP entity (see "[Using Annotations](#)" on page 8-1). At deployment time, OC4J stores named queries on the `EntityManager` (see "[How Do You Query for an EJB 3.0 Entity?](#)" on page 1-16).

At runtime, you can use the `EntityManager` to acquire, configure, and execute a named query (see "[Querying for an EJB 3.0 Entity Using the EntityManager](#)" on page 29-8).

Note: In this release, OC4J does not support EJB 3.0 named queries using native SQL. For more information, see "[Understanding Native SQL Query Syntax](#)" on page 1-29.

Using Annotations

[Example 8-1](#) shows how to use the `@NamedQuery` annotation to define an EJB QL query that you can acquire by name `findAllEmployeesByFirstName` at runtime 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 an EJB QL 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 an EJB 3.0 Entity Using the EntityManager"](#) on page 29-8.

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 an EJB 3.0 Dynamic Query

Using `EntityManager` methods `createQuery` and `createNativeQuery(String sqlString, Class resultType)`, you can create a `Query` object dynamically at runtime (see ["Using Java"](#) on page 8-3).

Using the `Query` methods `getResultList`, `getSingleResult`, or `executeUpdate` you can execute the query (see ["Executing a Query"](#) on page 29-11).

For more information, see:

- ["Acquiring an EntityManager"](#) on page 29-5
- ["Creating a Dynamic EJB QL Query with the Entity Manager"](#) on page 29-9
- ["Creating a Dynamic TopLink Expression Query with the EntityManager"](#) on page 29-9
- ["Creating a Dynamic Native SQL Query with the EntityManager"](#) on page 29-10
- ["Configuring Query Hints"](#) on page 29-10
- ["Executing a Query"](#) on page 29-11

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 we use Query method `getResultList`.

Example 8-4 Implementing and Executing a Dynamic Query

```
Query queryEmployees = entityManager.createQuery(
    "SELECT OBJECT(emp) FROM Employee emp WHERE emp.firstName = :firstname"
);

queryEmployeeByFirstName.setParameter("firstName", "Joan");

Collection employees = queryEmployees.getResultList();
```


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:

- [Chapter 9, "Implementing an EJB 3.0 MDB"](#)
- [Chapter 10, "Using EJB 3.0 MDB API"](#)

Implementing an EJB 3.0 MDB

This chapter explains how to implement an EJB 3.0 message-driven bean.

Note: In this release, OC4J supports a subset of the functionality specified in the EJB 3.0 public review draft. You may need to make code changes to your EJB 3.0 OC4J application after the EJB 3.0 specification is finalized and OC4J is updated to full EJB 3.0 compliance. For more information, see ["Understanding EJB Support in OC4J"](#) on page 3-1.

There are no OC4J-proprietary EJB 3.0 annotations. For all OC4J-specific configuration, you must still use the EJB 2.1 `orion-ejb-jar.xml` file.

For more information, see:

- ["What is a Message-Driven Bean?"](#) on page 1-33
- ["Using EJB 3.0 MDB API"](#) on page 10-1

Implementing an EJB 3.0 MDB

EJB 3.0 greatly simplifies the development of EJBs, 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.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 1-33.

Note: You can download an EJB 3.0 message-driven bean code example from:
<http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30mdb/doc/how-to-ejb30-mdb.html>
.

To implement an EJB 3.0 message-driven bean:

1. Configure your message service provider.

For more information, see:

- ["What Message Providers Can I use with My MDB?"](#) on page 2-16
- ["Configuring Message Services"](#) on page 23-1

2. Create the message-driven bean class.

You can create a plain old Java object (POJO) and define it as a message-driven bean with the `@MessageDriven` annotation.

3. Configure message service provider information:

You can define this information with the `@ActivationConfigProperty` annotation.

For more information, see:

- ["Configuring an EJB 3.0 MDB to Use a Non-J2CA Message Service Provider"](#) on page 10-2
- ["Configuring an EJB 3.0 MDB to Use a J2CA Message Service Provider"](#) 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:

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)
```

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 signature:

```
public void ejbTimeout(javax.ejb.Timer timer)
```

7. Optionally, define lifecycle callback methods using the appropriate annotations.

You do not need to define lifecycle methods: OC4J provides an implementation for all such methods. Define a method of your message-driven bean class as a lifecycle callback method only if you want to take some action of your own at a particular point in the message-driven bean's lifecycle.

For more information, see ["Configuring a Lifecycle Callback Method for an EJB 3.0 MDB"](#) on page 10-6

8. Complete the configuration of your message-driven bean (see ["Using EJB 3.0 MDB API"](#) on page 10-1).

Using EJB 3.0 MDB API

This chapter describes the various options that you must configure in order to use an EJB 3.0 message-driven bean.

Note: In this release, OC4J supports a subset of the functionality specified in the EJB 3.0 public review draft. You may need to make code changes to your EJB 3.0 OC4J application after the EJB 3.0 specification is finalized and OC4J is updated to full EJB 3.0 compliance. For more information, see ["Understanding EJB Support in OC4J"](#) on page 3-1.

There are no OC4J-proprietary EJB 3.0 annotations. For all OC4J-specific configuration, you must still use the EJB 2.1 `orion-ejb-jar.xml` file.

[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:

- ["What is a Message-Driven Bean?"](#) on page 1-33
- ["Implementing an EJB 3.0 MDB"](#) 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 Use a Non-J2CA Message Service Provider" on page 10-2	Basic
"Configuring an EJB 3.0 MDB to Use a J2CA Message Service Provider" on page 10-3	Basic
"Configuring an MDB for Fast Undeploy on Windows" on page 18-4	Advanced
"Configuring an MDB for Oracle RAC Failover" on page 18-5	Advanced
"Configuring Bean Instance Pool Size" on page 31-3	Basic
"Configuring a Transaction Timeout for a Message-Driven Bean" on page 21-2	Advanced
"Configuring Listener Threads" on page 10-5	Advanced
"Configuring Maximum Delivery Count" on page 10-5	Advanced
Configuring Dequeue Retry Count and Interval on page 10-5	Advanced
"Configuring an Interceptor on an EJB 3.0 MDB Message Listener Method" on page 10-5	Advanced
"Configuring a Lifecycle Callback Method for an EJB 3.0 MDB" on page 10-6	Basic

Configuring an EJB 3.0 MDB to Use a Non-J2CA Message Service Provider

You can configure an EJB 3.0 MDB to use a non-J2CA message service provider using annotations (see ["Using Annotations"](#) on page 10-2) or deployment XML (see ["Using Deployment XML"](#) on page 10-3).

For more information, see:

- ["Oracle Application Server JMS \(OracleAS JMS\) Provider: File-Based"](#)
- ["Oracle JMS \(OJMS\) Provider: Advanced Queueing \(AQ\)-Based"](#)

Using Annotations

You associate an EJB 3.0 MDB with a message service provider using the `@MessageDriven` annotation `activationConfig` attribute.

[Example 10-3](#) shows how to configure a message-driven bean to use a non-J2CA JMS message service provider. It assumes that connection factory `jms/MyQCF` and queue `jms/MyQueue` are defined in the `jms.xml` file. For more information on configuring a non-J2CA message service provider, see ["Configuring an OracleAS JMS Message Service Provider"](#) on page 23-1 or ["Configuring an OJMS Message Service Provider"](#) on page 23-3.

Example 10-1 @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 `@MessageDriven` annotation `activationConfig` attribute contains a list of `@ActivationConfigProperty` annotations whose `propertyName` and `propertyValue` attributes you use to specify activation configuration properties. [Table 10-2](#) lists the mandatory and commonly used activation configuration properties you must set.

Table 10–2 Mandatory @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	Specify the fully qualified class name of the destination type for your message service provider. For a JMS MDB, either <code>javax.jms.Queue</code> or <code>javax.jms.Topic</code> .
messageSelector	Specify the boolean expression of message properties that match the type of message your MDB should receive.

For a complete list of all activation configuration properties, download and unzip one of the `how-to-gjra-with-xxx.zip` files from

http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html, where `xxx` 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 "JMS Resource Adapter" in the *Oracle Containers for J2EE Services Guide*.

The actual names you use depend on your message service provider installation. For more information, see:

- "OracleAS JMS Destination and Connection Factory Names" on page 23-2
- "OJMS Destination and Connection Factory Names" on page 23-3

Using Deployment XML

You can associate an EJB 3.0 MDB with a non-J2CA message service provider by configuring the `ejb-jar.xml` file `message-driven` element with an `activation-config` element and `orion-ejb-jar.xml` file `message-driven-deployment` element with an `activation-config` element as you would for an EJB 2.1 MDB. Configuration in the `orion-ejb-jar.xml` file will override annotations and `ejb-jar.xml` file configuration.

For more information, see "Using Deployment XML" on page 18-1.

Configuring an EJB 3.0 MDB to Use a J2CA Message Service Provider

You can configure an EJB 3.0 MDB to use a J2CA message service provider using annotations (see "Using Annotations" on page 10-3).

For more information, see "J2EE Connector Architecture (J2CA) Adapter Message Provider" on page 2-18.

Using Annotations

You associate an EJB 3.0 MDB with a J2CA message service provider using the `@MessageDriven` annotation `activationConfig` attribute and `@MessageDrivenDeployment` annotation `resourceAdapter` attribute. You must use `@MessageDrivenDeployment` annotation to specify the resource adapter that your message-driven bean uses.

[Example 10–3](#) shows how to configure a message-driven bean to use the Oracle JMS resource adapter named "OracleASjms". It assumes that connection factory

OracleASjms/MyQCF is defined in `oc4j-ra.xml` file and destination name OracleASjms/MyQueue is defined in `oc4j-connectors.xml` file. For more information on configuring a J2CA message service provider, see ["Configuring a Message Service Provider Using J2CA"](#) on page 23-7.

Example 10–2 @MessageDriven and @MessageDrivenDeployment Annotation for a J2CA Message Service Provider

```
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.Message;
import javax.jms.MessageListener;

// Oracle deployment annotation for MDB
import oracle.j2ee.ejb.MessageDrivenDeployment;

@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 `@MessageDriven` annotation `activationConfig` attribute contains a list of `@ActivationConfigProperty` annotations whose `propertyName` and `propertyValue` attributes you use to specify activation configuration properties. [Table 10–2](#) lists the mandatory and commonly used activation configuration properties you must set.

Table 10–3 Mandatory @ActivationConfigProperty Attributes

Property Name	Value
<code>connectionFactoryJndiName</code>	The JNDI name of the message service provider connection factory. You define this name when you configure your message service provider.
<code>destinationName</code>	The JNDI name of the message service provider destination name. You define this name when you configure your message service provider.
<code>destinationType</code>	Specify the fully qualified class name of the destination type for your message service provider. For a JMS MDB, either <code>javax.jms.Queue</code> or <code>javax.jms.Topic</code> .
<code>messageSelector</code>	Specify the boolean expression of message properties that match the type of message your MDB should receive.

For a complete list of all activation configuration properties, download and unzip one of the `how-to-gjra-with-xxx.zip` files from

http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html, where xxx 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 "JMS Resource Adapter" in the *Oracle Containers for J2EE Services Guide*.

You may also set the optional attributes that [Table A-4](#) 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 23-8.

Using Deployment XML

You can associate an EJB 3.0 MDB with a J2CA message service provider using the `orion-ejb-jar.xml` file by configuring the `message-driven-deployment` element with an `activation-config` element and setting the `message-driven-deployment` element `resource-adapter` attribute as you would for an EJB 2.1 MDB. Configuration in the `orion-ejb-jar.xml` file will override annotations, if present.

For more information, see "[Using Deployment XML](#)" on page 18-3.

Configuring Listener Threads

The number of listener threads is an Oracle-specific option that you configure using the EJB 2.1 `orion-ejb-jar.xml` file.

For more information, see "[Configuring Listener Threads](#)" on page 18-6.

Configuring Maximum Delivery Count

The maximum delivery count is an Oracle-specific option that you configure using the EJB 2.1 `orion-ejb-jar.xml` file.

For more information, see "[Configuring Maximum Delivery Count](#)" on page 18-7.

Configuring Dequeue Retry Count and Interval

The dequeue retry count and dequeue retry interval are Oracle-specific options that you configure using the EJB 2.1 `orion-ejb-jar.xml` file.

For more information, see "[Configuring Dequeue Retry Count and Interval](#)" on page 18-8.

Configuring an Interceptor on an EJB 3.0 MDB Message Listener Method

You can designate one non-business method as the interceptor method for a message-driven bean (see "[Using Annotations](#)" on page 10-6). The method must have a signature of:

```
public Object <MethodName>(InvocationContext) throws Exception
```

For more information, see "[Understanding EJB 3.0 Interceptors](#)" on page 2-11.

Using Annotations

[Example 10-3](#) 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 10-3 @AroundInvoke in an EJB 3.0 Message-Driven Bean

```
@MessageDriven
public class MessageLogger implements MessageListener
{
    @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 a Lifecycle Callback Method for 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 lifecycle events (see ["Using Annotations"](#) on page 10-6):

- **Post-construct:** a method called before the first message listener method invocation on the bean. This is at a point after which any dependency injection has been performed by the container.
- **Pre-destroy:** a method called at the time the bean is removed from the pool or destroyed.

The message-driven bean class method must have the following signature:

```
public void <MethodName>()
```

For more information, see ["Callback Methods"](#) on page 1-6.

Using Annotations

You can specify an EJB 3.0 message-driven bean class method as a lifecycle callback method using any of the following annotations:

- `@PostConstruct`
- `@PreDestroy`

[Example 10-4](#) shows how to use the `@PostConstruct` annotation to specify EJB 3.0 message-driven bean class method `initialize` as a lifecycle callback method.

Example 10–4 @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
    }
    ...
}
```


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 EJB 2.1 Session Bean API"](#)

Implementing an EJB 2.1 Session Bean

This chapter explains how to implement an EJB 2.1 session bean, including:

- ["Implementing an EJB 2.1 Stateless Session Bean"](#) on page 11-1
- ["Implementing an EJB 2.1 Stateful Session Bean"](#) on page 11-4

Note: You can download EJB code examples from:

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

For more information, see:

- ["What is a Session Bean?"](#) on page 1-8
- ["Using EJB 2.1 Session Bean API"](#) 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 1-8.

Table 11-1 *Parts of an EJB 2.1 Stateless Session Bean*

Part	Description
Home Interface (remote or local)	Extends <code>javax.ejb.EJBHome</code> and <code>javax.ejb.EJBLocalHome</code> and requires a single <code>create()</code> factory method, with no arguments, and a single <code>remove()</code> method.
Component Interface (remote or local)	Extends <code>javax.ejb.EJBObject</code> for the remote interface and <code>javax.ejb.EJBLocalObject</code> for the local interface. It defines the business logic methods, which are implemented in the bean implementation.
TimedObject Interface	Optionally implements the <code>javax.ejb.TimedObject</code> interface. For more information, see "Understanding EJB Timer Services" on page 2-23).
Bean implementation	Implements <code>SessionBean</code> . This class must be declared as public, contain a public, empty, default constructor, no <code>finalize()</code> method, and implements the methods defined in the component interface. Must contain a single <code>ejbCreate</code> method, with no arguments, to match the <code>create()</code> method in the home interface. Contains empty implementations for the container service methods, such as <code>ejbRemove</code> , and so on.

1. Create the home interfaces for the bean (see ["Implementing the Home Interfaces"](#) on page 11-7).

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 11-7).
 - b. To create the local home interface, extend `javax.ejb.EJBLocalHome` (see ["Implementing the Local Home Interface"](#) on page 11-8).
2. Create the component interfaces for the bean (see ["Implementing the Component Interfaces"](#) on page 11-9).

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 11-9).
 - 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:
 - 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 Lifecycle 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 11-10).

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 methods below
    * are called by the client code.
```

```

* ----- */

public String sayHello(String myName) throws EJBException
{
    return ("Hello " + myName);
}

/* -----
* Begin private methods. The methods below
* are used internally.
* ----- */

...

/* -----
* Begin EJB-required methods. The methods below 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)
{
}

// Lifecycle 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 11-2](#) summarizes 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 1-10.

Table 11-2 Pats of an EJB 2.1 Stateful Session Bean

Part	Description
Home Interface (remote or local)	Extends <code>javax.ejb.EJBHome</code> and <code>javax.ejb.EJBLocalHome</code> and requires one or more <code>create()</code> factory methods, and a single <code>remove()</code> method.
Component Interface (remote or local)	Extends <code>javax.ejb.EJBObject</code> for the remote interface and <code>javax.ejb.EJBLocalObject</code> for the local interface. It defines the business logic methods, which are implemented in the bean implementation.
Bean implementation	Implements <code>SessionBean</code> . This class must be declared as public, contain a public, empty, default constructor, no <code>finalize()</code> method, and implement the methods defined in the remote interface. Must contain <code>ejbCreate</code> methods equivalent to the <code>create()</code> methods defined in the home interface. That is, each <code>ejbCreate</code> method is matched—by its parameter signature—to a <code>create</code> method defined in the home interface. Implements the container service methods, such as <code>ejbRemove</code> , and so on. Also, optionally implements the <code>SessionSynchronization</code> interface for Container-Managed Transactions, which includes <code>afterBegin</code> , <code>beforeCompletion</code> , and <code>afterCompletion</code> .

1. Create the home interfaces for the bean (see ["Implementing the Home Interfaces"](#) on page 11-7).

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 11-7).
- b. To create the local home interface, extend `javax.ejb.EJBLocalHome` (see ["Implementing the Local Home Interface"](#) on page 11-8).

2. Create the component interfaces for the bean (see ["Implementing the Component Interfaces"](#) on page 11-9).

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 11-9).
- 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:

- a. Implement the `ejb*` 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 Lifecycle 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 11-10).

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 methods below
    * 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 methods below
    * are used internally.
    * ----- */

    ...

    /* -----
```

```
* Begin EJB-required methods. The methods below 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;
}

// Lifecycle 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	Can have only a single <code>create</code> method, with no parameters.
Stateful Session Bean	Can have more 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 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 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 unmarshals the object on both ends.
- Any exception can be thrown to the client, as long as it is serializable. Runtime exceptions, including `EJBException` and `RemoteException`, are transferred back to the client as remote runtime 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 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
}
```

Using EJB 2.1 Session Bean API

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:

- ["What is a Session Bean?"](#) on page 1-8
- ["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 31-3	Basic
"Configuring Bean Instance Pool Timeouts for Session Beans" on page 31-4	Advanced
"Configuring a Transaction Timeout for a Session Bean" on page 21-2	Advanced
"Configuring a Lifecycle 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 non-transient attributes of your stateful session bean with object types supported by passivation (see ["What Object Types Can Be Passivated?"](#) on page 1-12), 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:

- ["When does Stateful Session Bean Passivation Occur?"](#) on page 1-11
- ["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
<code>enable-passivation</code>	"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:

- ["When does Stateful Session Bean Passivation Occur?"](#) on page 1-11
- ["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
<code>idletime</code>	Positive, integer number of seconds before passivation occurs. To disable this criteria, specify a value of "never".	"300"
<code>memory-threshold</code>	Percentage of JVM memory that can be consumed before passivation occurs. To disable this criteria, specify a value of "never".	"80"
<code>max-instances</code>	Maximum positive integer number of bean instances allowed in memory. When this value is reached, OC4J attempts to passivate the oldest in-memory bean instance. If unsuccessful, it waits <code>call-timeout</code> number of milliseconds (set in <code>orion-ejb-jar.xml</code> file element <code>entity-deployment</code>) to see if a bean instance is removed from memory, either through another passivation, calling the bean <code>remove</code> method, or bean expiration, before throwing a <code>TimeoutExpiredException</code> back to the client. To disable bean pooling, set this value to a negative number.	"0" (unlimited)
<code>max-instances-threshold</code>	Percentage of <code>max-instances</code> number of beans that can be in memory before passivation occurs. When this threshold is reached, passivation of beans occurs. For example, if <code>max-instances</code> is 100 beans, when <code>max-instances-threshold</code> reaches 90, OC4J begins passivation.	"90"

Table 12–3 (Cont.) orion-ejb-jar.xml Element session-deployment Passivation Criteria

Attribute	Values	Default
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:

- ["Where is a Passivated Stateful Session Bean Stored?"](#) on page 1-13
- ["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 Lifecycle Callback Method for an EJB 2.1 Session Bean

[Table 12–5](#) lists the EJB 2.1 session bean callback methods you can specify (see ["Using Java"](#) on page 12-4).

Table 12–5 EJB 2.1 Session Bean Lifecycle Callback Methods

Method	Description
ejbCreate	The container invokes this method to create an instance of the bean.
ejbActivate	The container invokes this method right after it reactivates the bean.
ejbPassivate	The container invokes this method right before it passivates the bean. You can turn off passivation for stateful session beans (see "Configuring Passivation" on page 12-1).
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.

Table 12–5 (Cont.) EJB 2.1 Session Bean Lifecycle Callback Methods

Method	Description
<code>setSessionContext</code>	This method associates a bean instance with its context information. The container calls this method after the bean creation. The enterprise bean can store the reference to the context object in an instance variable, for use in transaction management. Beans that manage their own transactions can use the session context to get the transaction context.

Session bean callback method signatures are defined in the `javax.ejb.SessionBean` interface.

Note: Using EJB 2.1, you must implement all session bean callback methods. If you do not need to take any action, implement an empty method.

For more information, see ["Callback Methods"](#) on page 1-6.

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:

- [Chapter 13, "Implementing an EJB 2.1 Entity Bean"](#)
- [Chapter 14, "Using EJB 2.1 CMP Entity Bean API"](#)
- [Chapter 15, "Using EJB 2.1 BMP Entity Bean API"](#)
- [Chapter 16, "Using EJB 2.1 Query API"](#)

Implementing an EJB 2.1 Entity Bean

This chapter explains how to implement an EJB 2.1 entity bean, including:

- "Implementing an EJB 2.1 CMP Entity Bean" on page 13-1
- "Implementing an EJB 2.1 BMP Entity Bean" on page 13-6

Note: You can download EJB code examples from:
http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html.

For more information, see:

- "What is an EJB 2.1 Entity Bean?" on page 1-18
- "Using EJB 2.1 CMP Entity Bean API" on page 14-1
- "Using EJB 2.1 BMP Entity Bean API" on page 15-1

Implementing an EJB 2.1 CMP Entity Bean

Table 13–1 summarizes the important parts of an EJB 2.1 CMP entity bean 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 CMP Entity Bean?" on page 1-19.

Table 13–1 *Parts of an EJB 2.1 CMP Entity Bean*

Part	Description
Home Interface (remote or local)	Extends <code>javax.ejb.EJBHome</code> for the remote home interface, <code>javax.ejb.EJBLocalHome</code> for the local home interface, and requires a single <code>create()</code> factory method, with no arguments, and a single <code>remove()</code> method.
Component Interface (remote or local)	Extends <code>javax.ejb.EJBObject</code> for the remote interface and <code>javax.ejb.EJBLocalObject</code> for the local interface. It defines the business logic methods, which are implemented in the bean implementation.
Bean implementation	Implements <code>EntityBean</code> . This class must be declared as public, contain a public, empty, default constructor, no <code>finalize()</code> method, and implements the methods defined in the component interface. Must contain one or more <code>ejbCreate</code> methods to match the <code>create</code> methods in the home interface. Contains empty implementations for the container service methods, such as <code>ejbRemove</code> , 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 1-30

- 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 an EJB 2.1 CMP Entity Bean Primary Key"](#) on page 14-1).

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 CMP entity bean:
 - a. Implement the abstract get and set methods that correspond to the get and set method(s) declared in the home interfaces.

For a CMP entity bean, the get and set 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 1-8).
 - 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 `ejbCreate` methods is the type of the bean's primary key.

For a CMP entity bean, 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
 - 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 1-32) for use within the business methods of your entity bean.
5. Create the appropriate database schema (tables and columns) for the entity bean.
- For a CMP entity bean, 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:

- ["Configuring Automatic Database Table Creation"](#) on page 14-5

You can configure the container to create the required tables for CMP beans.

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 EJB 2.1 CMP Entity Bean API"](#) on page 14-1).

Using Java

[Example 13-1](#) shows a typical implementation of an EJB 2.1 CMP entity bean.

[Example 13-2](#) shows the corresponding remote home interface and [Example 13-3](#) shows the corresponding remote component interface.

Example 13-1 EJB 2.1 CMP Entity Bean Implementation

```
package cmpapp;

import javax.ejb.*;
import java.rmi.*;

public abstract class EmployeeBean implements EntityBean
{
    private EntityContext ctx;

    // cmp 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
{
    // cmp 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 CMP entity bean shown in [Example 13–1](#).

Example 13–4 `ejb-jar.xml` For an EJB 2.1 CMP Entity Bean

```
...
<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 BMP Entity Bean

Table 13–2 summarizes the important parts of an EJB 2.1 BMP entity bean and 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 BMP Entity Bean?"](#) on page 1-22.

Table 13–2 *Parts of an EJB 2.1 BMP Entity Bean*

Part	Description
Home Interface (remote or local)	Extends <code>javax.ejb.EJBHome</code> for the remote home interface, <code>javax.ejb.EJBLocalHome</code> for the local home interface, and requires a single <code>create()</code> factory method, with no arguments, and a single <code>remove()</code> method.
Component Interface (remote or local)	Extends <code>javax.ejb.EJBObject</code> for the remote interface and <code>javax.ejb.EJBLocalObject</code> for the local interface. It defines the business logic methods, which are implemented in the bean implementation.
Bean implementation	Implements <code>EntityBean</code> . This class must be declared as public, contain a public, empty, default constructor, no <code>finalize()</code> method, and implements the methods defined in the component interface. Must contain one or more <code>ejbCreate</code> methods to match the <code>create</code> methods in the home interface. Contains complete implementations for the container service methods, such as <code>ejbStore</code> , <code>ejbLoad</code> , <code>ejbRemove</code> , 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.

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 BMP entity bean:

- 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 a BMP entity bean, the get and set 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 1-8).

- 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 `ejbCreate` methods is the type of the bean's primary key.

For a BMP entity bean, 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 EJB 2.1 BMP `ejbCreate` Method"](#) on page 13-15.

- e. Provide a complete implementation for each of the `javax.ejb.EntityBean` interface container callback methods (see ["Configuring a Lifecycle Callback Method for an EJB 2.1 BMP Entity Bean"](#) on page 15-4).

For a BMP entity bean, 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 an EJB 2.1 BMP Entity Bean Query"](#) on page 15-3).
5. Create the appropriate database schema (tables and columns) for the entity bean.
For a BMP entity bean, 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 BMP entity bean.
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 EJB 2.1 BMP Entity Bean API"](#) on page 15-1).

Using Java

[Example 13-5](#) shows a typical implementation of an EJB 2.1 BMP entity bean. [Example 13-7](#) shows the corresponding home interface and [Example 13-6](#) shows the corresponding remote interface.

Example 13-5 EJB 2.1 BMP Entity Bean Implementation

```
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 below
 */
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 BMP entity bean shown in [Example 13–5](#).

Example 13–8 ejb-jar.xml For an EJB 2.1 BMP Entity Bean

```
...
<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 BMP Entity Bean 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 EJB 2.1 BMP `ejbCreate` Method

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 we already have 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 1-30.

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 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 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 CMP entity bean in [Example 13-1](#) and [Example 13-6](#) shows the remote home interface corresponding to the EJB 2.1 BMP entity bean 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 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 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 EJB must be serializable, because RMI marshals and unmarshals the object on both ends.

- Any exception can be thrown to the client. Runtime exceptions, including `EJBException` and `RemoteException`, are transferred back to the client as remote runtime exceptions.
- A remote component interface can throw user specified application exceptions.

[Example 13-3](#) shows the remote component interface corresponding to the EJB 2.1 CMP entity bean in [Example 13-1](#) and [Example 13-7](#) shows the remote component interface corresponding to the EJB 2.1 BMP entity bean 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 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 Enterprise JavaBeans 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 lifetime 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 EntityBean {
    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
}
```


Using EJB 2.1 CMP Entity Bean API

This chapter describes the various options that you must configure in order to use an EJB 2.1 CMP entity bean.

[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:

- ["What is an EJB 2.1 CMP Entity Bean?"](#) on page 1-19
- ["Implementing an EJB 2.1 CMP Entity Bean"](#) on page 13-1

Table 14–1 Configurable Options for an EJB 2.1 CMP Entity Bean

Options	Type
"Configuring an EJB 2.1 CMP Entity Bean Primary Key" on page 14-1	Basic
"Configuring Automatic Database Table Creation" on page 14-5	
"Configuring an EJB 2.1 CMP Entity Bean Container-Managed Persistence Field" on page 14-6	Basic
"Configuring an EJB 2.1 CMP Entity Bean Container-Managed Relationship Field" on page 14-8	Basic
"Configuring Default Mappings" on page 14-9	Basic
"Configuring Exclusive Write Access to the Database" on page 14-10	Advanced
"Configuring Lazy Loading on Finder Methods" on page 14-11	Advanced
"Configuring Bean Instance Pool Size" on page 31-3	Basic
"Configuring Bean Instance Pool Timeouts for Entity Beans" on page 31-4	Advanced

Configuring an EJB 2.1 CMP Entity Bean Primary Key

Every EJB 2.1 CMP entity bean must have a primary key field.

You can configure the primary key as a well-known Java type (see ["Configuring an EJB 2.1 CMP Entity Bean Primary Key Field"](#) on page 14-1) or as a special type that you create (see ["Configuring an EJB 2.1 CMP Entity Bean Composite Primary Key Class"](#) on page 14-3).

You can either assign primary key values yourself, or, more typically, you can associate a primary key field with a primary key value generator (see ["Configuring EJB 2.1 CMP Entity Bean Automatic Primary Key Generation"](#) on page 14-4).

Configuring an EJB 2.1 CMP Entity Bean Primary Key Field

For a simple EJB 2.1 CMP entity bean, 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"](#) 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.

Once this configuration is complete, the container manages the instantiation of primary keys of this type and initializes your entity bean primary key field accordingly.

If you specify your primary key type as `java.lang.Object`, you can rely on the container to automatically handle the allocation of primary key values (see ["Configuring EJB 2.1 CMP Entity Bean Automatic Primary Key Generation"](#) on page 14-4).

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*

```
<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>
```

Within the `orion-ejb-jar.xml` file, the primary key is mapped to the underlying database persistence storage by mapping the CMP field or primary key field defined in the `ejb-jar.xml` file to the database column name. [Example 14-2](#) shows the `EmpBean` persistence storage is defined as the `EMP` table in the database that is defined in the `jdbc/OracleDS` data source. Following the `<entity-deployment>` element definition, the primary key, `empNo`, is mapped to the `EMPNO` column in the `Emp` table, and the `empName` and `salary` CMP fields are mapped to `EMPNAME` and `SALARY` columns respectively in the `EMP` table.

Example 14-2 *orion-ejb-jar.xml for Primary Key Field*

```
<entity-deployment name="EmployeeBean" ...table="EMP"
  data-source="jdbc/OracleDS"... >
  <primkey-mapping>
```

```

    <cmp-field-mapping name="empNo" persistence-name="EMPNO" />
  </primkey-mapping>
  <cmp-field-mapping name="empName" persistence-name="EMPNAME" />
  <cmp-field-mapping name="salary" persistence-name="SALARY" />
  ...

```

Configuring an EJB 2.1 CMP Entity Bean Composite Primary Key Class

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 EJB, you must:

- Code your bean's `ejbCreate` method to return the primary key class type (see ["Implementing an EJB 2.1 Entity Bean"](#) on page 13-1)
- Configure your deployment XML to use it (see ["Using Deployment XML"](#) on page 14-4)

Once this configuration is complete, the container manages the instantiation of primary keys of this type and initializes your entity bean primary key field accordingly.

Using Java

[Example 14-3](#) shows an example primary key class.

Example 14-3 EJB 2.1 CMP Entity Bean Primary Key Class Implementation

```

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 14-4](#) 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 14-4 `ejb-jar.xml` For a Primary Key Class and Its Instance Variables

```
<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.

The CMP fields are mapped in the `orion-ejb-jar.xml` in the same manner as described in "[Configuring an EJB 2.1 CMP Entity Bean Primary Key Field](#)" on page 14-1. However, with a complex primary key, the mapping contains more than a single field; thus, the `<primkey-mapping><cmp-field-mapping>` element contains another subelement: the `<fields>` element. All of the fields of a primary key are each defined in a separate `<cmp-field-mapping>` element within the `<fields>` element, as [Example 14-5](#) shows.

Example 14-5 `orion-ejb-jar.xml` for Primary Key Field

```
<primkey-mapping>
  <cmp-field-mapping>
    <fields>
      <cmp-field-mapping name="empNo" persistence-name="EMPNO" />
    </fields>
  </cmp-field-mapping>
</primkey-mapping>
```

Configuring EJB 2.1 CMP Entity Bean Automatic Primary Key Generation

If you specify the type of your primary key field (see "[Configuring an EJB 2.1 CMP Entity Bean Primary Key Field](#)" on page 14-1) as `java.lang.Object` but do not

specify the primary key name, then the primary key is auto-generated by the container (see ["Using Deployment XML"](#) on page 14-5).

Using Deployment XML

[Example 14-6](#) shows the `ejb-jar.xml` for an unnamed primary key field of type `Object`.

Example 14-6 *ejb-jar.xml for Primary Key Field with Type Object*

```
<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.Object</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 creates a column in the entity bean table for the primary key of type `LONG`. The container uses random numbers for the primary key values. This is generated in the `orion-ejb-jar.xml` for the bean as [Example 14-7](#). In this case, the container will create a column named `autoid`.

Example 14-7 *orion-ejb-jar.xml for Automatically Generated Primary Key Field*

```
<primkey-mapping>
  <cmp-field-mapping name="auto_id" persistence-name="autoid"/>
</primkey-mapping>
```

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 Mappings"](#) on page 14-9).

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	True ¹ or False
		autodelete-tables	True or False ¹
Application (EAR)	orion-application.xml	autocreate-tables	True ¹ or False
		autodelete-tables	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 ["Configuring default-mapping Properties"](#) on page 8-11.³ 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 an EJB 2.1 CMP Entity Bean Container-Managed Persistence Field

You do not define CMP fields in the entity bean class: CMP fields are virtual only. OC4J supplies the implementation of the CMP fields.

You must define public, abstract get and set methods for the CMP fields, using the JavaBeans conventions (see ["Using Java"](#) on page 14-7). OC4J supplies the implementation of these methods. You must not expose these get and set methods in the remote interface of the entity bean.

You may assign only the following Java types to CMP 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 CMP 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 14-7). 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 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 Persistence Fields?"](#) on page 1-20.

Using Java

[Example 14-8](#) shows the abstract get and set methods for the CMP fields specified in the `ejb-jar.xml` file (see ["Using Deployment XML"](#) on page 14-7).

Example 14-8 EJB 2.1 Container-Managed Persistence Fields

```
package cmpapp;

import javax.ejb.*;
import java.rmi.*;

public abstract class EmployeeBean implements EntityBean
{
    private EntityContext ctx;

    // cmp 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 14-9](#) shows the `cmp-field` elements for the get and set methods specified in the bean class (see ["Using Java"](#) on page 14-7).

Example 14-9 ejb-jar.xml for an EJB 2.1 CMP 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 an EJB 2.1 CMP Entity Bean Container-Managed Relationship Field

You do not define CMR fields in the entity bean class: CMR fields are virtual only. OC4J supplies the implementation of the CMR fields.

You must define `public`, `abstract` get and set methods for the CMR fields in the local interface of the related entity bean, using the JavaBeans conventions (see ["Using Java"](#) on page 14-8). OC4J supplies the implementation of these methods. You must not expose these get and set methods in the remote interface of the entity bean.

You may assign only the following Java types to CMP fields: Java primitive types and Java serializable types. You may assign an entity bean local interface type (or a collection of such) to a CMR 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 14-9). 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 CMR 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 CMR 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:

- ["What are Container-Managed Relationship Fields?"](#) on page 1-20
- ["Configuring Default Mappings"](#) on page 14-9

Using Java

[Example 14-10](#) shows the abstract get and set methods for the CMR fields specified in the `ejb-jar.xml` file (see ["Using Deployment XML"](#) on page 14-9).

Example 14-10 EJB 2.1 Container-Managed Relationship Fields

```
package cmpapp;

import javax.ejb.*;
import java.rmi.*;

public abstract class EmployeeBean implements EntityBean
{

    private EntityContext ctx;

    // cmp 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 14-11](#) shows the `cmr-field` elements for the get and set methods specified in the bean class (see ["Using Java"](#) on page 14-8).

Example 14-11 *ejb-jar.xml for an EJB 2.1 CMR 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 Default Mappings

You can configure OC4J to automatically generate all required mappings at deployment time (see ["Using Deployment XML"](#) on page 14-9). To use this feature, you must:

- Omit all container-managed relationship configuration (see ["Configuring an EJB 2.1 CMP Entity Bean Container-Managed Persistence Field"](#) on page 14-6).
- Ensure that no `toplink-ejb-jar.xml` is present in the EJB module (see ["What is the `toplink-ejb-jar.xml` File?"](#) on page 2-8).

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 mapping, 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	<p>Optional element that determines what TopLink will do to prepare the database tables that are being mapped to. Valid values are:</p> <ul style="list-style-type: none"> ■ 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 <i>"Table already existed..."</i>) 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 <code>SQLException</code> 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. <p>If no <code>orion-ejb-jar.xml</code> file is defined in your EAR file, the OC4J container generates one during deployment. In this case, to specify a value for <code>db-table-gen</code>, use the TopLink system property <code>toplink.defaultmapping.dbTableGenSetting</code>. For example: <code>-Dtoplink.defaultmapping.dbTableGenSetting="DropAndCreate"</code>.</p> <p>The <code>orion-ejb-jar.xml</code> property overrides the system property. If both the <code>orion-ejb-jar.xml</code> property and the system property are present, TopLink retrieves the setting from the <code>orion-ejb-jar.xml</code> file.</p> <p>This setting overrides <code>autocreate-tables</code> and <code>autodelete-tables</code> configuration at the application (EAR) or system level. For more information, see "Automatic Database Table Creation" on page 6-6.</p>
extended-table-names	<p>An element used if the generated table names are not long enough to be unique. Values are restricted to <code>true</code> or <code>false</code> (default). When set to <code>true</code>, the TopLink run time will ensure that generated tables names are unique.</p> <p>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.</p> <p>When <code>extended-table-names</code> is set to <code>false</code> (the default), a simple table naming algorithm is used as follows: table names are defined as <code>TL_<bean_name></code>. For example, if the bean name is <code>Employee</code>, the associated table name would be <code>TL_EMPLOYEE</code>.</p> <p>However, if the same entity is defined in multiple JAR files in an application, or across multiple applications, table-naming collision is inevitable.</p> <p>To address this problem, set <code>extended-table-names</code> to <code>true</code>. When set to <code>true</code>, TopLink uses an alternative table-naming algorithm as follows: table names are defined as <code><bean_name>_<jar_name>_<app_name></code>. 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 <code>Employee</code>, which is in <code>Test.jar</code>, which is in <code>Demo.ear</code> (and the application name is "Demo"), then the corresponding table name will be <code>EMPLOYEE_TEST_DEMO</code>.</p> <p>If there is no <code>orion-ejb-jar.xml</code> file defined in the EAR file, the OC4J container generates one during deployment. In this case, to specify a value for <code>extended-table-names</code>, use the TopLink system property <code>toplink.defaultmapping.useExtendedTableNames</code>. For example: <code>-Dtoplink.defaultmapping.useExtendedTableNames="true"</code>.</p> <p>The <code>orion-ejb-jar.xml</code> property overrides the system property. If both the <code>orion-ejb-jar.xml</code> property and the system property are present, TopLink retrieves the setting from the <code>orion-ejb-jar.xml</code> file.</p>

Configuring Exclusive Write Access to the Database

The `exclusive-write-access` attribute of the `<entity-deployment>` element states that this is the only bean that accesses its table in the database and that no

external methods are used to update the resource. It informs the OC4J instance that any cache maintained for this bean will only be dirtied by this bean. Essentially, if you set this attribute to true, you are assuring the container that this is the only bean that will update the tables used within this bean. Thus, any cache maintained for the bean does not need to constantly update from the back-end database.

This flag does not prevent you from updating the table; that is, it does not actually lock the table. However, if you update the table from another bean or manually, the results are not automatically updated within this bean.

The default for this attribute is false. Because of the effects of the entity bean concurrency modes, this element is only allowed to be set to true for a read-only entity bean. OC4J will always reset this attribute to false for pessimistic and optimistic concurrency modes.

For more information, see ["How do You Avoid Database Resource Contention?"](#) on page 1-24.

Using Deployment XML

[Example 14-12](#) shows the `orion-ejb-jar.xml` file element `entity-deployment` attribute `exclusive-write-access` configured to enable exclusive write access.

Example 14-12 *orion-ejb-jar.xml for EJB 2.1 CMP Entity Bean Exclusive Write Access*

```
<entity-deployment ... exclusive-write-access="true"
...
</entity-deployment>
```

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 a CMP bean, one or more objects are retrieved with all of their CMP fields. So, for example, with the `findAllEmployees` method, this finder retrieves all employee objects with all of the CMP 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>
```

Note: If you set this attribute to true, the min/max-instances-per-pk attribute is ignored.

Using EJB 2.1 BMP Entity Bean API

This chapter describes the various options that you must configure in order to use an EJB 2.1 BMP entity bean.

[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:

- ["What is an EJB 2.1 BMP Entity Bean?"](#) on page 1-22
- ["Implementing an EJB 2.1 BMP Entity Bean"](#) on page 13-6

Table 15–1 Configurable Options for an EJB 2.1 BMP Entity Bean

Options	Type
"Configuring a Read-Only BMP Entity Bean" on page 15-1	Advanced
"Configuring BMP Commit Options" on page 15-2	Advanced
"Configuring an EJB 2.1 BMP Entity Bean Query" on page 15-3	Basic
"Configuring a Lifecycle Callback Method for an EJB 2.1 BMP Entity Bean" on page 15-4	Basic

Configuring a Read-Only BMP Entity Bean

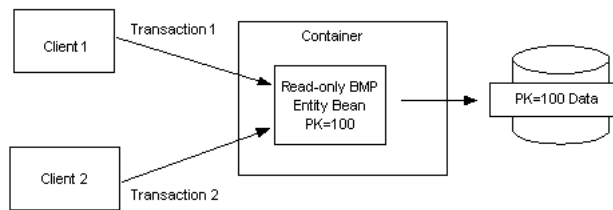
You can configure a BMP entity bean as read-only. By doing so, you enter into a contract with OC4J by which you guarantee not to change the BMP entity bean's state after it is activated. Unlike CMP read-only, no exception will be thrown if you do update a read-only BMP bean.

When you configure a BMP entity bean as read-only, OC4J uses a special case of commit option A (see ["Configuring BMP Commit Options"](#) on page 15-2) to improve performance by:

- 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 BMP entity bean by primary key are allocated a single instance. Both Client 1 and Client 2 are satisfied by the same cached read-only BMP entity bean instance. Because the BMP entity bean 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 lifecycle methods.

Figure 15–1 Read-Only BMP Entity Beans and Commit Option A

Using Deployment XML

[Example 15–1](#) shows the `orion-ejb-jar.xml` file `entity-deployment` element `locking-mode` attribute mode configured to specify a BMP entity bean as read-only.

Example 15–1 `orion-ejb-jar.xml` For Read-Only

```

<entity-deployment
  name="EmployeeBean"
  location="bmpapp/EmployeeBean"
  locking-mode="read-only"
>
...
</entity-deployment>
  
```

Configuring BMP Commit Options

For a BMP entity bean, 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 BMP entity bean to use commit option A (see ["Configuring a Read-Only BMP Entity Bean"](#) on page 15-1), you can further improve performance by taking advantage of read-only BMP entity bean caching (see ["Commit Options and BMP Applications"](#) on page 1-27).

Commit option C is the default.

For more information, see ["What are Entity Bean Commit Options?"](#) on page 1-26.

Using Deployment XML

[Example 15–2](#) 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 15–2 `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 an EJB 2.1 BMP Entity Bean Query

You must implement an `ejbFindByPrimaryKey` method for a BMP entity bean (see ["Implementing an EJB 2.1 BMP the ejbFindByPrimaryKey Method"](#) on page 15-3). Optionally, you may configure other finders (see ["Implementing Other EJB 2.1 BMP Finder Methods"](#) on page 15-3).

For more information, see ["Using EJB 2.1 Query API"](#) on page 16-1.

Implementing an EJB 2.1 BMP the ejbFindByPrimaryKey Method

The `ejbFindByPrimaryKey` implementation is a requirement for all BMP entity beans. 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 EJB 2.1 BMP Finder Methods

Optionally, you can create other finder methods in addition to the single `ejbFindByPrimaryKey`.

To create other finder methods, do the following:

1. Add the finder method to the home interface.

2. Implement the finder method in the BMP bean implementation.

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 BMP 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 Lifecycle Callback Method for an EJB 2.1 BMP Entity Bean

In a BMP entity bean, you are responsible for implementing all of the EJB 2.1 BMP entity bean lifecycle callback methods:

Implementing an EJB 2.1 BMP `ejbStore` Method

The `ejbStore` method is called by the container before the object is passivated or whenever a transaction is about to end. Its purpose is to save the persistent data to an outside resource, such as a database.

The container invokes the `ejbStore` method when the persistent data should be saved to the database. This synchronizes the state of the instance to the entity in the underlying database. For example, the container invokes before the container passivates the bean instance or removes the instance. The BMP bean is responsible for ensuring that all data is stored to some resource, such as a database, within this method.

```
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());
        }
    }
}
}

```

Implementing an EJB 2.1 BMP `ejbLoad` Method

The `ejbLoad` method is called by the container before the object is activated or whenever a transaction has begun, or when an entity bean is instantiated. Its purpose is to restore any persistent data that exists for this particular bean instance.

The container invokes the `ejbLoad` method whenever it needs to synchronize the state of the bean with what exists in the database. This method is invoked after activating the bean instance to refresh it with the state that is in the database. The purpose of this method is to repopulate the persistent data with the saved state. For most `ejbLoad` methods, this implies reading the data from a database into the instance data variables.

```

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());
    }
}

```

Implementing an EJB 2.1 BMP `ejbPassivate` Method

The `ejbPassivate` method is invoked directly before the bean instance is serialized for future use. It will be re-activated, through the `ejbActivate` method, the next time the user invokes a method on this instance.

Before the bean is passivated, you should release all resources and release any static information that would be too large to be serialized. Any large, static information that can be easily regenerated within the `ejbActivate` method should be released in this method.

In our example, the only resource that cannot be serialized is the open database connection. It is closed in this method and reopened in the `ejbActivate` method.

```

public void ejbPassivate()
{
    // Container invokes this method on an instance before the instance

```

```
        // becomes disassociated with a specific EJB object
        conn.close();
    }
}
```

Implementing an EJB 2.1 BMP `ejbActivate` Method

The container invokes this method when the bean instance is reactivated. That is, the user has asked to invoke a method on this instance. This method is used to open resources and rebuild static information that was released in the `ejbPassivate` method.

In addition, the container invokes this method after the start of any transaction.

Our employee example opens the database connection where the employee information is stored.

```
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
    conn = getConnection(dsName);
}
```

Implementing an EJB 2.1 BMP `ejbRemove` Method

The container invokes the `ejbRemove` method before removing the bean instance itself or by placing the instance back into the bean pool. This means that the information that was represented by this entity bean should be removed from within persistent storage. The employee example removes the employee and all associated information from the database before the instance is destroyed. Close the database connection.

```
public void ejbRemove() throws RemoveException
{
    //Container invokes this method before 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());
        }
    }
}
```

Using EJB 2.1 Query API

This chapter describes:

- [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:

- ["How Do You Query for an EJB 2.1 Entity Bean?"](#) on page 1-27
- ["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 1-30 and ["Default TopLink Finders"](#) on page 1-31).

For more information, see ["Understanding Finder Methods"](#) on page 1-30.

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 1-30 and ["Default TopLink Finders"](#) on page 1-31), 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>` sub-element.

The `<abstract-schema-name>` sub-element 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 sub-elements:

- `<description>`: optional explanatory text
- `<query-method>`: describes the finder method and includes the following sub-elements:
 - `<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>` sub-element 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 an EJB 2.1 CMP Entity Bean Container-Managed Relationship Field"](#) on page 14-8.

Using Java

[Example 16-1](#) shows a remote home interface called `EmpBeanHome`.

Example 16-1 Finder Methods in an EJB 2.1 CMP Entity Bean 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. We 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) throws 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>
  </entity>
</enterprise-beans>

```

```

</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:

- "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 1-32.

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):
 - a. For each entity bean that you plan to reference in your EJB QL query, configure the `<entity>` element `<abstract-schema-name>` sub-element.
 The `<abstract-schema-name>` sub-element 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 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 CMP entity bean with select methods.

Example 16-3 EJB 2.1 CMP Entity Bean Implementation 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
  persistence 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. We 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.
 * Above are the two design considerations 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, we 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 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 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 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 MDB"](#)
- [Chapter 18, "Using EJB 2.1 MDB API"](#)

Implementing an EJB 2.1 MDB

This chapter explains how to implement an EJB 2.1 message-driven bean.

For more information, see:

- ["What is a Message-Driven Bean?"](#) on page 1-33
- ["Using EJB 2.1 MDB API"](#) on page 18-1

Implementing an EJB 2.1 MDB

[Table 17-1](#) summarizes the important parts of an EJB 2.1 MDB entity bean and the following procedure describes how to implement these parts. For a typical implementation, see ["Using Java"](#) on page 17-2.

Table 17-1 *Parts of an EJB 2.1 MDB Entity Bean*

Part	Description
Bean implementation	<p>This class must be declared as <code>public</code>, contain a <code>public</code>, empty, default constructor, one <code>public</code>, <code>void</code> <code>ejbCreate</code> method with no arguments, and no <code>finalize()</code> method.</p> <p>Implements <code>javax.ejb.MessageDrivenBean</code> to provide an empty implementation for lifecycle method <code>ejbRemove</code> and an implementation of the <code>setMessageDrivenContext</code> method.</p> <p>Implements <code>javax.jms.MessageListener</code> to provide an implementation of the <code>onMessage</code> method.</p>

For more information, see ["What is a Message-Driven Bean?"](#) on page 1-33.

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:

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 `ejbCreate` methods is `void`.

- d. Provide an empty implementation for each of the `javax.ejb.MessageDrivenBean` interface container callback methods.
- 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:
 - ["Configuring an EJB 2.1 MDB to Use a Non-J2CA Message Service Provider"](#) on page 18-1
 - ["Configuring an EJB 2.1 MDB to Use a J2CA Message Service Provider"](#) on page 18-2
 - 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 OracleAS JMS (see ["Oracle Application Server JMS \(OracleAS JMS\) Provider: File-Based"](#) on page 2-17), 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 methods below
    * are used internally.
    * ----- */

    ...

    /* -----
    * Begin EJB-required methods. The methods below 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()
    {
    }

    /* setMessageDrivenContext method */
    public void setMessageDrivenContext(MessageDrivenContext ctx)
    {
        /* As with all EJBs, you must set the context in order to be
        able to use it at another time within the MDB methods. */
        m_ctx = ctx;
    }

    // Lifecycle Methods

    /* ejbRemove method */
    public void ejbRemove()
    {
    }

    /* -----
    * Begin JMS MessageListener-required methods. The methods
    * below are called by the container, and never called by
    * client code.
    * ----- */

    /**
    * Receives the incoming Message and displays the text.
    */
    public void onMessage(Message msg)
    {
        /* An 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_gc = qcf.createQueueConnection();
        // 3. Create the session over the queue connection.
        m_qs = m_gc.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_gc.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>
  </container-transaction>
</assembly-descriptor>
```

```
<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 Use a Non-J2CA Message Service Provider"](#) on page 18-1.

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 Enterprise JavaBeans 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 EJBs, 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
}
```

Using EJB 2.1 MDB API

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:

- ["What is a Message-Driven Bean?"](#) on page 1-33
- ["Implementing an EJB 2.1 MDB"](#) 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 Use a Non-J2CA Message Service Provider" on page 18-1	Basic
"Configuring an EJB 2.1 MDB to Use a J2CA Message Service Provider" on page 18-2	Basic
"Configuring an MDB for Fast Undeploy on Windows" on page 18-4	Advanced
"Configuring an MDB for Oracle RAC Failover" on page 18-5	Advanced
"Configuring Bean Instance Pool Size" on page 31-3	Basic
"Configuring a Transaction Timeout for a Message-Driven Bean" on page 21-2	Advanced
"Configuring Listener Threads" on page 18-6	Advanced
"Configuring Dequeue Retry Count and Interval" on page 18-8	Advanced

Configuring an EJB 2.1 MDB to Use a Non-J2CA Message Service Provider

You can configure an EJB 3.0 MDB to use a non-J2CA message service provider using deployment XML (see ["Using Deployment XML"](#) on page 18-1).

For more information, see:

- ["Oracle Application Server JMS \(OracleAS JMS\) Provider: File-Based"](#)
- ["Oracle JMS \(OJMS\) Provider: Advanced Queueing \(AQ\)-Based"](#)

Using Deployment XML

You can use 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"](#) on page 19-7.

Example 18–1 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 have defined connection factory jms/MyQCF and queue jms/MyQueue in the jms.xml file. For more information on configuring a non-J2CA message service provider, see ["Configuring an OracleAS JMS Message Service Provider"](#) on page 23-1 or ["Configuring an OJMS Message Service Provider"](#) on page 23-3.

Example 18–1 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>
```

For a complete list of all activation configuration properties, download and unzip one of the how-to-gjra-with-xxx.zip files from http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html, where xxx 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 "JMS Resource Adapter" in the *Oracle Containers for J2EE Services Guide*.

The actual names you use depend on your message service provider installation. For more information, see:

- ["OracleAS JMS Destination and Connection Factory Names"](#) on page 23-2
- ["OJMS Destination and Connection Factory Names"](#) on page 23-3

Configuring an EJB 2.1 MDB to Use a J2CA Message Service Provider

You can configure an EJB 3.0 MDB to use a J2CA message service provider using deployment XML (see ["Using Deployment XML"](#) on page 18-3).

For more information, see ["J2EE Connector Architecture \(J2CA\) Adapter Message Provider"](#) on page 2-18.

Using Deployment XML

You must use both `ejb-jar.xml` and `orion-ejb-jar.xml` file. 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"](#) on page 19-7.

To configure an EJB 3.0 MDB to use a J2CA message service provider:

1. Configure the `ejb-jar.xml` file.

[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 have defined connection factory `OracleASjms/MyQCF` in the `oc4j-ra.xml` file and destination name `OracleASjms/MyQueue` in the `oc4j-connectors.xml`. For more information on configuring a J2CA message service provider, see ["Configuring a Message Service Provider Using J2CA"](#) on page 23-7. To complete this configuration, you must

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>
```

2. Configure the `orion-ejb-jar.xml` file.

[Example 18-3](#) shows how to configure the `orion-ejb-jar.xml` 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 18–3 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>
```

For a complete list of all activation configuration properties, download and unzip one of the how-to-gjra-with-xxx.zip files from http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html, where xxx 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 "JMS Resource Adapter" in the *Oracle Containers for J2EE Services Guide*.

You may also set the optional attributes that [Table A–4](#) 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 23-8.

Configuring an MDB for Fast Undeploy on Windows

When you use an MDB, it is blocked in a receive state waiting for incoming messages. In a non-Windows environment, if you shutdown 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 OJMS provider (see "[Oracle JMS \(OJMS\) Provider: Advanced Queueing \(AQ\)-Based](#)" on page 2-17) 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 18-4), 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 OJMS 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 dequeuing fails (see ["Using Deployment XML"](#) on page 18-5)
- Configure the MDB client to retry if connection acquisition fails (see ["Using Java"](#) on page 18-5)

Note: The 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 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 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/TopicConnectionFactory/agtcf"
  destination-location="java:comp/resource/cartojms1/Topics/topic1"
  subscription-name="MDBSUB"
  dequeue-retry-count=3
  dequeue-retry-interval=90/>
...
```

Using Java

To support RAC failover, you must configure a standalone OJMS client running against an 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 re-establish the database connection if necessary.

Example 18-5 *Client Retrying After Connection Acquisition Failure*

```
import com.evermind.sql.DbUtil;
...
getMessage(QueueSession session)
{
    try
    {
        QueueReceiver rcvr = session.createReceiver(rcvrQueue);
        Message msgRec = rcvr.receive();
    }
    catch(Exception e )
    {

```

```
        if (exc instanceof JMSEException)
        {
            JMSEException jmsexc = (JMSEException) exc;
            sql_ex = (SQLException)(jmsexc.getLinkedException());
            db_conn = oracle.jms.AQjmsSession(session).getDBConnection();
            if ((DbUtil.oracleFatalError(sql_ex, db_conn))
            {
                // failover logic
            }
        }
    }
}
```

Configuring Listener Threads

By configuring the number of listener threads to x , where x is greater than one (see ["Using Deployment XML"](#) on page 18-6), OC4J will instantiate x number of message-driven bean instances all listening to the message-driven bean's message location in parallel.

Topics can only have one thread. Queues can have more than one.

Using Deployment XML

You set the number of listener threads 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 OracleAS JMS or Oracle JMS (OJMS), use the `listener-threads` attribute of the `<message-driven-deployment>` element.

For example, if you are using OracleAS JMS or Oracle JMS (OJMS), and you wanted to set the number of listener threads to 3, you would do as follows:

```
<message-driven-deployment ... listener-threads="3"
...
</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 `listenerThreads` configuration property.

For example, if you are using a J2CA adapter message service provider, and you wanted to set the number of listener threads to 3, you would do as follows:

```
<message-driven-deployment ... >
...
    <config-property>
        <config-property-name>listenerThreads</config-property-name>
        <config-property-value>3</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.

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 18-7).

After this number of re deliveries, the message is deemed undeliverable and is handled according to the policies of your message service provider. For example, OracleAS 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:

- [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 OracleAS JMS or Oracle JMS (OJMS), use the `max-delivery-count` attribute of the `<message-driven-deployment>` element.

For example, if you are using OracleAS JMS or Oracle JMS (OJMS), and 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>
```

J2CA Adapter Message Service Provider

If you are using a J2CA adapter message service provider, use the `<config-property>` element to set the `maxDeliveryCount` 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>
```

In either case, if you change this property using this method, you must restart OC4J to apply your changes.

Configuring Dequeue Retry Count and Interval

You can configure how often a message-driven bean's listener thread tries to re-acquire its JMS session once database failover has occurred and how many seconds to wait between retries (see ["Using Deployment XML"](#) on page 18-8).

This value is only for CMT transactions in a message-driven bean.

For more information about failover, see ["Understanding OC4J EJB Application Clustering Services"](#) on page 2-20.

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:

- [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 OracleAS JMS or Oracle JMS (OJMS), 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 OracleAS JMS or Oracle JMS (OJMS), 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>
```

J2CA Adapter Message Service Provider

If you are using a J2CA adapter message service provider, use the `<config-property>` element to set the `dequeueRetryCount` and `dequeueRetryInterval` configuration properties.

For example, if you are using a J2CA adapter message service provider, and you wanted to set the number of listener threads to 3, you would do as follows:

```
<message-driven-deployment ... >
...
  <config-property>
    <config-property-name>DequeueRetryCount</config-property-name>
    <config-property-value>3</config-property-value>
  </config-property>
  <config-property>
    <config-property-name>dequeueRetryInterval</config-property-name>
    <config-property-value>90</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.

Part VIII

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:

- [Chapter 19, "Configuring JNDI Services"](#)
- [Chapter 20, "Configuring Data Sources"](#)
- [Chapter 21, "Configuring Transaction Services"](#)
- [Chapter 22, "Configuring Security Services"](#)
- [Chapter 23, "Configuring Message Services"](#)
- [Chapter 24, "Configuring OC4J EJB Application Clustering Services"](#)
- [Chapter 25, "Configuring Timer Services"](#)

Configuring JNDI Services

This chapter describes:

- [Configuring Environment References](#)
- [Configuring the Initial Context Factory](#)
- [Setting JNDI Properties in an EJB](#)
- [Looking up an EJB 3.0 EJB](#)
- [Looking Up an EJB 3.0 Resource Manager Connection Factory](#)
- [Looking Up an EJB 3.0 Environment Variable](#)
- [Looking Up an EJB 2.1 EJB](#)
- [Looking Up an EJB 2.1 Resource Manager Connection Factory](#)
- [Looking Up an EJB 2.1 Environment Variable](#)

For more information, see:

- ["Understanding EJB JNDI Services"](#) on page 2-13
- ["Accessing an EJB 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 runtime using JNDI, you must define environment references to them. Environment references are static and cannot be changed by the bean.

This section describes configuring an environment reference to:

- [EJB Environment References](#)
- [Resource Manager Connection Factory Environment References](#)
- [Environment Variable Environment References](#)
- [Web Service Environment References](#)

In EJB 3.0, you can use annotations, resource injection, and default JNDI names (based on class and interface names) instead of defining environment references.

In EJB 2.1, you must define `<ejb-ref>` or `<ejb-local-ref>` elements in the appropriate deployment descriptor (see ["Where Do You Configure an EJB Environment Reference?"](#) on page 19-3).

When you define an environment reference, you can use the actual JNDI name or use a logical name ("[Should You Use Logical Names?](#)" on page 19-3) associated with it to increase deployment flexibility.

EJB Environment References

Before one EJB, acting in the role of a client can access another EJB, you must define an EJB reference to the target EJB.

For more information, see "[Configuring an Environment Reference to an EJB](#)" on page 19-3.

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 annotations and resource injection (see "[Looking Up an EJB 3.0 Resource Manager Connection Factory](#)" on page 19-22).

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:

- [Configuring an Environment Reference to a JDBC Data Source Resource Manager Connection Factory](#) on page 19-6
- [Configuring an Environment Reference to a JMS Destination or Connection Resource Manager Connection Factory](#) on page 19-7
- [Configuring an Environment Reference to a Java Mail Resource Manager Connection Factory](#) on page 19-10
- [Configuring an Environment Reference to a URL Resource Manager Connection Factory](#) on page 19-12

Environment Variable Environment References

You can define an environment variable with an environment reference to make the environment variable value accessible using JNDI.

For more information, see "[Configuring an Environment Reference to an Environment Variable](#)" on page 19-13

Web Service Environment References

You can define a web service with an environment reference to make the web service accessible using JNDI

For more information, see "[Configuring an Environment Reference to a Web Service](#)" on page 19-14.

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 EJB invoking an EJB from within the container.	ejb-jar.xml	orion-ejb-jar.xml
Stand-alone client	A pure-Java client invoking an EJB from outside of the container.	application-client.xml	orion-application-client.xml
Servlet or JSP	A servlet or JSP invoking an EJB from outside of the container.	web.xml	orion-web.xml

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 EJBs, 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 an EJB

Before one EJB, acting in the role of a client (call it the source EJB), can access another EJB (call it the target EJB), you must define an EJB reference to the target EJB in the deployment descriptor of the source EJB.

Note: In EJB 3.0, an environment reference to a target EJB is not needed. You can access a target EJB directly using annotations and resource injection (see ["Accessing an EJB 3.0 EJB"](#) on page 29-4).

This section describes:

- [Configuring an Environment Reference to a Remote EJB](#)
- [Configuring an Environment Reference to a Local EJB](#)

Configuring an Environment Reference to a Remote EJB

To define an EJB reference to the remote interface of a target EJB, you configure an `<ejb-ref>` element in the appropriate deployment descriptor.

Note: In EJB 3.0, an environment reference to a target EJB is not needed. You can access a target EJB directly using annotations and resource injection (see ["Accessing an EJB 3.0 EJB"](#) on page 29-4).

For information on looking up a target EJB, see ["Accessing an EJB from a Client"](#) on page 29-1.

To define a reference to the remote interface of an EJB 2.1 EJB:

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).
2. Within the `<ejb-ref>` element, define the `<home>` and `<remote>` sub-elements with the package and class name of the target EJB remote home and remote component interface, respectively.
3. Within the `<ejb-ref>` element, define the `<ejb-ref-type>` to the target bean's type: `Session` or `Entity`.
4. Within the `<ejb-ref>` element, define the `<ejb-ref-name>` and, optionally, the `<ejb-link>` sub-elements.

Note: If the bean interfaces are unique (for example, only one session bean uses the interface `Cart.class`) then the `<ejb-link>` is not required.

You can choose one of the following options:

- a. Configure `<ejb-ref-name>` with a logical name and configure `<ejb-link>` with the actual name of the target bean as [Example 19-1](#) shows.

This option provides indirection that offers assembly and deployment flexibility.

Example 19-1 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>
```

- b. Configure `<ejb-ref-name>` with a logical name as [Example 19-2](#) shows and, in 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 19-3](#) shows.

This option provides indirection that offers the most assembly and deployment flexibility.

Example 19-2 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>
```

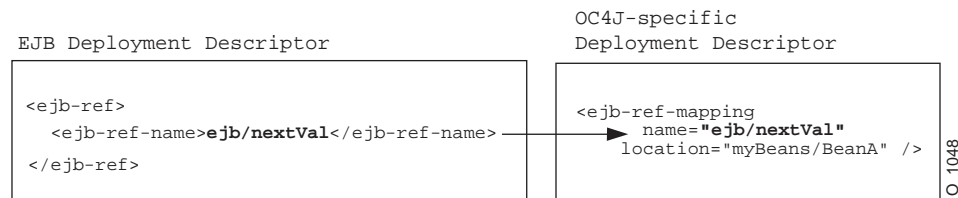
As [Figure 19-1](#) shows, in the `<ejb-ref-mapping>` element, configure the name attribute to match the `<ejb-ref-name>` and configure the location

attribute with the actual name of the target bean. In [Example 19-3](#), the logical name `ejb/nextVal` is mapped to the actual name of the target bean `myBeans/BeanA`.

Example 19-3 Mapping Logical Name to Actual Name with `ejb-ref-mapping`

```
<ejb-ref-mapping name="ejb/nextVal" location="myBeans/BeanA" />
```

Figure 19-1 EJB Reference 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 Local EJB

Before you can look up a target EJB from your client, you must define an EJB reference to the target EJB. To define an EJB reference to the local interface of a target EJB, you configure an `<ejb-local-ref>` element in the `ejb-jar.xml` deployment descriptor.

Note: In EJB 3.0, an environment reference to a target EJB is not needed. You can access a target EJB directly using annotations and resource injection (see ["Accessing an EJB 3.0 EJB"](#) on page 29-4).

For information on looking up a target EJB, see ["Accessing an EJB from a Client"](#) on page 29-1.

To define a reference to the local interface of an EJB:

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).
2. Within the `<ejb-local-ref>` element, define the `<local-home>` and `<local>` sub-elements with the package and class name of the target EJB remote home and remote component interface, respectively.
3. Within the `<ejb-local-ref>` element, define the `<ejb-ref-type>` to the target bean's type: `Session` or `Entity`.
4. Within the `<ejb-local-ref>` element, define the `<ejb-ref-name>` and, optionally, the `<ejb-link>` sub-elements.

Note: If the bean interfaces are unique (for example, only one session bean uses the interface `Cart.class`) then the `<ejb-link>` is not required.

You can choose one of the following options:

- a. Configure `<ejb-ref-name>` with a logical name and configure `<ejb-link>` with the actual name of the target bean as [Example 19–1](#) shows.

This option provides indirection that offers assembly and deployment flexibility.

Example 19–4 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</local-home>
  <local>myBeans.BeanA</local>
  <ejb-link>myBeans/BeanA</ejb-link>
</ejb-local-ref>
```

- b. Configure `<ejb-ref-name>` with a logical name as [Example 19–2](#) shows and, in 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 19–3](#) shows.

This option provides indirection that offers the most assembly and deployment flexibility.

Example 19–5 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</local-home>
  <local>myBeans.BeanA</local>
</ejb-local-ref>
```

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 19–3](#), the logical name `ejb/nextVal` is mapped to the actual name of the target bean `myBeans/BeanA`.

Example 19–6 Mapping Logical Name to Actual Name with `ejb-ref-mapping`

```
<ejb-ref-mapping name="ejb/nextVal" location="myBeans/BeanA" />
```

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`.

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 19-22).

For information on looking up a resource manager connection factory, see:

- ["Looking Up an EJB 3.0 Resource Manager Connection Factory"](#) on page 19-22
- ["Looking Up an EJB 2.1 Resource Manager Connection Factory"](#) on page 19-26

To define a reference to a JDBC DataSource:

1. 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 its `<res-ref-name>` element with a logical name for the JDBC data source resource manager connection factory.

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 19-43](#) on page 19-27), always prefix the logical name with "java:comp/env/".

[Figure 19-1](#) shows a `<resource-ref>` element in the `ejb-jar.xml` deployment descriptor with a logical name of `jdbc/OrderDB`, of type `javax.sql.DataSource`, and the authenticator of `Application`.

2. In the `data-sources.xml` file, define the desired DataSource and specify its actual JNDI name (see ["Configuring Data Sources"](#) on page 20-1).

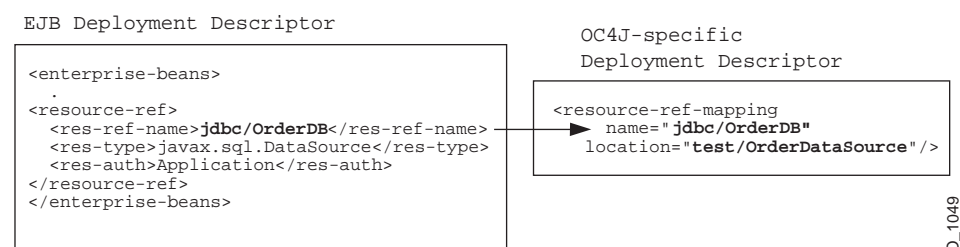
In this example, assume a DataSource is specified in the `data-sources.xml` file with the JNDI name of `/test/OrderDataSource`.

3. In the `orion-ejb-jar.xml` deployment descriptor, define a `<resource-ref-mapping>` element to map the logical name (defined in `ejb-jar.xml`) to the JNDI name (defined in `data-sources.xml`).

[Figure 19-2](#) 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`).

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 19-43](#) on page 19-27).

Figure 19-2 Mapping Logical to Actual JDBC Data Source Resource Manager Connection Factory



Configuring an Environment Reference to a JMS Destination or Connection Resource Manager Connection Factory

You can access a JMS destination (queue or topic) and JMS connection resource manager connection factory by creating an environment reference to them.

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 19-22).

For information on looking up a resource manager connection factory, see:

- ["Looking Up an EJB 3.0 Resource Manager Connection Factory"](#) on page 19-22
- ["Looking Up an EJB 2.1 Resource Manager Connection Factory"](#) on page 19-26

To define a reference to a JMS destination and JMS connection resource manager connection factory:

1. Configure your JMS service provider.

For more information, see:

- ["Configuring an OracleAS JMS Message Service Provider"](#) on page 23-1
- ["Configuring an OJMS Message Service Provider"](#) on page 23-3
- ["Configuring a Message Service Provider Using J2CA"](#) on page 23-7

2. Define the JNDI name for the JMS destination and connection factory.

For more information, see:

- ["OracleAS JMS Destination and Connection Factory Names"](#) on page 23-2
- ["OJMS Destination and Connection Factory Names"](#) on page 23-3
- ["J2CA Message Service Provider Connection Factory Names"](#) on page 23-8

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 sub-elements:

- `<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 19-7](#) shows a `<resource-env-ref>` element for a JMS topic resource manager connection factory.

Example 19-7 `<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 sub-elements:

- `<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 19–8](#) shows a `<resource-ref>` element for a JMS topic connection resource manager connection factory.

Example 19–8 `<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.

- 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 19–9](#) shows a `<resource-env-ref-mapping>` element for OracleAS JMS.

Example 19–9 OracleAS JMS `<resource-env-ref-mapping>`

```
<resource-env-ref-mapping
  name="rpTestTopic"
  location="jms/Topic/rpTestTopic">
</resource-env-ref-mapping>
```

- 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 19–10](#) shows a `<resource-ref-mapping>` element for OracleAS JMS.

Example 19–10 OracleAS JMS `<resource-ref-mapping>`

```
<resource-ref-mapping
  name="myTCF"
  location="jms/Topic/myTCF">
</resource-ref-mapping>
```

Configuring an Environment Reference to a Java Mail Resource Manager Connection Factory

You can access a Java mail session by creating a resource manager connection factory reference to it.

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 19-22).

For information on looking up a resource manager connection factory, see:

- ["Looking Up an EJB 3.0 Resource Manager Connection Factory"](#) on page 19-22
- ["Looking Up an EJB 2.1 Resource Manager Connection Factory"](#) on page 19-26

To define a reference to a Java mail resource manager connection factory:

1. Bind the `javax.mail.Session` reference within the JNDI name space in the `application.xml` file using the `<mail-session>` element, as follows:

```
<mail-session location="mail/MailSession"
  smtp-host="mysmtp.oraclecorp.com">
  <property name="mail.transport.protocol" value="smtp"/>
  <property name="mail.smtp.from" value="emailaddress@oracle.com"/>
</mail-session>
```

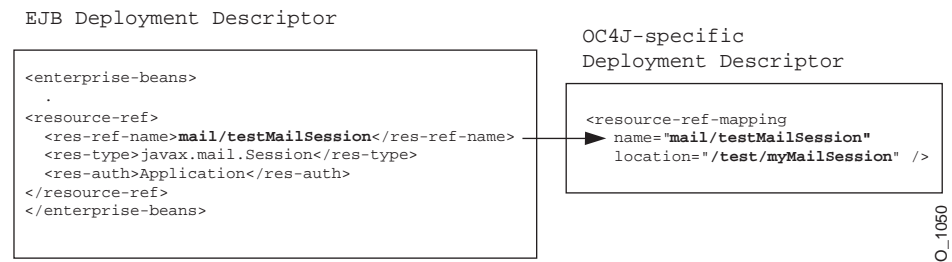
The location attribute contains the JNDI name specified in the location attribute of the `<resource-ref-mapping>` element in the OC4J-specific deployment descriptor.

2. Create a logical name within the `<res-ref-name>` element in the EJB deployment descriptor.

It is a best practice to start the reference name with "mail" but it is not required. In the bean code, the lookup of this reference is always prefaced by "java:comp/env" (for example, "java:comp/env/mail/myMail")

3. Map the logical name within the EJB deployment descriptor to the JNDI name, created in step 1, within the OC4J-specific deployment descriptor.
4. Lookup the object reference within the bean with the "java:comp/env/mail" preface and the logical name defined in the EJB deployment descriptor.

As shown in [Figure 19-3](#), the `Session` object was bound to the JNDI name `/test/myMailSession`. The logical name that the bean knows this resource as is `mail/testMailSession`. These names are mapped together within the OC4J-specific deployment descriptor. Thus, within the bean's implementation, the bean can retrieve the connection to the bound `Session` object by using the `"java:comp/env/mail/testMailSession"` environment element.

Figure 19–3 Session Resource Manager Mapping

This environment element is defined with the following information:

Element	Description
<res-ref-name>	The logical name of the Session object to be used within the originating bean. The name should be prefixed with "mail/". In our example, the logical name for our mail session is "mail/testMailSession".
<res-type>	The Java type of the resource. For the Java mail Session object, this is javax.mail.Session.
<res-auth>	Define who is responsible for signing on to the database. The value can be "Application" or "Container" based on who provides the authentication information.

Example 19–11 Defining an environment element for Java mail Session

The environment element is defined within the EJB deployment descriptor by providing the logical name, "mail/testMailSession", its type of javax.mail.Session, and the authenticator of "Application".

```

EJB
Deployment
Descriptor
<resource-ref>
  <res-ref-name>mail/testMailSession</res-ref-name>
  <res-type>javax.mail.Session</res-type>
  <res-auth>Application</res-auth>
</resource-ref>

```

The environment element of "mail/testMailSession" is mapped to the JNDI bound name for the connection, "test/myMailSession" within the OC4J-specific deployment descriptor.

```

OC4J-specific
Deployment
Descriptor
<resource-ref-mapping
  name="mail/testMailSession"
  location="/test/myMailSession" />

```

Once deployed, the bean can retrieve the Session object reference:

```

InitialContext ic = new InitialContext();
Session session = (Session) ic.lookup("java:comp/env/mail/testMailSession");

//The following uses the mail session object
//Create a message object
MimeMessage msg = new MimeMessage(session);

//Construct an address array
String mailTo = "whosit@oracle.com";
InternetAddress addr = new InternetAddress(mailTo);
InternetAddress addrs[] = new InternetAddress[1];
addrs[0] = addr;

```

```
//set the message parameters
msg.setRecipients(Message.RecipientType.TO, addrs);
msg.setSubject("testSend()" + new Date());
msg.setContent(msgText, "text/plain");

//send the mail message
Transport.send(msg);
```

Configuring an Environment Reference to a URL Resource Manager Connection Factory

You can access a URL connection by creating a resource manager connection factory reference to it.

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 19-22).

For information on looking up a resource manager connection factory, see:

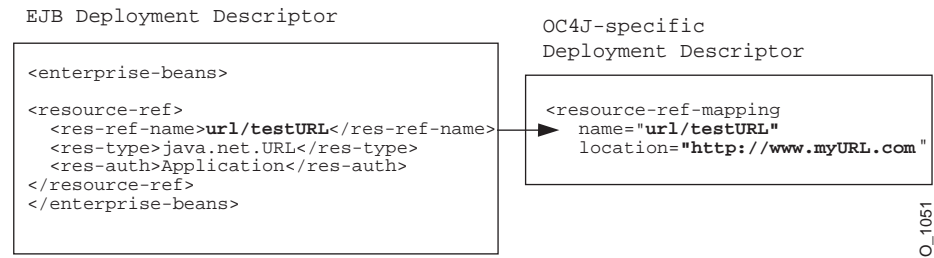
- ["Looking Up an EJB 3.0 Resource Manager Connection Factory"](#) on page 19-22
- ["Looking Up an EJB 2.1 Resource Manager Connection Factory"](#) on page 19-26

To define a reference to a URL resource manager connection factory:

1. Create a logical name within the `<res-ref-name>` element in the EJB deployment descriptor.

It is a best practice to start the reference name with "url" but it is not required. In the bean code, the lookup of this reference is always prefaced by "java:comp/env" (for example, "java:comp/env/url/myURL")
2. Map the logical name within the EJB deployment descriptor to the URL within the OC4J-specific deployment descriptor.
3. Lookup the object reference within the bean with the "java:comp/env/url" preface and the logical name defined in the EJB deployment descriptor.

As shown in [Figure 19-4](#), the URL object was bound to the URL "http://www.myURL.com". The logical name that the bean knows this resource as is "url/testURL". These names are mapped together within the OC4J-specific deployment descriptor. Thus, within the bean's implementation, the bean can retrieve a reference to the URL object by using the "java:comp/env/url/testURL" environment element.

Figure 19–4 URL Resource Manager Mapping

This environment element is defined with the following information:

Element	Description
<res-ref-name>	The logical name of the URL object to be used within the originating bean. The name should be prefixed with "url/". In our example, the logical name for our URL is "url/testURL".
<res-type>	The Java type of the resource. For the Java URL object, this is <code>java.net.URL</code> .
<res-auth>	Define who is responsible for signing on to the database. At this time, the only value supported is "Application". The application provides the authentication information.

Example 19–12 Defining an Environment Element for a URL

The environment element is defined within the EJB deployment descriptor by providing the logical name, "url/testURL", its type of `java.net.URL`, and the authenticator of "Application".

```

EJB Deployment Descriptor
<resource-ref>
  <res-ref-name>url/testURL</res-ref-name>
  <res-type>java.net.URL</res-type>
  <res-auth>Application</res-auth>
</resource-ref>

```

The environment element of "url/testURL" is mapped to the URL "http://www.myURL.com" within the OC4J-specific deployment descriptor.

```

OC4J-specific Deployment Descriptor
<resource-ref-mapping
  name="url/testURL"
  location="http://www.myURL.com" />

```

Once deployed, the bean can retrieve the URL object reference as follows:

```

InitialContext ic = new InitialContext();
URL url = (URL) ic.lookup("java:comp/env/url/testURL");

//The following uses the URL object
URLConnection conn = url.openConnection();

```

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>` sub-element, the type is defined in the

`<env-entry-type>` sub-element, and the value is defined in the `<env-entry-value>` sub-element. The `<env-entry-name>` is relative to the "java:comp/env" context.

[Example 19-13](#) shows how to define environment variables for `java:comp/env/minBalance` and `java:comp/env/maxCreditBalance` in the `ejb-jar.xml` file.

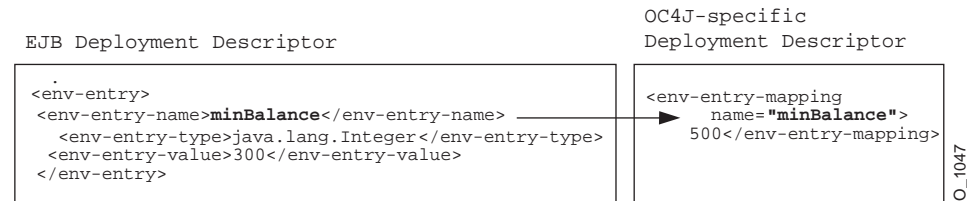
Example 19-13 *ejb-jar.xml For Environment Variables*

```
<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 19-5](#) shows how the `minBalance` environment variable value is overridden by the `orion-ejb-jar.xml` file and set to 500.

Figure 19-5 *Overriding Environment Variables in ejb-jar.xml with orion-ejb-jar.xml*



For more information on looking up environment variables, see:

["Looking Up an EJB 3.0 Environment Variable"](#) on page 19-23

["Looking Up an EJB 2.1 Environment Variable"](#) on page 19-27

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:

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 sub-elements:

- `<service-ref-name>`: a logical name for the web service.
- `<service-interface>`: the destination class type; either `javax.jms.Queue` or `javax.jms.Topic`.

[Example 19-14](#) 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 30-1](#) on page 30-1) is always prefaced by "java:comp/env" (for example, "java:comp/env/service/myService")

Example 19-14 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>` sub-element.

[Example 19-9](#) shows a `<resource-env-ref-mapping>` element for a web service.

Example 19-15 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 a Stateless Session Bean with a Web Service"](#) on page 30-1.

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 EJB, 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 you are using it in 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 EJB are collocated	Default (see "Configuring the Default Initial Context Factory" on page 19-16)
Any Client	Client and target EJB are deployed in the same application	Default (see "Configuring the Default Initial Context Factory" on page 19-16)
Any Client	Target EJB deployed in an application that is designated as the client's parent ¹	Default (see "Configuring the Default Initial Context Factory" on page 19-16)
EJB Client Servlet or JSP Client	Client and target EJB are not collocated, not deployed in the same application, and target EJB application is not client's parent ¹ .	<code>oracle.j2ee.rmi.RMIInitialContextFactory</code> (see "Configuring an Oracle Initial Context Factory" on page 19-16)
Stand-alone Java Client	Client and target EJB are not collocated, not deployed in the same application, and target EJB application is not client's parent ¹ .	<code>oracle.j2ee.naming.ApplicationClientInitialContextFactory</code> see "Configuring an Oracle Initial Context Factory" on page 19-16)

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:

- *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 EJB is simple: no JNDI properties are required.

Example 19–16 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 EJB"](#) on page 19-18.

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 19-17

- ["Configuring the Naming Provider URL for OC4J Standalone"](#) on page 19-18
- 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 19-17](#) shows.

Example 19-17 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 19-18](#) shows.

Example 19-18 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 19-22
 - [Looking Up an EJB 3.0 Environment Variable](#) on page 19-23
 - [Looking Up an EJB 2.1 Resource Manager Connection Factory](#) on page 19-26
 - [Looking Up an EJB 2.1 Environment Variable](#) on page 19-27
 - ["Accessing an EJB 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="6003" remote="6200" request="6004"/>
  ...
</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 hostname is `dpanda-us`, request port is 6004, and instances name is `home1`, then the provider URL would be:

```
opmn:ormi://dpanda-us:6003:home1/ejbsamples
```

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 hostname is `dpanda-us`, ORMI port is 23793, and the application name is `ejb30slsb`, then the provider URL would be:

```
ormi://dpanda-us:23793/ejb30slsb
```

Setting JNDI Properties in an EJB

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:

- [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:

- ["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`.

If setting the JNDI properties within the `jndi.properties` file, make sure that this file is accessible from the client CLASSPATH.

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 22-10.

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 19-13).

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 initial context constructor.

For an example, see ["Specifying Credentials in the Initial Context"](#) on page 22-11.

Looking up an EJB 3.0 EJB

Using EJB 3.0, you can look up an EJB using resource injection (see ["Using Annotations"](#) on page 19-19) or the `InitialContext` (see ["Using Initial Context"](#) on page 19-20).

Using Annotations

[Example 19-19](#) shows how to use annotations and dependency injection to access an EJB 3.0 EJB from an EJB client.

Example 19-19 Injecting an EJB 3.0 EJB in an EJB 3.0 EJB Client

```
@EJB AdminService bean;

public void privilegedTask()
{
    bean.adminTask();
}
```

[Example 19-20](#) shows how to use annotations and dependency injection to access an EJB 2.1 EJB from an EJB 3.0 EJB client.

Example 19-20 Injecting an EJB 2.1 EJB in an EJB 3.0 EJB Client

```
@EJB(
    name="ejb/shopping-cart", // optional
    beanName="cart1", // optional
    beanInterface=ShoppingCartHome.class, // optional
    description="The shopping cart for this application" //optional
)
```

```
private ShoppingCartHome myCartHome;
```

Using Initial Context

This section describes:

- [Looking Up the Remote Interface of an EJB 3.0 EJB Using ejb-ref](#)
- [Looking Up the Remote Interface of an EJB 3.0 EJB Using location](#)
- [Looking up the Local Interface of an EJB 3.0 EJB Using local-ref](#)
- [Looking up the Local Interface of an EJB 3.0 EJB Using local-location](#)

For more information, see ["Configuring the Initial Context Factory"](#) on page 19-15.

Looking Up the Remote Interface of an EJB 3.0 EJB Using ejb-ref

To look up the remote interface of an EJB using an `ejb-ref`:

1. Define an `ejb-ref` for the EJB in the `ejb-jar.xml` file.

Example 19-21 *ejb-jar.xml For an ejb-ref*

```
<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"](#) on page 19-3).

2. Determine whether or not a prefix is required (see ["Selecting an EJB Reference"](#) on page 29-3).
3. Look up the EJB using the `ejb-ref-name` and the appropriate prefix (if required).

Example 19-22 *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 19-15.

Looking Up the Remote Interface of an EJB 3.0 EJB Using location

To look up the remote interface of an EJB using its location:

1. Define the `entity-deployment` attribute location in the `orion-ejb-jar.xml` file.

Example 19-23 *orion-ejb-jar.xml For location*

```
<entity-deployment
  name="Test"
  location="app/Test"
  ...
>
...
</entity-deployment>
```

The default value for `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 29-3).
3. Look up the EJB using the `location`.

Example 19–24 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 19-15.

Looking up the Local Interface of an EJB 3.0 EJB Using `local-ref`

To look up the remote interface of an EJB using an `ejb-local-ref`:

1. Define an `ejb-local-ref` for the EJB in the `ejb-jar.xml` file.

Example 19–25 `ejb-jar.xml` For an `ejb-local-ref`

```
<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 19-5).

2. Determine whether or not a prefix is required (see ["Selecting an EJB Reference"](#) on page 29-3).
3. Look up the EJB using the `ejb-ref-name` and the appropriate prefix (if required).

Example 19–26 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 19-15.

Looking up the Local Interface of an EJB 3.0 EJB Using `local-location`

To look up the local interface of an EJB using its `local-location`:

1. Define the `entity-deployment` attribute `local-location` in the `orion-ejb-jar.xml` file.

Example 19–27 `orion-ejb-jar.xml` For `local-location`

```
<entity-deployment
  name="Test"
  local-location="app/Test"
  ...
>
...
</entity-deployment>
```

The default value for `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 29-3).
3. Look up the EJB using the `local-location`.

Example 19–28 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 19-15.

Looking Up an EJB 3.0 Resource Manager Connection Factory

Using EJB 3.0, you can look up a resource manager connection using resource injection (see ["Using Annotations"](#) on page 19-22) or the `InitialContext` (see ["Using Initial Context"](#) on page 19-22).

Using Annotations

[Example 19–29](#) shows how to use annotations and dependency injection to access an EJB 3.0 resource manager connection factory.

Example 19–29 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 19–30](#) shows how to use the initial context to look up an EJB 3.0 resource manager connection factory.

Example 19–30 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 19-15.

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 19-24).

Using Resource Injection

Using resource injection, you can rely on the container to initialize a field or a setter method (property) using either the:

- 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 19–31](#)) or the set-method (property) level as [Example 19–32](#) shows.

Example 19–31 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 19–32 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 19–33](#) shows.

Example 19–33 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 19–34](#) shows how you look up these environment variables within the bean's code using the `InitialContext`.

Example 19–34 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 19-15.

Looking Up an EJB 2.1 EJB

Using EJB 2.1, you can look up an EJB using the `InitialContext` (see ["Using Initial Context"](#) on page 19-24).

To look up an EJB 3.0 EJB using the `InitialContext`, use the same approach.

Using Initial Context

This section describes:

- [Looking Up the Remote Interface of an EJB 2.1 EJB Using ejb-ref](#)
- [Looking Up the Remote Interface of an EJB 2.1 EJB Using location](#)
- [Looking up the Local Interface of an EJB 2.1 EJB Using local-ref](#)
- [Looking up the Local Interface of an EJB 2.1 EJB Using local-location](#)

For more information, see ["Configuring the Initial Context Factory"](#) on page 19-15.

Looking Up the Remote Interface of an EJB 2.1 EJB Using ejb-ref

To look up the remote interface of an EJB using an `ejb-ref`:

1. Define an `ejb-ref` for the EJB in the `ejb-jar.xml` file.

Example 19–35 ejb-jar.xml For an ejb-ref

```
<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"](#) on page 19-3).

2. Determine whether or not a prefix is required (see ["Selecting an EJB Reference"](#) on page 29-3).
3. Look up the EJB using the `ejb-ref-name` and the appropriate prefix (if required).

Example 19-36 Looking Up Using ejb-ref

```

InitialContext ic = new InitialContext();
Object homeObject = context.lookup("java:comp/env/ejb/Test");
CartHome home = (CartHome)PortableRemoteObject.narrow(homeObject, CartHome.class);

```

For more information, see ["Configuring the Initial Context Factory"](#) on page 19-15.

Looking Up the Remote Interface of an EJB 2.1 EJB Using location

To look up the remote interface of an EJB using its `location`:

1. Define the `entity-deployment` attribute `location` in the `orion-ejb-jar.xml` file.

Example 19-37 orion-ejb-jar.xml For location

```

<entity-deployment
    name=EmployeeBean"
    location="app/EmployeeBean"
    ...
>
...
</entity-deployment>

```

The default value for `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 29-3).
3. Look up the EJB using the `location`.

Example 19-38 Looking Up Using location

```

InitialContext ic = new InitialContext();
Object homeObject = context.lookup("java:comp/env/app/EmployeeBean");
CartHome home = (CartHome)PortableRemoteObject.narrow(homeObject, CartHome.class);

```

For more information, see ["Configuring the Initial Context Factory"](#) on page 19-15.

Looking up the Local Interface of an EJB 2.1 EJB Using local-ref

To look up the remote interface of an EJB using an `ejb-local-ref`:

1. Define an `ejb-local-ref` for the EJB in the `ejb-jar.xml` file.

Example 19-39 ejb-jar.xml For an ejb-local-ref

```

<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 19-5).

2. Determine whether or not a prefix is required (see ["Selecting an EJB Reference"](#) on page 29-3).
3. Look up the EJB using the `ejb-ref-name` and the appropriate prefix (if required).

Example 19-40 Looking Up

```
InitialContext ic = new InitialContext();
Object homeObject = context.lookup("java:comp/env/ejb/Test");
CartHome home = (CartHome)PortableRemoteObject.narrow(homeObject, CartHome.class);
```

For more information, see ["Configuring the Initial Context Factory"](#) on page 19-15.

Looking up the Local Interface of an EJB 2.1 EJB Using local-location

To look up the local interface of an EJB using its `local-location`:

1. Define the `entity-deployment` attribute `local-location` in the `orion-ejb-jar.xml` file.

Example 19-41 orion-ejb-jar.xml For local-location

```
<entity-deployment
  name=EmployeeBean"
  local-location="app/EmployeeBean"
  ...
>
...
</entity-deployment>
```

The default value for `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 29-3).
3. Look up the EJB using the `local-location`.

Example 19-42 Looking Up Using local-location

```
InitialContext ic = new InitialContext();
Object homeObject = context.lookup("java:comp/env/app/EmployeeBean");
CartHome home = (CartHome)PortableRemoteObject.narrow(homeObject, CartHome.class);
```

For more information, see ["Configuring the Initial Context Factory"](#) on page 19-15.

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 19-27).

For more information on configuring resources, see ["Resource Manager Connection Factory Environment References"](#) on page 19-2.

Using Initial Context

[Example 19-43](#) 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 19-6) prefixed with `java:comp/env/jdbc` prefix.

Example 19-43 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 19-15.

Looking Up an EJB 2.1 Environment Variable

Using EJB 2.1, you can look up an environment variable using the `InitialContext` (see ["Using Initial Context"](#) on page 19-27).

For more information on configuring environment variables, see ["Configuring an Environment Reference to an Environment Variable"](#) on page 19-13.

Using Initial Context

[Example 19-34](#) shows how you look up these environment variables within the bean's code using the `InitialContext`.

Example 19-44 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 19-15.

Configuring Data Sources

This chapter describes:

- [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](#)

For more information, see:

- ["Understanding EJB Data Source Services"](#) on page 2-13
- ["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:

- ["What Types of Data Source does OC4J Support?"](#) on page 2-13
- ["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='Example Connection Pool'>
  <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
  connection-pool-name='Example Connection Pool'
  jndi-name='jdbc/COMMONLocalDS'
  name='COMMONLocalDS'
/>
```

For more information, see:

- 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:

- ["What Types of Data Source does OC4J Support?"](#) on page 2-13
- "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>
```

For more information, see:

- 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:

- ["What is a Default Data Source?"](#) on page 2-14
- "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:

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.

For more information, see:

- ["What is the ejb3-toplink-sessions.xml File?"](#) on page 2-9
- ["Customizing the TopLink Entity Manager"](#) on page 3-2

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:

- ["What is a Default Data Source?"](#) on page 2-14
- "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:

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.

Configuring Transaction Services

This chapter describes:

- [Configuring Transaction Timeouts](#)
- [Transaction Best Practices](#)

For more information, see:

- ["Understanding EJB Transaction Services"](#) on page 2-14
- ["Java Transaction API \(JTA\)"](#) in the *Oracle Containers for J2EE Services Guide*

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:

- [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:

- [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 21-1).

Configuring a Transaction Timeout for a Session Bean

You can configure a transaction timeout on a per-session bean basis (see ["Using Deployment XML"](#) on page 21-2). The per-session bean transaction timeout overrides the global transaction timeout (see ["Configuring a Global Transaction Timeout"](#) on page 21-1).

Using Deployment XML

In the `orion-ejb-jar.xml` file you set a per-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 on a per-message-driven bean basis (see ["Using Deployment XML"](#) on page 21-3).

Because the global transaction timeout (see ["Configuring a Global Transaction Timeout"](#) on page 21-1) does not apply to message-driven beans, you must configure transaction timeout on a per-message-driven bean basis 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 Providers Can I use with My MDB?"](#) on page 2-16) affects your transaction timeout options:

- Oracle Application Server JMS (OracleAS JMS): you cannot change the transaction timeout from the default of 86,400 seconds (1 day).
- Oracle JMS (OJMS): you can change the transaction timeout (see ["Non-J2CA Adapter Message Service Provider"](#) on page 21-3).
- J2EE Connector Architecture (J2CA) adapter message provider: you can change the transaction timeout (see ["J2CA Adapter Message Service Provider"](#) on page 21-3).

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 OracleAS JMS or Oracle JMS (OJMS), use the `transaction-timeout` attribute of the `<message-driven-deployment>` element.

For example, if you are using OracleAS JMS or Oracle JMS (OJMS), 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:

- [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 21-1](#) 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 21–2](#) shows. By doing so, you guarantee that only one connection will be held until the container-managed transaction commits.

Example 21–1 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 21–2 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:

- ["What is EJB Context?"](#) on page 1-6
- ["Accessing an EJB 3.0 EJBContext"](#) on page 29-14
- ["Accessing an EJB 2.1 EJBContext"](#) on page 29-21

Configuring Security Services

EJB application security involves two realms: granting permissions if you download into a browser and configuring your application for authentication and authorization. This chapter talks about setting up users, roles, and groups for EJBs.

This chapter explains:

- [Granting Permissions in Browser](#)
- [Authenticating and Authorizing EJB Applications](#)
- [Specifying Credentials in EJB Clients](#)
- [Retrieving Credentials from an EJB Using the JAAS API](#)
- [Configuring EJB 3.0 Security Options](#)

For more information, see:

- ["Understanding EJB Security Services"](#) on page 2-16
- *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";
```

Authenticating and Authorizing EJB Applications

For EJB authentication and authorization, you define the principals under which each method executes by configuring of 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*.

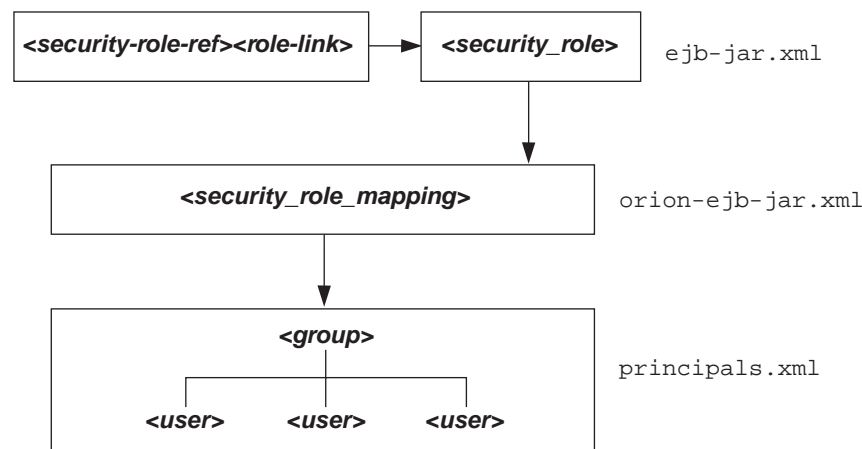
For authentication and authorization, this section focuses on XML configuration within the EJB deployment descriptors. EJB authorization is specified within the EJB and OC4J-specific deployment descriptors. You can manage the authorization piece of your security within the deployment descriptors, as 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



Q_1052

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](#)

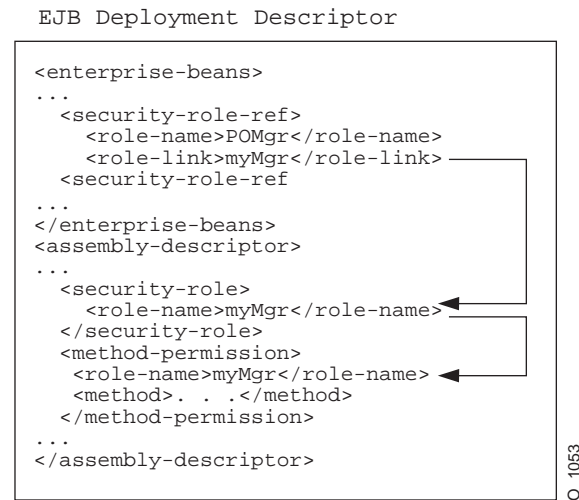
Specifying Users and Groups

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 shown in [Figure 22–2](#), 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 that it applies to. 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 EJB 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 EJB, 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 ["Configuring EJB 3.0 Security Options"](#) 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>

```

```
        </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 ["Configuring EJB 3.0 Security Options"](#) 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 EJB 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 EJB 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 J2EE 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 ["Configuring EJB 3.0 Security Options"](#) 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.

- 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.

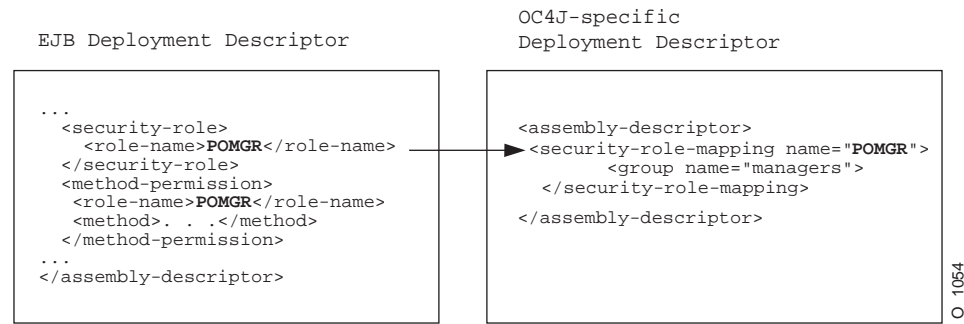
To map this role to a specific user, do the following:

```
<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="<default-ejb-caller-role>"
    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. This attribute defaults to `true`, which states that no security role checking occurs for these methods. If you set this attribute to `false`, the container will check for this default role on 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 EJB or other JNDI-accessible resource.

[Table 22–1](#) classifies EJB clients by where they are deployed relative to the target enterprise JavaBeans they invoke. Where you deploy the client relative to its target enterprise JavaBeans 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 EJB are collocated	No
Any client	Client and target EJB are deployed in the same application	No
Any client	Target EJB deployed in an application that is designated as the client's parent ¹	No
Any client	Client and target EJB 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 EJBs in a *remote* container (that is, if the client and target EJB are not collocated, not deployed in the same application, and the target EJB 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 EJBs (see "[Specifying Credentials in the Initial Context](#)" on page 22-11).
- **Stand-alone Java Client:** define credentials in the `jndi.properties` file deployed with the EAR file (see "[Specifying Credentials in JNDI Properties](#)" on page 22-10).
- **Servlet or JSP Client:** pass credentials within the `InitialContext`, which is created to look up the remote EJBs (see "[Specifying Credentials in the Initial Context](#)" on page 22-11).

For more information, see:

- "[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:

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 runtime, 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 19-18.

Specifying Credentials in the Initial Context

To specify credentials in the initial context you use to look up JNDI-accessible resources:

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:

- ["Configuring the Initial Context Factory"](#) on page 19-15
- ["Setting JNDI Properties in the Initial Context"](#) on page 19-19

Retrieving Credentials from an EJB Using the JAAS API

OC4J supports the use of standard JAAS API to retrieve the `Subject`, `Principal`, and credentials from within business methods and lifecycle methods of session beans (stateless and stateful) and entity beans.

[Example 22–9](#) shows how you can use the JAAS API to retrieve credentials in a business method of an EJB deployed to OC4J.

Example 22–9 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();
        ...
    }
}
```

Configuring EJB 3.0 Security Options

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
<code>@RunAs</code>	Defines the role of the application during execution in a J2EE 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
<code>@RolesAllowed</code>	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.
<code>@PermitAll</code>	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.
<code>@DenyAll</code>	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.
<code>@DeclareRoles</code>	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 22–10](#) shows how to use the `@RolesAllowed` annotation. For more information and examples, see the JSR250 specification.

Example 22–10 `@RolesAllowed`

```
@RolesAllowed("Users")
public class Calculator
{
    @RolesAllowed("Administrator")
    public void setNewRate(int rate)
    {
        ...
    }
}
```

Configuring Message Services

This chapter describes how to configure Java Message Service (JMS) and non-JMS message service providers, including:

- [Configuring an OracleAS JMS Message Service Provider](#)
- [Configuring an OJMS Message Service Provider](#)
- [Configuring a Message Service Provider Using J2CA](#)

For more information, see:

- ["What Message Providers Can I use with My MDB?"](#) on page 2-16
- ["Implementing an EJB 3.0 MDB"](#) on page 9-1
- ["Implementing an EJB 2.1 MDB"](#) on page 17-1
- ["Java Message Service"](#) in the *Oracle Containers for J2EE Services Guide*

Configuring an OracleAS JMS Message Service Provider

To configure the OracleAS JMS message service provider, you must:

1. Choose appropriate JNDI names for your destination and connection factory (see ["OracleAS JMS Destination and Connection Factory Names"](#)).
2. Configure the `<OC4J_HOME>/j2ee/home/config/jms.xml` file (see ["Configuring jms.xml"](#)).
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"](#) on page 19-7).
4. Associate the OracleAS JMS message service provider with the message-driven beans that will use it.

For more information, see:

- ["Configuring an EJB 3.0 MDB to Use a Non-J2CA Message Service Provider"](#) on page 10-2
- ["Configuring an EJB 2.1 MDB to Use a Non-J2CA Message Service Provider"](#) on page 18-1

For more information about OracleAS JMS, see ["Oracle Application Server JMS \(OracleAS JMS\) Provider: File-Based"](#) on page 2-17.

OracleAS 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 23-2).

[Table 23–1](#) lists the form of these names.

Table 23–1 OracleAS JMS Destination and Connection Factory Names

Type	Form
Queue	<code>jms/Queue/<QName></code>
Queue Connection Factory	<code>jms/Queue/<QCFName></code>
Topic	<code>jms/Topic/<TName></code>
Topic Connection Factory	<code>jms/Topic/<TCFName></code>

Configuring `jms.xml`

You configure OracleAS JMS options in the `<OC4J_HOME>/j2ee/home/config/jms.xml` file. In 10.1.3, 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:

- 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.
- 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 OracleAS 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 OracleAS JMS

```
<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>
```

Configuring an OJMS Message Service Provider

To configure the OJMS message service provider, you must:

1. Install and configure the OJMS provider (see ["Installing and Configuring the OJMS Provider"](#) on page 23-4).
2. Choose appropriate JNDI names for your destination and connection factory (see ["OracleAS JMS Destination and Connection Factory Names"](#)).
3. Configure the data-sources.xml file to identify your database (see ["Configuring data-sources.xml"](#)).
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"](#) on page 19-7).
5. Associate the OJMS message service provider with the message-driven beans that will use it.

For more information, see:

- ["Configuring an EJB 3.0 MDB to Use a Non-J2CA Message Service Provider"](#) on page 10-2
- ["Configuring an EJB 2.1 MDB to Use a Non-J2CA Message Service Provider"](#) on page 18-1

For more information, see ["Oracle JMS \(OJMS\) Provider: Advanced Queueing \(AQ\)-Based"](#) on page 2-17.

OJMS Destination and Connection Factory Names

The actual JNDI names for the JMS destination and connection factory depend on your OJMS installation as shown in [Table 23-2](#).

Table 23-2 OJMS 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/<ProviderName>/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 OJMS service (see ["Identify the JNDI Name of the Oracle AQ JMS Data Source"](#) on page 23-6)
- <QName>: the name of the queue you created in the database (see step 3 b in ["Installing and Configuring the OJMS Provider"](#) on page 23-4).
- <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 OJMS Provider"](#) on page 23-4).

- <TCFName>: the name of the topic connection factory. You may specify any arbitrary name.

Installing and Configuring the OJMS 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 OJMS 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 two-phase commit or system administration privileges, based on what the user needs. See the JTA chapter in the Oracle Containers for J2EE Services Guide for the two-phase commit privileges.

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 OJMS, 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 OJMS provider is installed. The JMS topics and queues use database tables and queues to facilitate messaging. The type of data source you use depends on the functionality you want.

Transactional Functionality

For no transactions or single-phase transactions, you can use either an emulated or non-emulated data sources. For two-phase commit transaction support, you can use only a non-emulated data source.

Example 23–2 Emulated With OCI JDBC Driver

The following shows an emulated data source that uses the OCI JDBC driver. Thus, it is J2EE 1.3 compliant, which supports session pooling. However, it can only maintain single-phase commit transactions.

The example is displayed in the format of an XML definition; see the *Oracle Containers for J2EE Configuration and Administration Guide* for directions on adding a new data source to the configuration.

```
<data-source
  class="oracle.j2ee.sql.DriverManagerDataSource"
  name="OracleDS"
  location="jdbc/CartEmulatedOracleCoreDS"
  xa-location="jdbc/xa/CartEmulatedOracleXADS"
  ejb-location="jdbc/CartEmulatedDS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="myuser"
  password="mypasswd"
  url="jdbc:oracle:oci8:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=mysun)
    (PORT=5521))(CONNECT_DATA=(SID=orcl)))"
  inactivity-timeout="30"
/>
```

Example 23–3 Emulated Data Source With Thin JDBC Driver

The following example contains an emulated data source that uses the thin JDBC driver. To support a two-phase commit transaction, use a non-emulated data source. For differences between emulated and non-emulated data sources, see the Data Source chapter in the *Oracle Containers for J2EE Services Guide*.

The example is displayed in the format of an XML definition; see the *Oracle Containers for J2EE Configuration and Administration Guide* for directions on adding a new data source to the configuration through the Oracle Enterprise Manager 10g tool.

```
<data-source
  class="oracle.j2ee.sql.DriverManagerDataSource"
  name="OracleDS"
  location="jdbc/emulatedOracleCoreDS"
  xa-location="jdbc/xa/emulatedOracleXADS"
  ejb-location="jdbc/emulatedDS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="jmsuser"
  password="jmsuser"
  url="jdbc:oracle:thin:@myhost.foo.com:1521:mydb"
/>
```

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

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

Identify the JNDI Name of the Oracle AQ JMS Data Source

Identify the JNDI name of the data source that is to be used as the OJMS provider within the `<resource-provider>` element.

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

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

- **class attribute**—The OJMS 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/emulatedDS" is to be used as the OJMS provider. This JNDI name is identified in the `ejb-location` element 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/emulatedDS"></property>
</resource-provider>
```

Configuring a Message Service Provider Using J2CA

To configure the J2CA message service provider, you must:

1. Install and configure the J2CA adapter (see ["Installing and Configuring a J2CA Adapter"](#) on page 23-8).
2. Choose appropriate JNDI names for your connection factory (see ["J2CA Message Service Provider Connection Factory Names"](#) on page 23-8).
3. Configure the appropriate deployment XML files (see ["Configuring OC4J Deployment XML Files"](#) on page 23-8).
4. Associate the J2CA message service provider with the message-driven beans that will use it.

For more information, see:

- ["Configuring an EJB 3.0 MDB to Use a J2CA Message Service Provider"](#) on page 10-3
- ["Configuring an EJB 2.1 MDB to Use a J2CA Message Service Provider"](#) on page 18-2

For more information, see ["J2EE Connector Architecture \(J2CA\) Adapter Message Provider"](#) on page 2-18.

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 Deployment XML Files"](#) on page 23-8).

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 OracleAS JMS and OJMS message service providers, as well as non-Oracle JMS providers such as WebSphereMQ, Tibco, and SonicMQ.

For more information, see "Overview: Administering Resource Adapters" in the *Oracle Containers for J2EE Resource Adapter Administrator's Guide*.

Configuring OC4J 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`

For more information, see:

- "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 OC4J EJB Application Clustering Services

This chapter describes the OC4J application clustering options you can configure for your EJB application, including:

- [Configuring EJB 3.0 and EJB 2.1 Stateful Session Bean Replication Policy](#)
- [Configuring Replication-Based Load Balancing](#)
- [Configuring Static Retrieval Load Balancing](#)
- [Configuring DNS Load Balancing](#)

For more information, see ["Understanding OC4J EJB Application Clustering Services"](#) on page 2-20

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:

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. Deploy your EJB to any one of the nodes in the cluster.

For more information, see ["State Replication"](#) on page 2-20.

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:

- `onRequestEnd` — replicate at the end of each EJB method call
- `onShutdown` — replicate when the JVM is terminating normally

The `scope` attribute is always set to `allAttributes` for a stateful session bean.

For more information, see ["State Replication Trigger"](#) on page 2-21.

Table 24–1 *Deployment XML Files for Replication Policy Configuration*

Scope	Affected Components	Deployment XML File	See also ...
Global	Web and EJB	application.xml	"Configuring Global Replication Policy in the application.xml File for Web and EJB Components" on page 24-2
Application-level	Web and EJB	orion-application.xml	"Configuring Application-Level Replication Policy in the orion-application.xml File for Web and EJB Components" on page 24-2
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

Configuring Global Replication Policy in the application.xml File for Web and EJB Components

When you configure the `application.xml` file with a state replication policy (see [Example 24–1](#)), it applies to all Web components and to all stateless session beans deployed to this instance of OC4J.

Example 24–1 *The application.xml For a Global Replication Policy*

```
<application>
...
    <replication-policy
        trigger="onRequestEnd"
        scope="allAttributes"
    />
...
</application>
```

Configuring Application-Level Replication Policy in the orion-application.xml File for Web and EJB Components

When you configure the `orion-application.xml` file with a state replication policy (see [Example 24–2](#)), it applies to all Web components and to all stateless session beans deployed to this instance of OC4J.

Example 24–2 *The orion-application.xml For an Application-Level Replication Policy*

```
<orion-application>
...
    <replication-policy
        trigger="onShutdown"
        scope="allAttributes"
    />
...
</orion-application>
```

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 24–3](#)), each bean can use a different type of replication independent of the Web component replication type.

Example 24–3 The orion-ejb-jar.xml For an Application-Level Replication Policy for EJBs

```

<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 Replication-Based Load Balancing

For both EJB 3.0 and EJB 2.1, to configure how a client's requests are load balanced across the OC4J instances in your cluster when you configure a replication policy (see ["Using System Properties"](#) on page 24-3).

For more information, see ["Replication-Based Load Balancing"](#) on page 2-22.

Using System Properties

In this release, configure the `oracle.j2ee.rmi.loadBalance` system property within each client 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.

Configuring Static Retrieval Load Balancing

To use static retrieval of OC4J instances for load balancing, 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`, use the `ormi://` prefix and a comma separated list of OC4J nodes in the form
`<hostname>:<port>/<application-name>`

For more information, see ["Static Retrieval Load Balancing"](#) on page 2-22.

Using JNDI Properties

[Example 24–4](#) 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 24–4 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:

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`, use the `ormi://` prefix, the single host name to which the OC4J IP addresses are mapped, and the common RMI port.

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 ["DNS Load Balancing"](#) on page 2-22.

Using JNDI Properties

In [Example 24–5](#), 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 24–5 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 Timer Services

This chapter describes:

- [Configuring an EJB 3.0 EJB with a J2EE Timer](#)
- [Configuring an EJB 2.1 EJB with a J2EE Timer](#)
- [Configuring an EJB with an OC4J Cron Timer](#)
- [Troubleshooting Timers](#)

Note: 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 2-23.

Configuring an EJB 3.0 EJB with a J2EE Timer

You can configure a J2EE timer on an EJB 3.0 stateless session bean or message-driven bean.

You can access the timer service using annotations and dependency injection (see ["Using Annotations"](#) on page 25-1), using the initial context API (see ["Using Initial Context"](#) on page 25-2).

You can implement the timeout call back by:

- annotating an existing method (see ["Using Annotations"](#) on page 25-1)
- by implementing the `javax.ejb.TimerObjectInterface`.

Using Annotations

[Example 25-1](#) shows how to use resource injection to acquire a J2EE timer in an EJB 3.0 EJB.

Example 25-1 Using @Resource to Acquire a J2EE Timer in an EJB 3.0 EJB

```
@Stateless public class EmployeeServiceBean implements EmployeeService
{
    ...
    @Resource Timer empDurationTimer;
    ...
}
```

```
    }  
}
```

Using Initial Context

[Example 25–2](#) shows how to use the initial context to look up a J2EE timer in an EJB 3.0 EJB.

Example 25–2 Using Initial Context to Look Up an EJB 3.0 J2EE Timer

```
InitialContext ctx = new InitialContext();  
TimerService ts = ctx.getTimerService();  
Timer myTimer = ts.createTimer(timeout, "EmpDurationTimer");
```

Configuring an EJB 2.1 EJB with a J2EE Timer

You can configure a J2EE timer on an EJB 2.1 stateless session bean, entity bean, or message-driven bean.

The EJB that creates the timer first retrieves the timer service—`TimerService` interface—through the `getTimerService` method of the `EJBContext` interface. From the `TimerService` interface, you can create a timer using one of the four provided `createTimer` methods that allow you to specify the timer as a single-event timer or as an interval timer. The timers are defined in milliseconds, even though most events are for much longer time periods. The expiration of the timer can be defined as a duration or in absolute time. In addition, the bean can pass some information to identify the timer, which must be serializable.

```
TimerService ts = ctx.getTimerService();  
Timer myTimer = ts.createTimer(timeout, "EmpDurationTimer");
```

The timer is created by a bean to designate when a callback method is invoked. The business logic that is to be executed when the timer expires is implemented in a callback method—`ejbTimeout`—within the application bean class. The bean class that uses the timer service must implement the `javax.ejb.TimerObject` interface, which contains the `ejbTimeout` method.

The created timer is associated with the identity of the bean. For entity beans, the `ejbTimeout` is invoked on the bean instance that created the bean; for stateless session beans and MDBs, the `ejbTimeout` method is invoked on any bean instance in the pool.

```
public abstract class EmployeeBean implements EntityBean, TimerObject  
...  
    public void ejbTimeout(Timer timer)  
    {  
        System.out.println("ejbTimeout() called at: " + new  
            Date(System.currentTimeMillis()) + " with info: " + timer.getInfo());  
        return;  
    }  
}
```

Note: There is no guarantee that the timers are executed in any order; therefore, your implementation within the `ejbTimeout` callback must be able to handle the callbacks in any sequence.

The `TimerService` provides the following methods for creating the different types of timers:

```

public interface javax.ejb.TimerService {
    /* After a specified duration*/
    public Timer createTime(long duration, java.io.Serializable info);
    /* At a specified interval */
    public Timer createTime(long initialDuration, long intervalDuration,
                           java.io.Serializable info);
    /* At a certain time */
    public Timer createTime(java.util.Date expiration, java.io.Serializable info);
    /* A certain duration after a specified date and time */
    public Timer createTime(java.util.Date initialExpiration,
                           long intervalDuration, java.io.Serializable info);
    public Collection getTimers();
}

```

The `getTimers` method retrieves all active timers associated with the bean.

Note: Timers and their handles are local objects; therefore, they should not be passed through the bean remote interface.

Configuring an EJB with an OC4J Cron Timer

You can use an OC4J cron timer with:

- EJB 3.0 stateless session beans and message-driven beans
- EJB 2.1 EJBs of any type

You can schedule a timer to execute regularly at specified intervals. In the UNIX world, these are known as cron timers.

The following are 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 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:

- 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.

Example 25–4 *Complex 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
```

You can create cron timers either through the `createTimer` method that takes a `String` with the previous five fields in it—separated by spaces—or with the `createTimer` method that has variables for each field. To create the cron timers, use the following Oracle-specific `createTimer` APIs:

```
EJBTimer createTimer(String cronline, Serializable info) throws
IllegalArgumentException, IllegalStateException;
```

```
EJBTimer createTimer(int minute, int hour, int dayOfMonth, int month, int
dayOfWeek, int year, Serializable info) throws IllegalArgumentException,
IllegalStateException;
```

Create the cron timers in the same manner as other timers by retrieving the extended Oracle-specific timer service, and schedule a cron timer using the `createTimer` method. However, since cron timers are Oracle-specific, you cast the returned object as an `EJBTimerService` object. The following example provides a `String` with the five variables separated by spaces. The timer is scheduled to execute every minute.

```
import oracle.ias.container.timer.EJBTimer;
import oracle.ias.container.timer.EJBTimerService;
...
String cron = "1 * * * *";
EJBTimerService ets = (EJBTimerService) ctx.getTimerService();
EJBTimer et = ets.createTimer(cron, info);
```

You can also provide a class that is to be invoked within the `createTimer` method, as follows:

```
EJBTimer createTimer(String cronline, String className, Serializable info) throws
IllegalArgumentException, IllegalStateException;
```

```
EJBTimer createTimer(int minute, int hour, int dayOfMonth, int month, int
dayOfWeek, String className, Serializable info) throws IllegalArgumentException,
IllegalStateException;
```

For arbitrary Java classes, the `info` variable can be either `null` or be a `String[]` of parameters to pass to the main method of the class.

For example, you can have the `mypackage.MyClass` invoked when the get timer fires:

```
EJBTimerService ets = (EJBTimerService) ctx.getTimerService();
EJBTimer et = ets.createTimer(cron, "mypackage.MyClass", info);
```

You must provide a main method within the `mypackage.MyClass`, which is used as the entry point, as follows:

```
public static void main( String args[] )
```

Troubleshooting Timers

This section describes:

- [How to Retrieve Information About the Timer](#)
- [How to Retrieve a Persisted Timer](#)
- [Executing the Timer Within the Scope of a Transaction](#)

- [What Does a `NoSuchObjectLocalException` Mean with Timers?](#)

How to Retrieve Information About the 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.

How to Retrieve 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.

Executing the Timer Within the Scope of a Transaction

The timer is normally created or cancelled within the scope of a transaction. Thus, the bean normally is configured as being within a transaction. Typically this is configured with `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"](#)

Configuring Deployment Descriptor Files

This chapter describes how to configure the various deployment descriptor files an OC4J application may use, including:

- [Configuring the ejb-jar.xml File](#)
- [Configuring the topink-ejb-jar.xml File](#)
- [Configuring the orion-ejb-jar.xml File](#)
- [Configuring the ejb3-topink-sessions.xml File](#)

For more information, see "[Understanding EJB Deployment Descriptor Files](#)" on page 2-7.

Configuring the ejb-jar.xml File

This section describes:

- [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 2-7.

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 Persistence Manager](#)" on page 3-5). After generation, you can use the TopLink Workbench to customize and re-export this file (see "[Using TopLink Workbench](#)" on page 2-3).

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 2-2.

Configuring the toplink-ejb-jar.xml File

This section describes:

- [Creating toplink-ejb-jar.xml During Migration](#)
- [Creating toplink-ejb-jar.xml with TopLink Workbench](#)

For more information, see:

- ["What is the toplink-ejb-jar.xml File?"](#) on page 2-8
- ["OC4J and the toplink-ejb-jar.xml File"](#) in the *Oracle TopLink Developer's Guide*

Creating toplink-ejb-jar.xml During Migration

EJB 2.1 only; When you migrate an Orion CMP application to TopLink persistence (see ["Migrating to the TopLink Persistence Manager"](#) on page 3-5), 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 re-export (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 Persistence Manager"](#) on page 3-5), 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:

- ["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

When you deploy an EJB application to sOC4J, you must specify any vendor-specific configuration in one of the following ways:

- Package an `orion-ejb-jar.xml` with the desired vendor-specific configuration and deploy.
- After deployment, use the Application Server Control deployment profile to make the vendor-specific configuration.

For more information, see ["What is the orion-ejb-jar.xml File?"](#) on page 2-8.

Configuring the ejb3-toplink-sessions.xml File

This section describes:

- ["Creating ejb3-toplink-sessions.xml with TopLink Workbench"](#)

For more information, see ["What is the ejb3-toplink-sessions.xml File?"](#) on page 2-9.

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:

- "Understanding the TopLink Workbench" in the *Oracle TopLink Developer's Guide*
- "Creating project.xml with TopLink Workbench" in the *Oracle TopLink Developer's Guide*.

Packaging an EJB Application

This section describes:

- [Sharing Classes Between EJB Applications](#)

For more information, see:

- *Oracle Application Server Enterprise Deployment Guide*
- ["Understanding Packaging"](#) on page 2-9

Packaging an Application with Both EJB 3.0 and EJB 2.1 EJBs

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`, two EJB 2.1 entity beans with `ejb-jar.xml`, and three EJB 3.0 session beans with `ejb-jar.xml`, 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 EJBs, you can do one of the following:

- If two EJBs use the same classes, include all classes and the EJBs in the same JAR file. After deployment, both EJBs 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 EJBs 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.

If you receive a `ClassCastException`, 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*.

Handling Out of Memory Exceptions at Runtime

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 Runtime

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.

Deploying an EJB Application to OC4J

This section describes:

- [Deploying a Large EJB Application](#)
- [Deploying Incrementally](#)
- [Troubleshooting Application Deployment](#)

For more information, see:

- *Oracle Application Server Enterprise Deployment Guide*
- ["Understanding Deployment"](#) on page 2-9

Deploying a Large EJB Application

This section describes:

- [Tuning the VM to Avoid Out Of Memory Errors During Deployment](#)
- [Disabling Batch Compilation to Avoid Out Of Memory Errors During Deployment](#)

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 be specified as: `java -Xmx750m -Xms512m -XX:PermSize=128m -XX:MaxPermSize=256m`.

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 Out of Memory error, 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 Out of Memory exceptions 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), you may also attempt to compile in non-batch mode. Although non-batch mode requires less memory, this mode will result in a longer deployment time.

To enable or disable batch compilation, use the `<application>` element or `<orion-application>` 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-application.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.

You can use the `updateEJBModule` command in both the `admin.jar` and `admin_client.jar` utilities to incrementally update a deployed application with one or more EJBs contained within an EJB JAR file.

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 re-starting OC4J.

The incremental redeployment operation will automatically stop the application containing the EJB(s) to be updated, then automatically restart the application when finished.

Note: During redeployment, all idle client connections to the EJB 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 EJB.

The general procedure for using incremental deployment is:

1. Deploy an application with a large number of enterprise JavaBeans.
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`.

[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*.

Troubleshooting Application Deployment

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).

For more information, see ["Troubleshooting an EJB Application"](#) on page 31-5.

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:

- [Chapter 29, "Accessing an EJB from a Client"](#)
- [Chapter 30, "Using a Stateless Session Bean with a Web Service"](#)
- [Chapter 31, "Administering an EJB Application"](#)

Accessing an EJB from a Client

This chapter explains how to access an EJB from a client, including:

- [What Type of Client Do You Have?](#)
- [Configuring the Client](#)
- [Accessing an EJB 3.0 EJB](#)
- [Accessing an EJB 3.0 EJB in Another Application](#)
- [Accessing an EJB 3.0 Entity Using an EntityManager](#)
- [Accessing an EJB 3.0 EJBContext](#)
- [Accessing an EJB 2.1 EJB](#)
- [Accessing an EJB 2.1 EJB in Another Application](#)
- [Accessing an EJB 2.1 MDB](#)
- [Accessing an EJB 2.1 EJBContext](#)
- [Handling Parameters](#)
- [Handling Exceptions](#)

For more information, see:

- ["How Do You Use an EJB in Your Application?"](#) on page 2-11
- ["Looking up an EJB 3.0 EJB"](#) on page 19-19
- ["Looking Up an EJB 3.0 Resource Manager Connection Factory"](#) on page 19-22
- ["Looking Up an EJB 3.0 Environment Variable"](#) on page 19-23
- ["Looking Up an EJB 2.1 EJB"](#) on page 19-24
- ["Looking Up an EJB 2.1 Resource Manager Connection Factory"](#) on page 19-26
- ["Looking Up an EJB 2.1 Environment Variable"](#) on page 19-27

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 EJB from a variety of clients, including:

- [EJB Client](#)

- [Stand-alone Java Client](#)
- [Servlet or JSP Client](#)

How you access an EJB, 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 EJB (call it the source EJB) accesses another EJB (call it the target EJB), the source EJB is the client of the target EJB.

If you are using EJB 3.0, using 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.

Stand-alone Java Client

A stand-alone Java client is a client that executes outside of OC4J but accesses EJB resources deployed to OC4J.

Typically, a stand-alone Java client accesses EJB resources by making use of Java RMI calls. You must code a stand-alone 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 EJB from a stand-alone 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 is not supported for a stand-alone 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 EJB Using RMI from a Stand-Alone Java Client"](#) on page 29-16).

Servlet or JSP Client

A servlet or JSP can access an EJB.

OC4J does not support dependency injection in the web container. Consequently, you can only 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 EJB 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 include on your client's classpath. The Source column indicates from where you get a copy of the required JAR.

Table 29–1 OC4J Client Classpath Requirements

OC4J JAR	Source	All Clients	OracleAS JMS Client	Oracle AQ JMS Client	J2CA JMS Client
oc4jclient.jar	<OC4J_HOME>/j2ee/<instance>	✓	✓	✓	✓
ejb.jar	<OC4J_HOME>/j2ee/<instance>/lib	✓	✓	✓	✓
jndi.jar	<OC4J_HOME>/j2ee/<instance>/lib	✓	✓	✓	✓
optic.jar ¹	<OC4J_HOME>/opmn/lib	✓	✓	✓	✓
jta.jar	<OC4J_HOME>/j2ee/<instance>/lib		✓	✓	✓
jms.jar	<OC4J_HOME>/j2ee/<instance>/lib		✓	✓	✓
javax77.jar	<OC4J_HOME>/j2ee/<instance>/lib		✓	✓	✓
adminclient.jar	<OC4J_HOME>/j2ee/<instance>/lib			✓	
jdbc.jar	<OC4J_HOME>/j2ee/<instance>/../lib			✓	
dms.jar	<OC4J_HOME>/j2ee/<instance>/lib			✓	
J2CA connector JAR	Connector provider				✓

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 19-16).

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 EJB, 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: stand-alone or as part of Oracle Application Server (see ["Configuring the Initial Context Factory"](#) on page 19-15).

Specifying Security Credentials

If the client and target EJB 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 EJB (see ["Specifying Credentials in EJB Clients"](#) on page 22-10).

Selecting an EJB Reference

In EJB 3.0, to access an EJB 3.0 EJB 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 look up with a predefined environment references.

In EJB 2.1 or in EJB 3.0 (for stand-alone Java clients or servlet or JSP clients), to access an EJB or resource, you must do a JNDI look up on a predefined environment references (see ["Configuring Environment References"](#) on page 19-1). To access an EJB 2.1 EJB 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 EJB 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 EJB, see ["Configuring an Environment Reference to an EJB"](#) on page 19-3.

[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	Optional
	<code>RMIInitialContext</code>	Not Used
Stand-alone Java Client	Default	Optional
	<code>ApplicationClientInitialContext</code>	Mandatory
Servlet or JSP Client	Default	Optional
	<code>RMIInitialContext</code>	Not Used

[Example 29-1](#) shows how to look up an EJB with logical name `ejb/HelloWorld` using the `java:comp/env/ejb/` prefix and [Example 29-2](#) shows how to look up this EJB without the prefix.

Example 29-1 Looking Up an EJB with the Prefix

```
InitialContext ic = new InitialContext();
HelloHome hh = (HelloHome)ic.lookup("java:comp/env/ejb/HelloWorld");
```

Example 29-2 Looking Up an EJB without the Prefix

```
InitialContext ic = new InitialContext();
HelloHome hh = (HelloHome)ic.lookup("ejb/HelloWorld");
```

Accessing an EJB 3.0 EJB

You can directly lookup 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 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.

For more information, see ["Looking up an EJB 3.0 EJB"](#) on page 19-19.

Accessing an EJB 3.0 EJB in Another Application

Normally, you cannot have EJBs communicating across EAR files, that is, across applications that are deployed in separate EAR files. The only way for an EJB to access an EJB 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 EJBs, each deployed within their EAR file, called `sales` and `inventory`, where the `sales` EJB needs to invoke the `inventory` EJB to check to see if enough widgets are available. Unless the `sales` EJB defines the `inventory` EJB to be its parent, the `sales` EJB cannot invoke any methods in the `inventory` EJB, because they are both deployed in separate EAR files. So, define the `inventory` EJB to be the parent of the `sales` EJB and the `sales` EJB 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 an EJB 3.0 Entity Using an EntityManager

In an EJB 3.0 application, the `javax.persistence.EntityManager` is the runtime access point for persisting entities to and loading entities from the database.

This section describes:

- [Acquiring an EntityManager](#)
- [Creating a New CMP Entity Instance](#)
- [Querying for an EJB 3.0 Entity Using the EntityManager](#)
- [Modifying a CMP Entity Instance](#)
- [Detaching and Merging a CMP Entity Bean Instance](#)

For more information, see ["How Do You Query for an EJB 3.0 Entity?"](#) on page 1-16.

Note: You can download an EJB 3.0 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 `EntityManager` depends on your client type (["What Type of Client Do You Have?"](#) on page 29-1).

You can acquire an `EntityManager`:

- [Using Annotations in an EJB 3.0 EJB Client](#)
- [Using the InitialContext in an EJB Client](#)
- [Using the InitialContext in a Web Client](#)

Using Annotations in an EJB 3.0 EJB Client

In this release, you can use the `@Resource` annotation to inject an `EntityManager` in an EJB 3.0 EJB clientonly, as [Example 29–3](#) shows.

Example 29–3 Using @Resource to Inject an EntityManager in a Stateless Session Bean

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession
{
    @Resource protected EntityManager entityManager;

    public void createEmployee(String fName, String lName)
    {
        Employee employee = new Employee();
        employee.setFirstName(fName);
        employee.setLastName(lName);
        entityManager.persist(employee);
    }
    ...
}
```

Using the InitialContext in an EJB Client

Alternatively, you can acquire an `EntityManager` using the initial context to perform a JNDI lookup as [Example 29–4](#) shows. In an EJB client, you use the binding:

```
java:comp/ejb/EntityManager
```

For more information, see ["Configuring the Initial Context Factory"](#) on page 19-15.

Example 29–4 Using InitialContext to Lookup an EntityManager in a Stateless Session Bean

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession
{
    protected EntityManager entityManager;
    ...
    public EntityManager getEntityManager()
    {
        if (entityManager == null)
        {
            try
            {
                entityManager = (EntityManager)(new InitialContext()).lookup(
                    "java:comp/ejb/EntityManager"
                );
            }
            catch (Exception e)
            {
            }
        }
        return entityManager;
    }
    ...
}
```

Using the InitialContext in a Web Client

Using OC4J, you can acquire an `EntityManager` in a Web client (servlet or JSP) using the initial context to perform a JNDI lookup as [Example 29–5](#) shows. In an EJB client, you use the binding:

```
java:comp/ejb/<EJB Module Name>/EntityManager
```

Note: As [Example 29-5](#) shows, in your Web client, 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 19-15.

Example 29-5 Using InitialContext to Lookup an EntityManager in a Servlet

```
public class InsertServlet extends HttpServlet
{
    private static final String CONTENT_TYPE = "text/html; charset=windows-1252";
    ...
    public void doPost(HttpServletRequest request, HttpServletResponse response) throws
        ServletException, IOException
    {
        ...
        UserTransaction ut = null;
        ...
        try
        {
            Context initCtx = new InitialContext();

            ut = (UserTransaction)initCtx.lookup("java:comp/UserTransaction");
            ut.begin();

            Employee employee = new Employee();
            employee.setEmpNo(empId);
            employee.setEname(name);
            employee.setSal(sal);

            Context context = new InitialContext();
            EntityManager entityManager = (EntityManager)context.lookup(
                "java:comp/ejb/ejb30EMfromweb-ejb/EntityManager"
            );
            entityManager.persist(employee);

            ut.commit();

            this.getServletContext().getRequestDispatcher("/jsp/success.jsp").forward(
                request, response
            );
        }
        catch(Exception e)
        {
            ...
        }
    }
    ...
}
```

Creating a New CMP Entity Instance

To create a new CMP entity instance, use `EntityManager` method `persist` passing in the entity Object as [Example 29-6](#) 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 29-13).

Example 29–6 Creating a CMP Entity with the EntityManager

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession
{
    @Resource 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 an EJB 3.0 Entity Using the EntityManager

Using the `EntityManager`, you can:

- [Finding an Entity by Primary Key with the Entity Manager](#)
- [Creating a Named Query with the EntityManager](#)
- [Creating a Dynamic EJB QL Query with the Entity Manager](#)
- [Creating a Dynamic TopLink Expression Query with the EntityManager](#)
- [Creating a Dynamic Native SQL Query with the EntityManager](#)
- [Configuring Query Hints](#)
- [Executing a Query](#)

For more information, see:

- ["How Do You Query for an EJB 2.1 Entity Bean?"](#) on page 1-27
- ["Using EJB 3.0 Query API"](#) on page 8-1

Finding an Entity by Primary Key with the Entity Manager

As [Example 29–7](#) shows, if 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 29–7 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 an EJB 3.0 Named Query"](#) on page 8-1), you can acquire it at runtime using EntityManager method `createNamedQuery` as [Example 29-8 Creating a Named Query with the EntityManager](#) shows. If the named query takes parameters, you set them using Query method `setParameter`.

Example 29-8 Creating a Named Query with the EntityManager

```
Query queryEmployeesByFirstName = entityManager.createNamedQuery(  
    "findAllEmployeesByFirstName"  
);  
queryEmployeeByFirstName.setParameter("firstName", "John");  
Collection employees = queryEmployeesByFirstName.getResultList();
```

Creating a Dynamic EJB QL Query with the Entity Manager

[Example 29-9](#) shows how to create an ad hoc EJB QL query at runtime using EntityManager method `createQuery`.

Example 29-9 Creating a Dynamic Query Using the EntityManager

```
Query queryEmployees = entityManager.createQuery(  
    "SELECT OBJECT(employee) FROM Employee employee"  
);
```

[Example 29-10](#) 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 29-10 Creating a Dynamic EJB QL Query with Parameters Using the EntityManager

```
Query queryEmployees = entityManager.createQuery(  
    "SELECT OBJECT(emp) FROM Employee emp WHERE emp.firstName = :firstname"  
);  
queryEmployeeByFirstName.setParameter("firstName", "John");
```

Creating a Dynamic TopLink Expression Query with the EntityManager

As [Example 29-11](#) shows, using the `oracle.toplink.ejb.cmp3.EntityManager` method `createQuery(Expression expression, Class resultType)`, you can create a query based on a TopLink Expression.

For more information, see "Understanding TopLink Expressions" in the *Oracle TopLink Developer's Guide*.

Alternatively, you can use `javax.persistence.EntityManager` and specify the TopLink Expression as a query hint (see ["Configuring Query Hints"](#) on page 29-10).

Example 29-11 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 the EntityManager method `createNativeQuery(String sqlString, Class resultType)`, you can create a query based on a native SQL String that you supply as [Example 29–12](#) shows.

Example 29–12 *Creating a Dynamic Native SQL Query with the EntityManager*

```
Query queryEmployees = entityManager.createNativeQuery(
    "Select * from EMP_TABLE where Salary < 50000", Employee.class
);
```

Note: OC4J does not support EntityManager method `createNativeQuery(String sqlString)`.

[Example 29–13](#) shows how to create an ad hoc native SQL query that takes a parameter named `salary` using EntityManager method `createNativeQuery(String sqlString, Class resultClass)`. You set the parameter using Query method `setParameter`.

Example 29–13 *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);
```

Configuring Query Hints

A query hint is name-value pair that you can use to configure a query with a vendor-specific option that is not available in the EJB 3.0 specification.

OC4J, using the TopLink persistence manager, provides the hints shown in [Table 29–3](#).

[Example 29–14](#) shows how to use Query method `setHint` to configure a named query to always refresh the TopLink cache from the database.

Example 29–14 *Configuring a Query with a Hint*

```
Query queryEmployeesByFirstName = entityManager.createNamedQuery(
    "findAllEmployeesByFirstName"
);
queryEmployeesByFirstName.setHint("refresh", new Boolean(true));
```

Table 29–3 Query Hints that OC4J Supports

Hint Name	Hint Value	Description
fetchSize	java.lang.Integer	Allows the user to set the fetch size of a TopLink query in bytes. For more information, see "JDBC Fetch Size" in the <i>Oracle TopLink Developer's Guide</i> .
referenceClass	java.lang.Class	Overrides the target class of the query.
cacheUsage	java.lang.Integer	Specifies how the query uses the TopLink cache. Values are as defined for the following fields of <code>oracle.toplink.queryframework.ObjectLevelReadQuery</code> : <ul style="list-style-type: none"> CheckCacheByExactPrimaryKey CheckCacheByPrimaryKey CheckCacheOnly CheckCacheThenDatabase ConformResultsInUnitOfWork For more information, see: <ul style="list-style-type: none"> "Configuring Cache Usage for In-Memory Queries" in the <i>Oracle TopLink Developer's Guide</i> "Understanding the Cache" in the <i>Oracle TopLink Developer's Guide</i>
refresh	java.lang.Boolean	Set to true if the TopLink cache should be refreshed from the database when this query executes. For more information, see "Understanding the Cache" in the <i>Oracle TopLink Developer's Guide</i> .
lockMode	java.lang.Integer	Specifies whether or not the query uses pessimistic locking. Values are as defined for the following fields of <code>oracle.toplink.queryframework.ObjectLevelReadQuery</code> : <ul style="list-style-type: none"> LOCK: issues SELECT FOR UPDATE. LOCK_NOWAIT: issues SELECT FOR UPDATE NO WAIT. NO_LOCK: pessimistic locking is not used. DEFAULT_LOCK_MODE: fine grained locking will occur. For more information, see the <i>Oracle TopLink API Reference</i> .
expression	oracle.toplink.expressions.Expression	Allows querying using the TopLink Expression API For more information, see: <ul style="list-style-type: none"> "Understanding TopLink Expressions" in the <i>Oracle TopLink Developer's Guide</i> "Creating a Dynamic TopLink Expression Query with the EntityManager" on page 29-9
timeout	java.lang.Integer	Sets the query timeout in milliseconds.

Executing a Query

As [Example 29–15](#) shows, to execute a query that returns multiple results, use Query method `getResultList`. This method returns a `java.util.List`.

Example 29–15 Executing a Query that Returns Multiple Results

```
Collection employees = queryEmployees.getResultList();
```

As [Example 29–16](#) shows, to execute a query that returns a single result, use Query method `getSingleResult`. This method returns a `java.lang.Object`.

Example 29–16 Executing a Query that Returns a Single Result

```
Object obj = query.getSingleResult();
```

As [Example 29-17](#) 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 29-17 Executing an Updating Query

```
Query queryRenameCity = entityManager.createQuery(
    "UPDATE Address add SET add.city = 'Ottawa' WHERE add.city = 'Nepean'"
);
int rowCount = queryRenameCity.executeUpdate();
```

Modifying a CMP Entity Instance

You can modify a CMP 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 29-13).

Using an Updating Query

Create an updating query (see ["Creating a Named Query with the EntityManager"](#) on page 29-9 or ["Creating a Dynamic EJB QL Query with the Entity Manager"](#) on page 29-9) and execute the query using the `EntityManager` (see ["Executing a Query"](#) on page 29-11).

Using the Entity's Public API

Use the `EntityManager` to find or otherwise query for the entity (see ["Querying for an EJB 3.0 Entity Using the EntityManager"](#) on page 29-8).

Use the entity's public API to change its persistent state.

Refreshing from the Database

As [Example 29-18](#) 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 29-18 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 29-19](#) shows, you can use `EntityManager` method `remove` to delete an entity from the database.

Example 29-19 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 29-20](#) 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 29-20 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 a CMP Entity Bean Instance

An `EntityManager` is said to have a persistence context. When you create (see ["Creating a New CMP Entity Instance"](#) on page 29-7) or find (see ["Querying for an EJB 3.0 Entity Using the EntityManager"](#) on page 29-8) 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 29-21](#) shows, you can use `EntityManager` method `merge` to merge the state of detached entity into the current persistence context of the `EntityManager`.

Example 29–21 Merging an Entity into the Persistence Context of an EntityManager

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession
{
    ...
    public void updateAddress(Address addressExample)
    {
        entityManager.merge(addressExample);
    }
    ...
}
```

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 29-14).

For more information, see:

- ["What is EJB Context?"](#) on page 1-6
- ["What is Session Context?"](#) on page 1-13
- ["What is Message Driven Context?"](#) on page 1-35

Using Resource Injection

In an EJB 3.0 EJB client, you can use `@Resource` injection to access the `EJBContext` as [Example 29–22](#) shows.

Example 29–22 Accessing EJBContext Using @Resource

```
@Resource SessionContext ctx;
```

Accessing an EJB 2.1 EJB

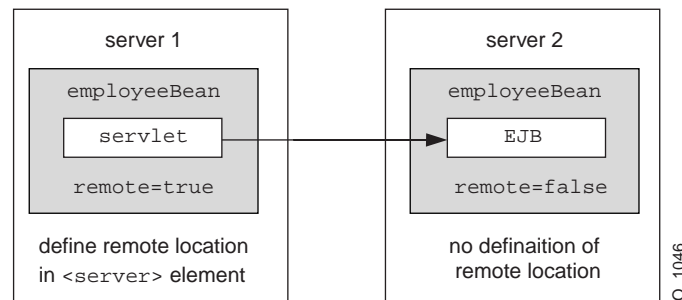
This section describes:

- [Accessing an EJB 2.1 EJB Remotely](#)
- [Accessing an EJB 2.1 EJB Locally](#)
- [Accessing an EJB 2.1 EJB Using RMI from a Stand-Alone Java Client](#)

Accessing an EJB 2.1 EJB Remotely

A remote multi-tier situation exists when you have the servlets executing in one server which are to connect and communicate with EJBs in another server. Both the servlets and EJBs are contained in the same application. When you deploy the application to two different servers, the servlets normally look for the local EJB 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 EJBs in server 2, you must set the `remote` attribute appropriately in the application before deploying on both servers.

Figure 29-1 Multi-Tier Example

The `remote` attribute in the `<ejb-module>` element in `orion-application.xml` for the EJB module denotes whether the EJBs 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 EJBs 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 stand-alone 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:

- ["Configuring the Initial Context Factory"](#) on page 19-15
- ["Setting JNDI Properties for RMI"](#) in the *Oracle Containers for J2EE Services Guide*.

5. Look up the remote EJB.

For more information, see:

- ["Looking Up the Remote Interface of an EJB 2.1 EJB Using ejb-ref"](#) on page 19-24

- ["Looking Up the Remote Interface of an EJB 2.1 EJB Using location"](#) on page 19-25

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 EJB Locally

A local multi-tier situation exists when both the servlets and EJBs 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 EJBs 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:

- ["Configuring the Initial Context Factory"](#) on page 19-15
- "Setting JNDI Properties for RMI" in the *Oracle Containers for J2EE Services Guide*.

3. Look up the local EJB.

For more information, see:

- ["Looking up the Local Interface of an EJB 2.1 EJB Using local-ref"](#) on page 19-25
- ["Looking up the Local Interface of an EJB 2.1 EJB Using local-location"](#) on page 19-26

Accessing an EJB 2.1 EJB Using RMI from a Stand-Alone Java Client

[Example 29-23](#) shows the type of look up that you can use from a stand-alone Java client (see ["Stand-alone Java Client"](#) on page 29-2) in this release to look up an OC4J-deployed EJB without having to specify an RMI port. [Example 29-23](#) shows how to look up the EJB named `MyCart` in the J2EE application `ejbsamples` deployed to the OC4J instance named `oc4j_inst1` running on host `myServer`.

Example 29-23 Accessing an EJB 2.1 EJB Using RMI from a Stand-Alone 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:

- ["Configuring an Oracle Initial Context Factory"](#) on page 19-16
- ["Configuring the Naming Provider URL for OC4J and Oracle Application Server"](#) on page 19-17
- ["Configuring the Naming Provider URL for OC4J Standalone"](#) on page 19-18

Accessing an EJB 2.1 EJB in Another Application

Normally, you cannot have EJBs communicating across EAR files, that is, across applications that are deployed in separate EAR files. The only way for an EJB to access an EJB 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 EJBs, each deployed within their EAR file, called `sales` and `inventory`, where the `sales` EJB needs to invoke the `inventory` EJB to check to see if enough widgets are available. Unless the `sales` EJB defines the `inventory` EJB to be its parent, the `sales` EJB cannot invoke any methods in the `inventory` EJB, because they are both deployed in separate EAR files. So, define the `inventory` EJB to be the parent of the `sales` EJB and the `sales` EJB 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 an EJB 2.1 MDB

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.

This section describes:

- [Sending a Message to a JMS Destination Using EJB 2.1](#)
- [Sending a Message to a J2CA Destination Using EJB 2.1](#)

Sending a Message to a JMS Destination Using EJB 2.1

To send a message to a JMS destination using EJB 2.1:

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 (see ["Configuring an Environment Reference to a JMS Destination or Connection Resource Manager Connection Factory"](#) on page 19-7) or the explicit JNDI name you defined when you configured your JMS provider (see ["Configuring an EJB 2.1 MDB to Use a Non-J2CA Message Service Provider"](#) on page 18-1).

Oracle recommends that you use logical names as shown in this procedure and its examples.

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 29–24](#) shows how a servlet client sends a message to a queue.

[Example 29–25](#) shows how a JSP client sends a message over a topic.

Example 29–24 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. We're receiving msgs, 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 msgs sent to us via 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 recipient as CLIENT. Thus, we will only receive msgs that have
    // RECIPIENT set to 'CLIENT'
    QueueReceiver rcv = sess.createReceiver (q, "RECIPIENT = 'CLIENT'");

    int nrcvd = 0;
    long trtimes = 0L;
    long tctimes = 0L;
    // First msg 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 msg in message Map
        // We check the msgMap to see if this is the message that we 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 msgs from queue
    for (Message msg = rcv.receive(1000); msg != null; msg = rcv.receive(1000))
        nrcvd++;

    rcv.close();

    return nrcvd;
}
}
```

Example 29–25 JSP Client Sends Message to a Topic

```
<%@ page import="javax.jms.*, javax.naming.*, java.util.*" %>
<%
//1a. Lookup the MessageBean topic
jndiContext = new InitialContext();
topic = (Topic)jndiContext.lookup("rpTestTopic");

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

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

//5. Create the publisher for any messages destined for the topic
topicPublisher = topicSession.createPublisher(topic);

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

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

//7. Close publisher, session, and connection for topic
topicPublisher.close();
topicSession.close();
topicConnection.close();
%>
Message sent!
```


Sending a Message to a J2CA Destination Using EJB 2.1

To send a message to a J2CA destination using EJB 2.1:

1. Look up both the J2CA connection factory.

You can look up this resource using a predefined logical name (see ["Configuring an Environment Reference to a JMS Destination or Connection Resource Manager Connection Factory"](#) on page 19-7) or the explicit JNDI name you defined when you configured your JMS provider (see ["Configuring an EJB 2.1 MDB to Use a Non-J2CA Message Service Provider"](#) on page 18-1).

Oracle recommends that you use logical names as shown in this procedure and its examples.

2. Obtain a `javax.resource.cci.ConnectionFactory`.

If the EIS is a JMS message service provider, there will likely be connection factory choices for queue or topic. For example, the Oracle JMS Connector offers a `QueueConnectionFactory` and a `TopicConnectionFactory`.

3. Use the factory to obtain a `javax.resource.cci.Connection`.
4. Use the connection to obtain a `javax.resource.cci.Interaction`.
5. Configure the interaction and use `Interaction` method `execute` to send the message.

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 `get` and `set` method when you implement your bean.

For more information, see:

- ["What is EJB Context?"](#) on page 1-6
- ["Implementing the `setSessionContext` Method"](#) on page 11-10
- ["Implementing the `setEntityContext` and `unsetEntityContext` Methods"](#) on page 13-20
- ["Implementing the `setMessageDrivenContext` Method"](#) on page 17-6

Handling Parameters

This section describes:

- [Passing Parameters Into an EJB](#)
- [Handling Parameters Returned by an EJB](#)

Passing Parameters Into an EJB

When you implement an EJB or write the client code that calls EJB methods, you must be aware of the parameter-passing conventions used with EJBs.

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 non-remote object that implements the `java.io.Serializable` interface can be passed. A non-remote 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 non-remote object is complex—such as a class containing several fields—only the non-static and non-transient 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 EJB

The `EmployeeBean` `getEmployee` method 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:

- [Recovering From a NamingException While Accessing a Remote EJB](#)
- [Recovering From a NullPointerException While Accessing a Remote EJB](#)
- [Recovering From Deadlock Conditions](#)

Recovering From a NamingException While Accessing a Remote EJB

If you are trying to remotely access an EJB and you receive an `javax.naming.NamingException` error, your JNDI properties are probably not

initialized properly. See ["Load Balancing"](#) on page 2-22 for a discussion on setting up JNDI properties when accessing an EJB from a remote object or remote servlet.

Recovering From a NullPointerException While Accessing a Remote EJB

When accessing a remote EJB 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 EJB 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 above
Context context = new InitialContext (env);
```

See ["Load Balancing"](#) on page 2-22 for more information on `dedicated.rmicontext`.

Recovering From Deadlock Conditions

If the call sequence of several beans cause a deadlock scenario, OC4J notices the deadlock condition and throws a `Remote` exception that details the deadlock condition in one of the offending beans.

Using a Stateless Session Bean with 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.

This section describes:

- [Calling Out to a Web Service](#)
- [Exposing a Stateless Session Bean to a Web Service](#)

For more information, see the *Oracle Application Server Web Services Developer's Guide*.

Calling Out to a Web Service

From within a stateless session bean, you can obtain a web service and invoke its methods.

You must use the initial context (see ["Using Initial Context"](#) on page 30-1) and you must create an environment reference for the web service (see ["Configuring an Environment Reference to a Web Service"](#) on page 19-14) before you can look it up.

Using Initial Context

After you define an environment reference to a web service (see ["Configuring an Environment Reference to a Web Service"](#) on page 19-14), you can use the initial context to look up the web service and invoke its methods from within your stateless session bean as [Example 30-1](#) shows.

Example 30-1 *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(...);  
        ...  
    }  
}
```

Exposing a Stateless Session Bean to a Web Service

Oracle Application Server Web Services run EJBs that are deployed as Oracle Application Server Web Services in response to a request issued by a Web Service client.

Refer to *Oracle Application Server Web Services Developer's Guide* for complete information.

Administering an EJB Application

This chapter describes:

- [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.

Using Oracle Enterprise Manager 10g 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. You can use Application Server Control for most administration tasks.

For more information, see:

- "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:

- [Logging Namespaces](#)
- [Logging Levels](#)
- [Configuring Logging with Application Server Control Logging MBean](#)
- [Configuring Logging Using the j2ee-logging.xml File](#)
- [Configuring Logging Using System Properties](#)

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 log levels of: `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 like 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:

- ["Logging Namespaces"](#) on page 31-2
- ["Logging Levels"](#) on page 31-2

Configuring Logging Using System Properties

You can configure OC4J logging using the `oracle.j2ee.logging` system property. This system property has the form:

```
oracle.j2ee.logging.<log-level>=<log-namespace>
```

Where:

- `<log-level>` is one of `fine`, `finer`, or `finest`.

- `<log-namespace>` 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
```

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:

- [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:

- [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 size with the following attributes of the `<entity-deployment>` element, for entities, and the `<session-deployment>` element, for session beans:

- The `max-instances` attribute sets the maximum number of bean instances allowed in the pool. The default is 0, which means infinite.

For example, if you wanted to set the maximum entity implementation instances to 20, you would do as follows:

```
<entity-deployment ... max-instances="20"
...
</entity-deployment>
```

- The `min-instances` attribute sets the minimum number of bean instances allowed in the pool.

For example, if you wanted to set the minimum entity implementation instances to 2, you would do as follows:

```
<entity-deployment ... min-instances="2"
...
</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).

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:

- [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 `<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:

- [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:

- [Validating XML Files](#)
- [Debugging the ejb-jar.xml File](#)
- [Debugging Generated 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.j`).

[Example 31-3](#) shows how to set this option in the `oc4j.j` file.

Example 31-3 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 31-1).

Debugging the ejb-jar.xml File

You can configure OC4J to write out the `ejb-jar.xml` file it creates based on your EJB 3.0 annotations (see ["Troubleshooting Application Deployment"](#) on page 28-3).

See also ["Validating XML Files"](#) on page 31-5.

Debugging Generated Code

By default, when OC4J deploys an 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:

- [Preserving Generated Code in the Default Directory](#)
- [Preserving Generated Code in a Directory You Specify](#)
- [Disabling Generated Code Preservation](#)

Preserving Generated Code in the Default Directory

If you set system property `KeepWrapperCode` to `true`, OC4J preserves the wrapper code it generates in the default directory `<OC4J_HOME>/j2ee/home/application-deployments/<ear-name>/<ejb-name>/generated`.

If you undeploy your application, OC4J deletes the wrapper code in this directory.

Preserving Generated 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.

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.

Disabling Generated Code Preservation

If you set system property `KeepWrapperCode` to `false` (or leave this system property unset), OC4J will not preserve the wrapper code it generates.

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`. This deployment descriptor file conforms to the XML schema document (XSD) located at http://www.oracle.com/technology/oracleas/schema/orion-ejb-jar-10_0.xsd.

This appendix describes:

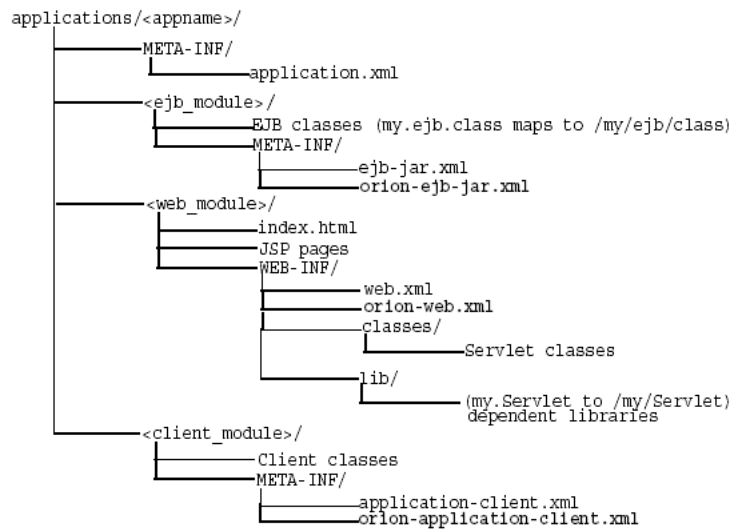
- [OC4J and the orion-ejb-jar.xml File](#)
- [TopLink Persistence Manager Support](#)
- [OC4J-Specific Deployment Descriptor for EJBs](#): Overall description of each element section.
 - [Enterprise Beans Section](#)
 - * [Persistence Manager Section \(persistence-manager\)](#)
 - * [Session Bean Section \(session-deployment\)](#)
 - * [Entity Bean Section \(entity-deployment\)](#)
 - * [Message Driven Bean Section \(message-driven-deployment\)](#)
 - * [EJB 1.1 CMP Field Mapping Section \(cmp-field-mapping\)](#)
 - * [Method Definition](#)
 - [Assembly Descriptor Section](#)
- [Element Description](#): An alphabetical listing and description for each element.

For more information, see "[Understanding EJB Deployment Descriptor Files](#)" on page 2-7

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 shown in [Figure A-1](#).

Figure A–1 Development Application Directory Structure

TopLink Persistence Manager Support

Table A–1 summarizes which `orion-ejb-jar.xml` options are applicable to TopLink CMP. For example:

- To configure `<entity-deployment>` attribute `call-timeout`, you must use the corresponding TopLink persistence manager 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 manager API.

Table A–1 OC4J `orion-ejb-jar.xml` Features and TopLink

orion-ejb-jar.xml Feature	Configurable in orion-ejb-jar.xml	Configurable in toplink-ejb-jar.xml
<code><entity-deployment></code>		
<code>clustering-schema</code>	✓	
<code>copy-by-value</code>	✓	
<code>data-source</code>	✓	
<code>location</code>	✓	
<code>max-instances</code>	✓	
<code>min-instances</code>	✓	
<code>max-tx-retries</code>	✓	
<code>disable-wrapper-cache</code>	✓	
<code>name</code>	✓	
<code>pool-cache-timeout</code>	✓	
<code>wrapper</code>	✓	
<code>local-wrapper</code>	✓	
<code>call-timeout</code>		✓
<code>exclusive-write-access</code>		
<code>true</code>		✓

Table A-1 (Cont.) OC4J orion-ejb-jar.xml Features and TopLink

orion-ejb-jar.xml Feature	Configurable in orion-ejb-jar.xml	Configurable in toplink-ejb-jar.xml
false		
do-select-before-insert		
true		
false		
isolation		✓
locking-mode		
pessimistic		✓
optimistic		
read-only		✓
old_pessimistic		
update-changed-fields-only		
true		✓
false		
table		✓
force-update		
true		
false		✓
data-synchronization-option		
ejbCreate		
ejbPostCreate		
batch-size		
Any value greater than 1		
<ior-security-config>	✓	
<env-entry-mapping>	✓	
<resource-ref-mapping>	✓	
<resource-env-ref-mapping>	✓	
<primkey-mapping>		✓
<cmp-field-mapping>		
one-to-one-join		
inner		✓
outer ¹		
shared		✓
<finder-method>		✓
<persistence-type> ²		✓

¹ TopLink supports both outer and inner joins at run time. You can manually configure EJB descriptors with these options. For more information, "Understanding TopLink Queries" in the *Oracle TopLink Developer's Guide*.

² The persistence-type attribute's table column size, if present, is discarded. For more information, "Migrating OC4J Orion Persistence to OC4J TopLink Persistence" in the *Oracle TopLink Developer's Guide*.

For EJB 3.0 applications, you can access TopLink persistence manager 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 TopLink Entity Manager"](#) on page 3-2.

For EJB 2.1 applications, you can access TopLink persistence manager API using `orion-ejb-jar.xml` element `pm-properties`. For more information, see ["Customizing the TopLink Persistence Manager"](#) on page 3-5.

Note: When using OC4J with TopLink, use TopLink Workbench to modify the `toplink-ejb-jar.xml` file. Refer to the *Oracle TopLink Developer's Guide* for complete information on migrating information from the `orion-ejb-jar.xml` file.

The `ejb3-toplink-sessions.xml` and `toplink-ejb-jar.xml` file conform to the XML schema documents located at `<OC4J_HOME>\toplink\config\xsds`.

OC4J-Specific Deployment Descriptor for EJBs

The OC4J-specific deployment descriptor contains extended deployment information for session beans, entity beans, message driven beans, and security for these EJBs. 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 sections below:

- [Enterprise Beans Section](#)
- [Assembly Descriptor Section](#)

Enterprise Beans Section

The `<enterprise-beans>` section defines additional deployment information for all EJBs: 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 Section \(persistence-manager\)](#)
- [Session Bean Section \(session-deployment\)](#)
- [Entity Bean Section \(entity-deployment\)](#)
- [Message Driven Bean Section \(message-driven-deployment\)](#)
- [EJB 1.1 CMP Field Mapping Section \(cmp-field-mapping\)](#)
- [Method Definition](#)

Persistence Manager Section (persistence-manager)

The <persistence-manager> section provides additional deployment information for the TopLink persistence manager for EJB 2.1 applications only. 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>
```

For more information on the attributes for the <persistence-manager> element, see:

- ["Customizing the TopLink Persistence Manager"](#) on page 3-5
- ["Configuring pm-properties"](#) in the *Oracle TopLink Developer's Guide*.

Session Bean Section (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=...
<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=... />
<resource-ref-mapping location=... name=... >
  <lookup-context location=...>
    <context-attribute name=... value=... />
  </lookup-context>
</resource-ref-mapping>
<resource-env-ref-mapping location=... name=... />
</session-deployment>
```

Table A-2 lists the attributes for the `<session-deployment>` element and indicates which are applicable to stateless session beans only, stateful session beans only, or both.

Each of the element groups are discussed in the following sections of the OC4J documentation set:

- A session bean example, which includes the `<session-deployment>` element (where relevant), is described in:
 - ["Implementing an EJB 3.0 Session Bean"](#) on page 4-1
 - ["Implementing an EJB 2.1 Session Bean"](#) on page 11-1
- 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*.
- 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 19-13.
- The `<ejb-ref-mapping>` element maps any EJB references to JNDI names and is discussed in ["Configuring an Environment Reference to an EJB"](#) on page 19-3.
- 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.
- 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.

Table A-2 Attributes for `<session-deployment>` Element

Attribute	Stateless	Stateful	Description
pool-cache-timeout	✓		<p>The pool-cache-timeout applies for stateless session EJBs. This parameter specifies how long to keep stateless sessions cached in the pool.</p> <p>For stateless session beans, if you specify a pool-cache-timeout, then at every pool-cache-timeout interval, all beans in the pool, of the corresponding bean type, 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.</p> <p>Default Value: 60 (seconds)</p>

Table A-2 (Cont.) Attributes for <session-deployment> Element

Attribute	Stateless	Stateful	Description
call-timeout	✓	✓	<p>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.</p> <p>If the timeout is reached, a <code>TimeoutException</code> is thrown. This excludes database connections.</p> <p>Default Values: 90000 milliseconds. Set to 0 if you want the timeout to be forever. See the EJB section in the <i>Oracle Application Server Performance Guide</i> for more information.</p>
copy-by-value	✓	✓	<p>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 is 'true'.</p>
location	✓	✓	The JNDI-name to which this bean will be bound.
local-location	✓	✓	The local JNDI name to which this EJB will be bound.
max-instances	✓	✓	<p>The number of bean instances allowed in memory—either instantiated or pooled. When this value is reached, the container attempts to passivate the oldest bean instance from memory. If unsuccessful, the container waits the number of milliseconds set in the <code>call-timeout</code> attribute to see if a bean instance is removed from memory, either through passivation, its <code>remove</code> method, or bean expiration, before a <code>TimeoutExpiredException</code> is thrown back to the client. To allow an infinite number of bean instances, the <code>max-instances</code> attribute can be set to zero. Default is 0, which means infinite. This applies to both stateless and stateful session beans.</p> <p>To disable instance pooling, set <code>max-instances</code> to any negative number in the <code>orion-ejb-jar.xml</code> file. This will create a new instance at the start of the EJB call and release it at the end of the call.</p>
min-instances	✓		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.
max-tx-retries	✓	✓	<p>This parameter specifies the number of times to retry a transaction that was rolled back due to system-level failures. The default is 0.</p> <p>For a stateful session bean, if a <code>RuntimeException</code>, <code>Error</code>, or <code>RemoteException</code> is thrown, the OC4J does not do a retry.</p> <p>Generally, we 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 <code>max-tx-retries=0</code>.</p> <p>See the EJB section in the <i>Oracle Application Server Performance Guide</i> for more information.</p>

Table A-2 (Cont.) Attributes for <session-deployment> Element

Attribute	Stateless	Stateful	Description
tx-retry-wait	✓	✓	This parameter specifies the time to wait in seconds between retrying the transaction. The default is 60 seconds.
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).
persistence-filename		✓	Path to the file where sessions are stored across restarts.
timeout		✓	<p>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.</p> <p>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.</p> <p>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.</p> <p>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.</p> <p>Default Value: 1800 seconds</p>
transaction-timeout	✓	✓	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.
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 min.). To disable, specify "never."
memory-threshold		✓	<p>This attribute defines a threshold for how much used JVM memory is allowed before passivation should occur.</p> <p>Specify an integer that is translated as a percentage.</p> <p>When reached, beans are passivated, even if their idle timeout has not expired. Default: 80%. To disable, specify "never."</p>
max-instances-threshold		✓	<p>Percentage of max-instances number of beans that can be in memory before passivation occurs.</p> <p>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%.</p> <p>To disable, specify "never."</p>
resource-check-interval		✓	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 "never."

Table A-2 (Cont.) Attributes for <session-deployment> Element

Attribute	Stateless	Stateful	Description
passivate-count		✓	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.
wrapper	✓	✓	Name of the OC4J wrapper class for this bean. This is an internal server value and should not be edited.
local-wrapper	✓	✓	Name of the OC4J local home wrapper class for this bean. This is an internal server value and should not be edited.
replication		✓	Configuration of the state replication for stateful session beans. Values can be inherited (default) onShutdown, onRequestEnd, or none. See "State Replication" on page 2-20 for more information.

Entity Bean Section (entity-deployment)

The <entity-deployment> section provides additional deployment information for an entity bean deployed within this JAR file. The <entity-deployment> section contains the following structure:

```

<entity-deployment call-timeout=... clustering-schema=...
    copy-by-value=... data-source=... exclusive-write-access=...
    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=... />
<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>
```

Table A-3 lists the attributes for the <entity-deployment> element.

Each of the element groups are discussed in the following sections of the OC4J documentation set:

- Entity bean examples, which include the <entity-deployment> element (where relevant), are described in:
 - ["Implementing an EJB 3.0 Entity"](#) on page 6-1
 - ["Implementing an EJB 2.1 Entity Bean"](#) on page 13-1
- 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*.
- The <primkey-mapping> element maps the primary key to the CMP field it represents. For more information, see:
 - ["What is a CMP Entity Bean Primary Key?"](#) on page 1-21
 - ["What is a BMP Entity Bean Primary Key?"](#) on page 1-23
- The <cmp-field-mapping> element maps each <cmp-field> element to its database row. For more information, see ["What are Container-Managed Persistence Fields?"](#) on page 1-20.
- The <finder-method> element is used to create finder methods for EJB 1.1 entity beans. For more information on EJB 3.0 and EJB 2.1, see ["How Do You Query for an EJB 2.1 Entity Bean?"](#) on page 1-27
- 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 19-13.
- The <ejb-ref-mapping> element maps any EJB references to JNDI names and is discussed in ["Configuring an Environment Reference to an EJB"](#) on page 19-3.
- 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 19-14
- 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.
- 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.

- 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 and is discussed in "What are Entity Bean Commit Options?" on page 1-26.
- The `<message-destination-ref-mapping>` element maps logical destinations to resource adapter destinations.

For example, in `ejb-jar.xml` a `<message-destination>` named `DealerToPlayerMessages` is declared. In file `application-client.xml` the `<message-destination-ref>` named `jms/PlayerResponseDestination` is linked to that `<message-destination>`, and in file `ejb-jar.xml` the `<message-destination-ref>` named `jms/ToPlayerDest` is also linked to that `<message-destination>`.

Note that the `<message-destination-type>` for both of these is `javax.jms.Topic`. The deployer maps both of these `<message-destination-ref>`s to a resource adapter topic at JNDI location `OracleASJMSRSubcontext/MyT` by adding this to the `orion-ejb-jar.xml` file:

```
<message-destination-ref-mapping
  location = "OracleASJMSRSubcontext/MyT"
  name = "jms/ToPlayerDest"/>
```

Table A-3 Attributes for `<entity-deployment>` Element

Attribute	Description
<code>call-timeout</code>	<p>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.</p> <p>If the timeout is reached, a <code>TimeoutException</code> is thrown. This excludes database connections.</p> <p>Default Values: 90000 milliseconds. Set to 0 if you want the timeout to be forever. See the EJB section in the <i>Oracle Application Server Performance Guide</i> for more information.</p>
<code>clustering-schema</code>	Do not use. Not needed in this release.
<code>copy-by-value</code>	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 is 'true'.
<code>data-source</code>	The name of the data source used if using container-managed persistence.

Table A-3 (Cont.) Attributes for <entity-deployment> Element

Attribute	Description
exclusive-write-access	<p>Whether or not the EJB-server has exclusive write (update) access to the database back-end. This can be used only for entity beans that use a "read_only" locking mode. In this case, it increases the performance for common bean operations and enables better caching.</p> <p>This parameter corresponds to which commit option is used (A, B or C, as defined in the EJB specification). When exclusive-write-access = true, this is commit option A.</p> <p>Default is false for beans with locking-mode=optimistic or pessimistic and true for locking-mode=read-only.</p> <p>The exclusive-write-access is forced to false if locking is pessimistic or optimistic, and is not used with EJB clustering. The exclusive-write-access can be false with read-only locking, but read-only won't have any performance impact if exclusive-write-access=false, since ejbStores are already skipped when no fields have been changed. To see a performance advantage and avoid doing ejbLoads for read-only beans, you must also set exclusive-write-access=true.</p> <p>See "Configuring Exclusive Write Access to the Database" on page 14-10 for more information.</p>
do-select-before-insert	<p>If false, you avoid executing a select before an insert. The extra select normally checks to see if the entity already exists before doing the insert to avoid duplicates.</p> <p>If a unique key constraint is defined for the entity, then we recommend setting this to false. If there is no unique key constraint, setting this to false leads to not detecting a duplicate insert. To prevent duplicate inserts in this case, leave it set to true.</p> <p>For performance, Oracle recommends setting this to false to avoid the extra select before insert. Default Value: true</p>
location	The JNDI-name to which this bean will be bound.
local-location	Defines the local JNDI name to which this EJB will be bound
isolation	<p>Specifies the isolation-level for database actions. The valid values for Oracle databases are 'serializable' and 'committed'. The default is 'committed'. Non-Oracle databases can be the following: 'none', 'committed', 'serializable', 'uncommitted', and 'repeatable_read'.</p> <p>For more information, see "How do You Avoid Database Resource Contention?" on page 1-24 and <i>Oracle Application Server Performance Guide</i>.</p>
locking-mode	<p>This attribute applies only to Orion CMP.</p> <p>For more information on TopLink CMP locking modes, see "Concurrency (Locking) Mode" on page 1-25.</p>
max-instances	The maximum number of bean implementation instances to be kept instantiated or pooled. The default is 0, which means infinite. See "Configuring Bean Instance Pool Size" on page 31-3 for more information.
min-instances	The minimum number of bean implementation instances to be kept instantiated or pooled. The default is 0. See "Configuring Bean Instance Pool Size" on page 31-3 for more information.

Table A-3 (Cont.) Attributes for <entity-deployment> Element

Attribute	Description
max-tx-retries	<p>This parameter specifies the number of times to retry a transaction that was rolled back due to system-level failures. The default is 0.</p> <p>Generally, we 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.</p> <p>Default Value: 0. See the EJB section in the <i>Oracle Application Server Performance Guide</i> for more information.</p>
update-changed-fields-only	Specifies whether the container updates only modified fields or all fields to persistence storage for CMP entity beans when <code>ejbStore</code> is invoked. The default is true, which specifies to only update modified fields.
name	The name of the bean, which matches the name of a bean in the assembly section of the EJB deployment descriptor (<code>ejb-jar.xml</code>).
pool-cache-timeout	The amount of time in seconds that the bean implementation instances are to be kept in the "pooled" (unassigned) state, specifying 'never' retains the instances until they are garbage collected. The default is 60.
table	The name of the table in the database if using container-managed persistence.
validity-timeout	<p>The maximum amount of time (in milliseconds) that an entity is valid in the cache (before being reloaded). Useful for loosely coupled environments where rare updates from legacy systems occur. This attribute is only valid for entity beans with locking mode of <code>read_only</code> and when <code>exclusive-write-access="true"</code> (the default).</p> <p>We recommend that if the data is never being modified externally (and therefore you've set <code>exclusive-write-access=true</code>), that you can set this to 0 or -1, to disable this option, since the data in the cache will always be valid for read-only EJBs that are never modified externally.</p> <p>If the EJB is generally not modified externally, so you're using <code>exclusive-write-access=true</code>, yet occasionally the table is updated so you need to update the cache occasionally, then set this to a value corresponding to the interval you think the data may be changing externally.</p>
force-update	If OC4J does not believe that any of the persistence data has changed, the <code>force-update</code> attribute set to true means that OC4J will still execute the EJB lifecycle by invoking the <code>ejbStore</code> method. This manages data in transient fields and sets appropriate persistent fields during the <code>ejbStore</code> method. For example, an image might be kept in one format in memory, but stored in a different format in the database. The default is false.
wrapper	Name of the OC4J remote home wrapper class for this bean. This is an internal server value and should not be edited.
local-wrapper	Name of the OC4J local home wrapper class for this bean. This is an internal server value and should not be edited.
delay-updates-until-commit	This attribute is valid only for CMP entity beans. Defers the flushing of transactional data until commit time or not. The default is true. Set this value to false to update persistence data after completion of every EJB method invocation - except <code>ejbRemove()</code> and the finder methods.
findByPrimaryKey-lazy-loading	<p>true or false</p> <p>Default: true</p>

Message Driven Bean Section (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=... >
<env-entry-mapping name=...></env-entry-mapping>
<ejb-ref-mapping location=... name=... />
<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>
```

[Table A-4](#) lists the attributes for the <message-driven-deployment> element and their J2CA message service provider resource adapter <config-property> equivalents (where appropriate).

Note: J2CA message service provider resource adapters read only certain <message-driven-deployment> attributes (see [Table A-4](#)) and ignore all other attributes. For these other attributes, use the resource adapter equivalent <config-property> given in [Table A-4](#).

Each of the element groups are discussed in the following sections of the OC4J documentation set:

- A message-driven bean example, which includes the <message-driven-deployment> element, is described in:
 - ["Implementing an EJB 3.0 MDB"](#) on page 9-1
 - ["Implementing an EJB 2.1 MDB"](#) on page 17-1
- For information on message service providers that OC4J supports, see ["What Message Providers Can I use with My MDB?"](#) on page 2-16.
- 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 19-13.
- The <ejb-ref-mapping> element maps any EJB references to JNDI names and is discussed in ["Configuring an Environment Reference to an EJB"](#) on page 19-3.
- 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.
- 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"](#) on page 19-7 for more information.

- The `<message-destination-ref-mapping>` element is only used if you are using a J2CA message service provider. Use this element to map logical destinations to J2CA resource adapter destinations. For more information, see ["Configuring an Environment Reference to a JMS Destination or Connection Resource Manager Connection Factory"](#) on page 19-7.
- 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.

For more information, see:

- ["Configuring an EJB 3.0 MDB to Use a J2CA Message Service Provider"](#) on page 10-3
- ["Configuring an EJB 2.1 MDB to Use a J2CA Message Service Provider"](#) on page 18-2

Table A-4 Attributes for `<message-driven-deployment>` Element

Attribute	<code><config-property></code> Equivalent	Description
connection-factory-location	ConnectionFactoryTimeout	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 xxxConnectionFactories details what type of factory is being defined.
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.
resource-adapter ¹	Use attribute.	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 Message Service Provider Using J2CA" on page 23-7.
name ¹	Use attribute.	The name of the bean, which matches the name of a bean in the assembly section of the EJB deployment descriptor (ejb-jar.xml).
subscription-name	SubscriptionName	If this is a topic, the subscription name is defined in the subscription-name attribute.
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 Listener Threads" on page 18-6.

Table A-4 (Cont.) Attributes for <message-driven-deployment> Element

Attribute	<config-property> Equivalent	Description
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 21-2
dequeue-retry-count	DequeueRetryCount	Specifies how often the listener thread tries to re-acquire the JMS session once database failover has occurred. The default is "0." This value is only for CMT transactions in an MDB. For more information, see: <ul style="list-style-type: none"> ▪ "Configuring Dequeue Retry Count and Interval" on page 18-8 ▪ "Understanding OC4J EJB Application Clustering Services" on page 2-20
dequeue-retry-interval	DequeueRetryInterval	Specifies the interval between retries. The default is 60 seconds. For more information, see: <ul style="list-style-type: none"> ▪ "Configuring Dequeue Retry Count and Interval" on page 18-8 ▪ "Understanding OC4J EJB Application Clustering Services" on page 2-20
max-instances ¹	Use attribute.	The maximum number of bean implementation instances to be kept instantiated or pooled. The default is 0, which means infinite. See "Configuring Bean Instance Pool Size" on page 31-3 for more information. For message-driven beans, the default pooling setting is typically appropriate. Change this value only if MDB lifecycle methods are very expensive and you need fine-grained control over how often instances are created and managed in the pool.
min-instances ¹	Use attribute.	The minimum number of bean implementation instances to be kept instantiated or pooled. The default is 0. See "Configuring Bean Instance Pool Size" on page 31-3 for more information.
cache-timeout ¹	Use attribute.	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 in the pool, of the corresponding bean type, are removed. If the value specified is zero or negative, then the cache-timeout is disabled and beans are not removed from the pool. Default Value: 60 (seconds)
max-delivery-count	MaxDeliveryCnt	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). After this number of re deliveries, the message is deemed undeliverable and is handled according to the policies of your message service provider. For example, OracleAS JMS will put the message on its exception queue (jms/Oc4jJmsExceptionQueue). For more information, see "Configuring Maximum Delivery Count" on page 18-7.

¹ J2CA message service provider resource adapters read this attribute but ignore all other <message-driven-deployment> attributes. For other <message-driven-deployment> attributes, use the resource adapter equivalent <config-property>.

EJB 1.1 CMP Field Mapping Section (cmp-field-mapping)

If you still use EJB 1.1 CMP entity beans, use the following elements to map the CMP fields to the database.

The following are the XML elements used for CMP persistent data field mapping within the `orion-ejb-jar.xml` file:

```
<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>
```

Method Definition

The following structure is used to specify the methods (and possibly parameters of that method) of the 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>
```

The style used can be one of the following:

1. 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>
```

2. 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>>
```

3. 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>
```

The `<method>` element is used within the security and MDB sections. See ["Specifying Logical Roles in the EJB Deployment Descriptor"](#) on page 22-3 for more information.

Assembly Descriptor Section

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>
  <default-method-access>
    <security-role-mapping impliesAll=... name=...>
      <group name=... />
      <user name=... />
    </security-role-mapping>
  </default-method-access>
</assembly-descriptor>
```

Each of the element groups are discussed in the following sections of the OC4J documentation set:

- The `<security-role-mapping>` element is described in ["Mapping Logical Roles to Users and Groups"](#) on page 22-8.

- The `<default-method-access>` element is described in ["Specifying a Default Role Mapping for Undefined Methods"](#) on page 22-9.

Element Description

`<assembly-descriptor>`

The mapping of the assembly descriptor elements.

`<cmp-field-mapping>`

Deployment information for a container-managed persistence field. If no subtags are used to define different behavior, the field is persisted through serialization or native handling of "recognized" primitive types.

Attributes:

- `ejb-reference-home` - The JNDI-location of the fields remote EJB-home if the field is an entity EJBObject or an EJBHome.
- `name` - The name of the field.
- `persistence-name` - The name of the field in the database table.
- `persistence-type` - The database type (valid values varies from database to database) of the field.

`<collection-mapping>`

Specifies a relational mapping of a Collection type. A Collection consists of n unordered items (order is not specified and not relevant). The field containing the mapping must be of type `java.util.Collection`.

Attributes:

- `table` - The name of the table in the database.

`<context-attribute>`

An attribute sent to the context. The only mandatory attribute in JNDI is the `'java.naming.factory.initial'` which is the classname of the context factory implementation.

Attributes:

- `name` - The name of the attribute.
- `value` - The value of the attribute.

`<data-bus>`

The name and url of a specific Databus for an OC4J object.

Attributes:

- `data-bus-name` - The user-defined name of the Databus.
- `url` - The URL of the Databus, which is similar to a JDBC URL.

`<default-method-access>`

The default method access policy for methods not tied to a method-permission.

`<description>`

A short description.

<ejb-name>

The `ejb-name` element specifies an enterprise bean's name. This name is assigned by the `ejb-jar` file producer to name the enterprise bean in the `ejb-jar` file's deployment descriptor. The name must be unique among the names of the enterprise beans in the same `ejb-jar` file. The enterprise bean code does not depend on the name; therefore the name can be changed during the application-assembly process without breaking the enterprise bean's function. There is no architected relationship between the `ejb-name` in the deployment descriptor and the JNDI name that the Deployer will assign to the enterprise bean's home. The name must conform to the lexical rules for an NMTOKEN.

<ejb-ref-mapping>

The `ejb-ref` element that is used for the declaration of a reference to another enterprise bean's home. The `ejb-ref-mapping` element ties this to a JNDI-location when deploying.

Attributes:

- `location` - The JNDI location to look up the EJB home from.
- `name` - The `ejb-ref`'s name. Matches the name of an `ejb-ref` in `ejb-jar.xml`.

<enterprise-beans>

The beans contained in this EJB JAR file.

<entity-deployment>

Deployment information for an entity bean.

Attributes:

- `call-timeout` - The time (long milliseconds in decimal) to wait for any resource that the EJB uses, except database connections, if it is busy (before throwing a `RemoteException`, treating it as a deadlock). This is also used as a SQL query timeout. If the timeout occurs before the SQL query finishes, a SQL exception is thrown. If zero, the timeout is disabled. The default is 90 seconds.
- `clustering-schema` - Not recommended to use.
- `copy-by-value` - Whether or not to copy all the incoming/outgoing parameters for all incoming and outgoing EJB calls. Set to 'false' if your application does not assume copy-by-value semantics for these parameters. The default is 'true'.
- `data-source` - The name of the data source used if using container-managed persistence.
- `delay-updates-until-commit` - Defers the flushing of transactional data until commit time or not. The default is true. If you want each change to be updated in the database, set this element to false.
- `do-select-before insert` - If false, you avoid executing a select before an insert. The extra select normally checks to see if the entity already exists before doing the insert to avoid duplicates.

If a unique key constraint is defined for the entity, then we recommend setting this to false. If there is no unique key constraint, setting this to false leads to not detecting a duplicate insert. To prevent duplicate inserts in this case, leave it set to true.

For performance, Oracle recommends setting this to false to avoid the extra select before insert. Default Value: true

- **exclusive-write-access** - Whether or not the EJB-server has exclusive write (update) access to the database back-end. This can be used only for entity beans that use a "read_only" locking mode. In this case, it increases the performance for common bean operations and enables better caching. The default is false. See ["Configuring Exclusive Write Access to the Database"](#) on page 14-10 for more information.
- **findByPrimaryKey-lazy-loading="true | false"** - For entity bean finder methods, lazy loading can cause the select method to be invoked more than once. To turn on lazy loading and enforce only a single execution of this finder method, set this property to true. The default is false. See ["Configuring Lazy Loading on Finder Methods"](#) on page 14-11 for more information.
- **isolation** - Specifies the isolation-level for database actions. The valid values for Oracle databases are 'serializable' and 'committed'. The default is 'committed'. Non-Oracle databases can be the following: 'none', 'committed', 'serializable', 'uncommitted', and 'repeatable_read'. For more information, see ["How do You Avoid Database Resource Contention?"](#) on page 1-24 and *Oracle Application Server Performance Guide*.
- **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 this bean will be bound to.
- **locking-mode** - The concurrency modes configure when to block to manage resource contention or when to execute in parallel. For more information, see ["How do You Avoid Database Resource Contention?"](#) on page 1-24 and *Oracle Application Server Performance Guide*. The concurrency modes are as follows:
 - **PESSIMISTIC**: 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.
 - **OPTIMISTIC**: Multiple users can execute the entity bean in parallel. It does not monitor resource contention; thus, the burden of the data consistency is placed on the database isolation modes. This is the default.
 - **READ-ONLY**: Multiple users can execute the entity bean in parallel. The container does not allow any updates to the bean's state.
- **max-instances** - The number of maximum bean implementation instances to be kept instantiated or pooled. The default is 0, which means infinite. See ["Configuring Bean Instance Pool Size"](#) on page 31-3 for more information.
- **min-instances** - The number of minimum bean implementation instances to be kept instantiated or pooled. The default is 0. See ["Configuring Bean Instance Pool Size"](#) on page 31-3 for more information.
- **max-tx-retries**—The number of times to retry a transaction that was rolled back due to system-level failures. The default is 0. Leave the setting to zero if using the serializable isolation level. Within a transaction, the container uses the max-tx-retries value of the first invoked bean within the transaction. The performance guide recommends that you leave this value at 0 and add retries only where errors are seen that could be resolved through a retry.
- **tx-retry-wait**—This parameter specifies the time to wait in seconds between retrying the transaction. The default is 60 seconds.
- **name** - The name of the bean, this matches the name of a bean in the assembly 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 'never' retains the instances until they are garbage collected. The default is 60.
- `table` - The name of the table in the database if using container-managed persistence.
- `validity-timeout` - The maximum amount of time (in milliseconds) that an entity is valid in the cache (before being reloaded). Useful for loosely coupled environments where rare updates from legacy systems occur. This attribute is only valid for entity beans with locking mode of `read_only` and when `exclusive-write-access="true"` (the default).

We recommend that if the data is never being modified externally (and therefore you've set `exclusive-write-access=true`), that you can set this to 0 or -1, to disable this option, since the data in the cache will always be valid for read-only EJBs that are never modified externally.

If the EJB is generally not modified externally, so you're using `exclusive-write-access=true`, yet occasionally the table is updated so you need to update the cache occasionally, then set this to a value corresponding to the interval you think the data may be changing externally.

- `update-changed-fields-only` - Specifies whether the container updates only modified fields or all fields to persistence storage for CMP entity beans when `ejbStore` is invoked. The default is true, which specifies to only update modified fields.
- `wrapper` - Name of the OC4J remote home wrapper class for this bean. (internal server attribute, do not edit)

<entity-ref>

Specifies the configuration for persisting an entity reference via its primary key. The child-tag of this tag is the specification of how to persist the primary key.

Attributes:

- `home` - JNDI location of the EJBHome to get lookup the beans at.

<env-entry-mapping>

Overrides the value of an `env-entry` in the assembly descriptor. It is used to keep the EAR clean from deployment-specific values. The body is the value.

Attribute:

- `name` - The name of the context parameter.

<fields>

Specifies the configuration of a field-based (java class field) mapping persistence for this field. The fields that are to be persisted have to be public, non-static, non-final and the type of the containing object has to have an empty constructor.

<finder-method>

The definition of a container-managed finder method. This defines the selection criteria in a `findByXXX()` method in the bean's home.

Attributes:

- `partial` - Whether or not the specified query is a partial one. A partial query is the 'where' clause or the 'order' (if it starts with order) clause of the SQL query. Queries are partial by default. If `partial="false"` is specified then the full query is to be entered as value for the query attribute and you need to make sure that the

query produces a result-set containing all of the CMP fields. This is useful when doing advanced queries involving table joins and similar.

- query - The query part of an SQL statement. This is the section following the WHERE keyword in the statement. Special tokens are \$number which denotes an method argument number and \$name which denotes a cmp-field name. For instance the query for "findByAge(int age)" would be (assuming the cmp-field is named 'age'): "\$1 = \$age".
- lazy-loading - For entity bean finder methods, lazy loading can cause the select method to be invoked more than once. To turn on lazy loading and enforce only a single execution of this finder method, set this property to true. The default is false. See ["Configuring Lazy Loading on Finder Methods"](#) on page 14-11 for more information.
- prefetch-size - Oracle JDBC drivers include extensions that allow you to set the number of rows to prefetch into the client while a result set is being populated during a query. This reduces round trips to the database by fetching multiple rows of data each time data is fetched—the extra data is stored in client-side buffers for later access by the client. The number of rows to prefetch can be set as desired. The default number of rows to prefetch to the client is 10. The number set here is passed along to the JDBC driver. See the *Oracle Database JDBC Developer's Guide and Reference* for more information on using prefetch with a JDBC driver.

<group>

A group that this <security-role-mapping> implies. That is, all members of the specified group are included in this role.

Attributes:

- name - The name of the group.

<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*.

<lookup-context>

The specification of an optional `javax.naming.Context` implementation used for retrieving the resource. This is useful when using third party modules, such as a third party JMS server. Either use the context implementation supplied by the resource vendor or, if none exists, write an implementation that negotiates with the vendor software.

Attribute:

- location - The name looked for in the foreign context when retrieving the resource.

<map-key-mapping>

Specifies a mapping of the map key. Map keys are always immutable.

Attributes:

- type - The fully qualified class name of the type of the value. Examples are `com.acme.Product`, `java.lang.String`, and so on.

<message-driven-deployment>

Deployment information for a MDB.

Attributes:

- **connection-factory-location:** The JNDI location of the connection factory to use. The JMS Destination Connection Factory is specified in the `connection-factory-location` attribute. The syntax is `"java:comp/resource" + resource provider name + "TopicConnectionFactory" or "QueueConnectionFactory" + user defined name`. The `xxxConnectionFactory` details what type of factory is being defined.
- **destination-location:** 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.
- **name -** The name of the bean, this matches the name of a bean in the assembly descriptor (`ejb-jar.xml`).
- **subscription-name:** If this is a topic, the subscription name is defined in the `subscription-name` attribute.
- **listener-threads:** 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 thread.
- **transaction-timeout:** 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.
- **dequeue-retry-count—**Specifies how often the listener thread tries to re-acquire the JMS session once database failover has incurred. This value is only for CMT transactions in an MDB. The default is "0." See ["Understanding OC4J EJB Application Clustering Services"](#) on page 2-20 for more information.
- **dequeue-retry-interval—**Specifies the interval between retries. The default is 60 seconds.

<method>

Specify the methods (and possibly parameters of that method) of the bean.

<method-intf>

The `method-intf` element allows a method element to differentiate between the methods with the same name and signature that are defined in both the remote and home interfaces. The `method-intf` element must be one of the following: `Home` or `Remote`.

<method-name>

The `method-name` element contains a name of an enterprise bean method, or the asterisk (*) character. The asterisk is used when the element denotes all the methods of an enterprise bean's remote and home interfaces.

<method-param>

The `method-param` element contains the fully-qualified Java type name of a method parameter.

<method-params>

The `method-params` element contains a list of the fully-qualified Java type names of the method parameters.

<orion-ejb-jar>

An `orion-ejb-jar.xml` file contains the OC4J-specific deployment information for an EJB. It is used to specify initial deployment properties. After each deployment the deployment file is reformatted and altered by the server for additional information.

Attributes:

- `deployment-time` - The time (long milliseconds in decimal) of the last deployment, if not matching the last editing date the JAR will be redeployed. (internal server value, do not edit)
- `deployment-version` - The version of OC4J this JAR was deployed with, if it's not matching the current version then it will be redeployed. (internal server value, do not edit)

<primkey-mapping>

Designates how the primary key is mapped.

<properties>

Specifies the configuration of a property-based (bean properties) mapping persistence for this field. The properties have to adhere to the usual JavaBeans specification and the type of the containing object has to have an empty constructor. This is also designated within the EJB specification.

<resource-ref-mapping>

The `resource-ref` element is used for the declaration of a reference to an external resource such as a data source, JMS queue, or mail session. The `resource-ref-mapping` ties this to a JNDI-location when deploying.

Attributes:

- `location` - The JNDI location to look up the resource factory from.
- `name` - The `resource-ref` name. Matches the name of an `resource-ref` in `ejb-jar.xml`.

<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.

Attributes:

- `location` - The JNDI location from which to look up the administered resource.
- `name` - The `resource-env-ref` name in `ejb-jar.xml`.

<security-role-mapping>

The runtime mapping (to groups and users) of a role. Maps to a security-role of the same name in the assembly descriptor.

Attributes:

- `impliesAll` - Whether or not this mapping implies all users. The default is false.
- `name` - The name of the role

<session-deployment>

Deployment information for a session bean.

Attributes:

- **pool-cache-timeout**—How long to keep stateless sessions cached in the pool. Only applies to stateless session beans. Legal values are positive integer values or 'never'. For stateless session beans, if you specify a pool-cache-timeout, then at every pool-cache-timeout interval, all beans in the pool, of the corresponding bean type, 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.

Default Value: 60 (seconds)

- **call-timeout**—The time (long milliseconds in decimal) to wait for any resource that the EJB uses, excluding database connections, if it is busy. After this times out, a `RemoteException` is thrown and the EJB is treated as involved in a deadlock. If value is set to 0, OC4J waits for the EJB "forever". The default is 90,000.
- **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 is 'true'.
- **local-wrapper**—Name of the OC4J wrapper class for this bean. This is an internal server value and should not be edited.
- **location**—The JNDI-name that this bean will be bound to.

max-instances - This attribute controls the number of bean instances allowed in memory—either instantiated or pooled. When this value is reached, the container attempts to passivate the oldest bean instance from memory. If unsuccessful, the container waits the number of milliseconds set in the `call-timeout` attribute to see if a bean instance is removed from memory, either through passivation, its `remove()` method, or bean expiration, before a `TimeoutExpiredException` is thrown back to the client. To allow an infinite number of bean instances, the `max-instances` attribute can be set to zero. Default is 0, which is infinite. This applies to both stateless and stateful session beans.

- **max-instances-threshold** - This attribute defines the percentage of `max-instances` number of beans that can be in memory before passivation occurs. When this threshold is reached, passivation of beans occurs. For example, if `max-instances` is 100 beans, when `max-instances-threshold` reaches 90, OC4J begins passivation.
- **max-tx-retries**—The number of times to retry a transaction that was rolled back due to system-level failures. The default is 0. Within a transaction, the container uses the `max-tx-retries` value of the first invoked bean within the transaction. The performance guide recommends that you leave this value to 0 and add retries only where errors are seen that could be resolved through a retry.
- **tx-retry-wait**—This parameter specifies the time to wait in seconds between retrying the transaction. The default is 60 seconds.
- **memory-threshold** - 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 "never."
- **min-instances** - The number of minimum bean implementation instances to be kept instantiated or pooled. The default is zero. This applies only to stateless session beans.

- **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`).
- **resource-check-interval** - 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 "never."
- **passivate-count** - 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**—Path to the file where sessions are stored across restarts.
- **timeout**—Inactivity timeout in seconds. If the value is zero or negative, then all timeouts are disabled. The default is 30 minutes. 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 30 minutes, then adjust the timeout value accordingly.
- **wrapper**—Name of the OC4J wrapper class for this bean. This is an internal server value and should not be edited.

<set-mapping>

Specifies a relational mapping of a Set type. A Set consists of *n* unique unordered items (order is not specified and not relevant). The field containing the mapping must be of type `java.util.Set`.

Attributes:

- **table** - The name of the table in the database.

<user>

A user that this security-role-mapping implies.

Attributes:

- **name** - The name of the user.

<value-mapping>

Specified a mapping of the primary key part of a set of fields.

Attributes:

- **immutable** - Whether or not the value can be trusted to be immutable once added to the `Collection`. Setting this to true will optimize database operations extensively. The default value is "true" for set-mapping and "false" for collection-mapping.
- **type** - The fully qualified class name of the type of the value. Examples are `com.acme.OrderEntry`, `java.lang.String`, and so on.

Glossary

This glossary defines terms specific to OC4J. For general, J2EE terminology, see <http://java.sun.com/j2ee/reference/glossary/>.

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 via 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>.

Index

Symbols

@ActivationConfigurationProperty, 9-2, 10-2, 10-4
@AroundInvoke, 5-3, 10-6
@AttributeOverride, 7-13
@Basic, 7-8
@Column, 7-6
@DeclareRoles, 22-12
@DenyAll, 22-12
@EJB, 1-7
@Embeddable, 7-12
@Embedded, 7-12
@GeneratedIdTable, 7-3
@Id, 7-2
@Inheritance, 7-17
@InheritanceJoinColumn, 7-17
@JoinColumn, 7-7
@Lob, 7-9
@ManyToMany, 7-12
@ManyToOne, 7-11
@MessageDriven, 9-2, 10-2, 10-3
@MessageDrivenDeployment, 10-3
@NamedQuery, 8-1
@OneToMany, 7-11
@OneToOne, 7-10
@PermitAll, 22-6, 22-12
@PostActivate, 5-2
@PostConstruct, 5-2, 10-6
@PostLoad, 7-15
@PostPersist, 7-15
@PostRemove, 7-15
@PostUpdate, 7-15
@PreDestroy, 5-2, 10-6
@PrePassivate, 5-2
@PrePersist, 7-15
@PreRemove, 7-15
@PreUpdate, 7-15
@Resource, 1-7
@RolesAllowed, 22-4, 22-12
@RunAs, 22-7, 22-12
@SecondaryTable, 7-6
@SequenceGenerator, 7-4
@Serialized, 7-9
@Table, 7-5
@TableGenerator, 7-4
@Transient, 1-14

@Version, 7-14

A

<abstract-schema-name> element, 16-1, 16-5
accessing EJBs
 in another application, 29-5, 29-17
aggregate object relational mappings
 understanding, 7-12
annotations
 @ActivationConfigurationProperty, 9-2, 10-2, 10-4
 @AroundInvoke, 5-3, 10-6
 @AttributeOverride, 7-13
 @Basic, 7-8
 @Column, 7-6
 @DeclareRoles, 22-12
 @DenyAll, 22-12
 @EJB, 1-7
 @Embeddable, 7-12
 @Embedded, 7-12
 @GeneratedIdTable, 7-3
 @Id, 7-2
 @Inheritance, 7-17
 @InheritanceJoinColumn, 7-17
 @JoinColumn, 7-7
 @Lob, 7-9
 @Local, 4-2, 4-3
 @ManyToMany, 7-12
 @ManyToOne, 7-11
 @MessageDriven, 9-2, 10-2, 10-3
 @MessageDrivenDeployment, 10-3
 @NamedQuery, 8-1
 @OneToMany, 7-11
 @OneToOne, 7-10
 @PermitAll, 22-6, 22-12
 @PostActivate, 5-2
 @PostConstruct, 5-2, 10-6
 @PostLoad, 7-15
 @PostPersist, 7-15
 @PostRemove, 7-15
 @PostUpdate, 7-15
 @PreDestroy, 5-2, 10-6
 @PrePassivate, 5-2
 @PrePersist, 7-15
 @PreRemove, 7-15

- @PreUpdate, 7-15
- @Remote, 4-2, 4-3
- @Remove, 4-3
- @Resource, 1-7
- @RolesAllowed, 22-4, 22-12
- @RunAs, 22-7, 22-12
- @SecondaryTable, 7-6
- @SequenceGenerator, 7-4
- @Serialized, 7-9
- @Stateful, 4-3
- @Stateless, 4-2
- @Table, 7-5
- @TableGenerator, 7-4
- @Transient, 1-14
- @Version, 7-14
- about, 1-7
- application.xml file
 - example, 2-6
 - modifying, 2-6
 - overview, 2-6
- archiving
 - directions, 2-6
 - EAR file, 2-7
 - EJBs, 2-5
- <assembly-descriptor> element, A-18, A-19

B

- bean
 - accessing remotely, 1-3, 1-4
 - activation, 1-9, 1-11, 12-3
 - creation, 12-3
 - environment, 1-6
 - implementing, CMP, EJB 2.1, 13-2, 13-7, 17-1
 - passivation, 1-11, 12-3
 - steps for invocation, 1-3, 1-5
- bean implementation
 - EJB 2.1, overview, 1-4
 - EJB 3.0, overview, 1-2
- bean-managed transactions
 - about, 2-15
- BMP
 - commit options, 1-27
 - database schema, 13-3, 13-8
 - ejbCreate implementation, 13-15
 - read-only and commit option A, 1-27, 15-1
- BMP entity bean
 - read-only, 15-1

C

- cache-timeout attribute, A-16
- call-timeout attribute, A-7, A-11, A-20, A-26
- child EJB, 29-5, 29-17
- ClassCastException, 27-2
- client
 - accessing EJB 3.0 entity, 29-5
- clients
 - about, 29-1
 - EJB, 29-2

- JSP, 29-2
- servlet, 29-2
- stand-alone Java, 29-2
- clustering services
 - about, 2-20
 - DNS load balancing, about, 2-22
 - DNS load balancing, configuring, 24-4
 - failover, 2-22
 - HTTP and stateful session bean
 - combination, 2-20
 - HTTP sessions, 2-20
 - load balancing, about, 2-22
 - replication-based load balancing, about, 2-22
 - replication-based load balancing,
 - configuring, 24-3
 - state replication, 2-20
 - state replication, inherited, 2-21
 - state replication, on end of request, 2-21
 - state replication, on shutdown, 2-21
 - stateful session beans, 2-21
 - static retrieval load balancing, about, 2-22
 - static retrieval load balancing, configuring, 24-3
- clustering-schema attribute, A-11, A-20
- CMP
 - commit options, 1-26
 - overview, 1-19, 1-22
- <cmp-field-mapping> element, A-10, A-19
- CMT
 - retry JMS message dequeue, A-16, A-24
- <collection-mapping> element, A-19
- command-line options, 28-2
- commit options
 - A and read-only BMP, 1-27, 15-1
 - about, 1-26
 - BMP, 1-27
 - CMP, 1-26
- <commit-option> element, A-11
- component interface
 - EJB 2.1, overview, 1-4
 - EJB 3.0, overview, 1-2
- composite primary key
 - about, 1-21
 - class, 14-3
- concurrency mode, 1-25
 - optimistic, 1-25
 - pessimistic, 1-25
 - read-only, 1-25
- concurrency modes, 1-24
- <config-property> element, A-15
- config-property
 - ConnectionFactoryTimeout, A-15
 - DequeueRetryCount, A-16
 - DequeueRetryInterval, A-16
 - DestinationLocation, A-15
 - ListenerThreads, A-15
 - MaxDeliveryCnt, A-16
 - SubscriptionName, A-15
 - TransactionTimeout, A-16
- connectionFactoryJndiName attribute, 10-3, 10-4
- connection-factory-location attribute, A-15, A-24

- ConnectionFactoryTimeout config-property, A-15
- container-managed persistence. *see* CMP
- container-managed transactions
 - about, 2-15
 - rollback, 21-5
- <container-transaction> element, 17-2, 17-5
- context
 - entity, 1-24
 - entity bean, 13-20
 - message-driven bean, 1-35
 - session, 1-6, 1-13
 - session bean, 11-10
 - transaction, 1-6
- <context-attribute> element, A-19
- copy-by-value attribute, A-7, A-11, A-20, A-26
- create method
 - EJBHome interface, 1-5, 11-7
 - home interface, 13-18
- CreateException, 11-7, 11-8, 13-19

D

- data sources
 - about, 2-13
 - managed, 2-13
 - native, 2-13
- database resource contention
 - concurrency mode, 1-25
 - transaction isolation, 1-24
- <data-bus> attribute, A-19
- data-source attribute, A-11, A-20
- DataSource object, 19-7
- data-sources.xml file, 13-8, 13-15
- Date, 16-8
- DBMS_AQADM package, 23-4
- deadlock
 - recovery, 29-23
- debugging
 - ejb-jar.xml, 31-5
 - generated code, 31-6
 - validating XML, 31-5
 - wrapper code, 31-6
 - WrapperCodeDir, 31-6
- dedicated.rmicontext property, 29-23
- default mapping
 - default table generator, 14-5
- default table generator
 - default mapping, 14-5
- <default-method-access> element, 22-9, A-19
- <delay-updates-until-commit> attribute, A-20
- delay-updates-until-commit attribute, A-13
- deployment
 - ejb-jar.xml creation, 26-1
 - error recovery, 28-1
 - incremental, 28-2
- deployment descriptor, 2-4
 - EJB 2.1, overview, 1-4
 - EJB 3.0, overview, 1-3
 - entity bean, A-9, A-10
 - JDBC DataSource, 19-2

- message-driven bean, A-14
- persistence manager, A-5
- security, 22-2, 22-3, 22-8
- session bean, A-6
- deployment descriptors
 - ejb-jar.xml, creating at deployment time, 26-1
 - ejb-jar.xml, creating at migration time, 26-1
 - ejb-jar.xml, creating with JDeveloper, 26-1
 - orion-ejb-jar.xml, configuration, 26-2
 - toplink-ejb-jar.xml, configuration, 26-2
 - toplink-ejb-jar.xml, creating at migration time, 26-2
 - toplink-ejb-jar.xml, creating with TopLink Workbench, 26-2
- dequeue-retry-count attribute, 18-8, A-16, A-24
- DequeueRetryCount config-property, A-16
- dequeue-retry-interval attribute, 18-8, A-16, A-24
- DequeueRetryInterval config-property, A-16
- <description> element, A-19
- destination-location attribute, A-15, A-24
- DestinationLocation config-property, A-15
- destinationName attribute, 10-3, 10-4
- <destination-type> element, 17-5
- destinationType attribute, 10-3, 10-4
- do-select-before-insert attribute, A-12, A-20

E

- EAR file
 - creation, 2-7
- EJB
 - archive, 2-5
 - client
 - setting JMS port, 29-2
 - setting RMI port, 29-2
 - deployment descriptor, 2-4
 - development suggestions, 2-1
 - difference between session and entity, 1-36
 - home interface, 11-7
 - implementing, CMP, EJB 2.1, 13-2, 13-7, 17-1
 - local interface, 11-9, 13-20
 - looking up, EJB 2.1, about, 19-24
 - looking up, EJB 3.0, about, 19-19
 - looking up, EJB 3.0, using annotations, 19-19, 19-22
 - looking up, local interface using
 - ejb-local-ref, 19-21, 19-25
 - looking up, local interface using
 - local-location, 19-21, 19-26
 - looking up, remote interface using ejb-ref, 19-20, 19-24
 - looking up, remote interface using
 - location, 19-20, 19-25
 - parameter passing, 29-21
 - passivation, 1-11
 - pool size, entity beans, 31-3
 - pool size, session beans, 31-3
 - pool timeouts, entity beans, 31-4
 - pool timeouts, session beans, 31-4
 - queries, about, 1-16, 1-27

- queries, EJB QL, 1-17, 1-27
- queries, EntityManager, 1-18
- queries, finder methods, 1-30
- queries, select methods, 1-32
- queries, SQL, 1-17, 1-29
- queries, syntax, 1-16, 1-27
- queries, TopLink, 1-28
- referencing other EJBs, 27-1, 27-2
- remote interface, 11-9, 13-19
- replication, 24-2
- security, 22-1
- standalone client, 29-2
- <ejb> element, 2-6
- EJB 2.1
 - CMP entity bean, configuration, 14-1, 15-1
 - composite primary key, class, 14-3
 - composite primary key, configuring, 14-3
 - JDK required, 3-4
 - MDB, configuration, 18-1
 - message-driven bean, configuration, 18-1
 - persistence, 3-4
 - persistence manager, 3-4
 - persistence manager customization, 3-5
 - primary key, configuring, 14-1
 - sequencing, configuration, 14-4
 - session bean, configuration, 12-1
 - stateless session bean, implementing, 11-1, 11-4, 13-1, 13-6, 17-1
 - support, 3-4
- EJB 3.0
 - CMP entity bean, configuration, 7-1
 - defining an EJB 3.0 application, 3-2
 - entity manager, 3-2
 - entity manager customization, 3-2
 - EntityManager, about, 1-16
 - JDK required, 3-1
 - MDB, configuration, 10-1
 - message-driven bean, configuration, 10-1
 - persistence, 3-2
 - primary key, automatic generation, 7-3
 - primary key, configuring, 7-2
 - primary key, sequencing, 7-3
 - sequencing, configuration, 7-3
 - session bean, configuration, 5-1
 - stateful session bean, implementing, 4-2
 - stateless session bean, implementing, 4-1
 - support, 3-1
- EJB finders
 - default finders, about, 1-31
- EJB QL
 - about, 1-17, 1-27
- EJB services
 - clustering, about, 2-20
 - clustering, DNS load balancing, 2-22, 24-4
 - clustering, failover, 2-22
 - clustering, HTTP sessions, 2-20
 - clustering, load balancing, 2-22
 - clustering, replication-based load balancing, 2-22, 24-3
 - clustering, state replication, 2-20
 - clustering, stateful session beans, 2-21
 - clustering, static retrieval load balancing, 2-22, 24-3
- EJB support
 - EJB 2.1, 3-4
 - EJB 3.0, 3-1
- ejb3-toplink-sessions.xml
 - about, 2-9
 - XSD, 2-9
- ejbActivate method, 1-9, 1-11, 1-21, 1-23, 12-3, 15-5, 15-6
- EJBContext
 - setRollbackOnly, 21-5
- EJBContext interface, 1-6
- ejbCreate method, 1-21, 1-23, 11-7, 12-3, 13-15
 - initializing primary key, 13-15
 - SessionBean interface, 1-9, 1-11, 1-35
- EJBException, 11-7, 11-8, 11-9, 13-19, 13-20
- ejbFindByPrimaryKey method, 13-15, 15-3
- EJBHome interface, 11-2, 11-4, 11-7, 13-2, 13-6, 13-18, 13-19
 - create method, 13-18
- ejb-jar.xml
 - about, 2-7
 - creating at deployment time, 26-1
 - creating at migration time, 26-1
 - creating with JDeveloper, 26-1
 - XSD, EJB 2.1, 2-8
 - XSD, EJB 3.0, 2-8
- ejb-jar.xml file, 2-4
- <ejb-link> element, 19-5
- ejbLoad method, 1-21, 1-23, 15-5
- EJBLocalHome interface, 11-2, 11-4, 11-8, 13-2, 13-6, 13-18, 13-19
- EJBLocalObject interface, 11-2, 11-4, 11-9, 13-2, 13-7, 13-19, 13-20
- <ejb-location> element, 13-15
- <ejb-mapping> element, 19-5
- <ejb-module> element, 29-15, 29-16
- <ejb-name> element, 19-5, A-20
- EJBObject interface, 11-2, 11-4, 11-9, 13-2, 13-7, 13-19
- ejbPassivate method, 1-9, 1-11, 1-21, 1-23, 12-3, 15-5
- ejbPostCreate method, 1-21, 1-23
- <ejb-ql> element, 16-2, 16-5
- <ejb-ref> element, 19-5
- ejb-reference-home attribute, A-19
- <ejb-ref-mapping> element, A-6, A-10, A-14, A-20
- <ejb-ref-name> element, 19-5, 29-4
- ejbRemove method, 1-9, 1-11, 1-21, 1-23, 1-35, 12-3, 15-6
- ejbStore method, 1-21, 1-23, 15-4
- enable-passivation attribute, 12-2, 12-3
- <enterprise-beans> element, A-4, A-20
- entity
 - lifecycle methods, EJB 3.0, 1-15
 - overview, 1-14
 - PostLoad annotation, 1-16
 - PostPersist annotation, 1-15
 - PostRemove annotation, 1-15
 - PostUpdate annotation, 1-15

- PrePersist annotation, 1-9, 1-11, 1-15, 1-34
- PreRemove annotation, 1-15
- PreUpdate annotation, 1-15
- entity bean
 - commit options, A, 1-27, 15-1
 - commit options, about, 1-26
 - commit options, and CMP, 1-26
 - commit options, BMP, 1-27
 - context, 1-24, 13-20
 - context information, 13-20, 17-6
 - creating, 13-18
 - deployment descriptor, A-9, A-10
 - EJB 2.1 CMP, configuration, 14-1, 15-1
 - EJB 3.0 CMP, configuration, 7-1
 - EJB 3.0 see entity, 1-14
 - finder methods, 13-18
 - about, 13-15
 - home interface, 13-18
 - lifecycle methods, EJB 2.1, 1-20
 - lifecycle methods, EJB 3.0, 1-15
 - overview, 1-18
 - primary key, 1-23
 - remote interface, 13-19
- entity context, 1-24
- Entity Manager
 - queries, about, 1-18
- entity manager
 - about, 3-2
 - customization, 3-2
 - TopLink customization, 3-2
- EntityBean interface
 - ejbActivate method, 1-21, 1-23
 - ejbCreate method, 1-21, 1-23
 - ejbLoad method, 1-21, 1-23
 - ejbPassivate method, 1-21, 1-23
 - ejbPostCreate method, 1-21, 1-23
 - ejbRemove method, 1-21, 1-23
 - ejbStore method, 1-21, 1-23
 - setEntityContext method, 13-20, 17-6
- <entity-deployment> element, 14-10, A-9, A-10, A-20
- entity-deployment
 - call-timeout attribute, A-11
 - clustering-schema attribute, A-11
 - copy-by-value attribute, A-11
 - data-source attribute, A-11
 - delay-updates-until-commit attribute, A-13
 - do-select-before-insert attribute, A-12
 - exclusive-write-access attribute, A-12
 - findByPrimaryKey-lazy-loading attribute, A-13
 - force-update attribute, A-13
 - isolation attribute, A-12
 - local-location attribute, A-12
 - local-wrapper attribute, A-13
 - location attribute, A-12
 - locking-mode attribute, A-12
 - max-instances attribute, A-12
 - max-tx-retries attribute, A-13
 - min-instances attribute, A-12
 - name attribute, A-13
 - pool-cache-timeout attribute, A-13
 - table attribute, A-13
 - update-changed-fields-only attribute, A-13
 - validity-timeout attribute, A-13
 - wrapper attribute, A-13
- EntityManager
 - about, 1-16
 - accessing an EJB 3.0 entity, 29-5
- <entity-ref> element, A-22
- <env-entry> element, 19-13
- <env-entry-mapping> element, A-6, A-10, A-14, A-22
- <env-entry-name> element, 19-13
- <env-entry-type> element, 19-14
- <env-entry-value> element, 19-14
- environment references, URL, 19-12, 19-14
- environment variables
 - configuring, 19-13
 - ejb-jar.xml, 19-13
 - looking up, EJB 2.1, 19-27
 - looking up, EJB 3.0, 19-23
 - orion-ejb-jar.xml, 19-14
 - overriding, 19-14
 - resource injection, 19-23
- environment, retrieval, 1-6
- error recovery, 27-2, 28-1
 - ClassCastException, 27-2
 - deadlock, 29-23
 - NamingException thrown, 29-22
 - NullPointerException thrown, 29-23
 - out of memory, 28-1
- exception queue, 18-7, A-16
- exclusive-write-access attribute, 14-10, A-12, A-21

F

- features, EJB, 3-5
- <fields> element, A-22
- findByPrimaryKey-lazy-loading attribute, 14-11, A-13, A-21
- <finder-method> element, A-10
- finder
 - lazy loading, 14-11
- finder methods, 13-15
 - about, 1-30
 - BMP, 15-3
 - entity bean, 13-18
- <finder-method> element, A-22
- force-update attribute, A-13

G

- generated code
 - debugging, 31-6
- getEJBHome method, 1-6
- getEnvironment method, 1-6
- getRollbackOnly method, 1-6
- getUserTransaction method, 1-6
- <group> element, A-23

H

home interface
 creating, 11-2, 11-4, 13-2, 13-6
 EJB 2.1, overview, 1-4
 EJB 3.0, overview, 1-2
HTTP sessions
 state replication, 2-20

I

idletime attribute, A-8
immutable attribute, A-27
impliesAll attribute, 22-9, A-25
incremental deployment, 28-2
injection, 1-7
interceptors
 configuring, message-driven bean, 10-5
 configuring, session beans
 session bean
 interceptors, configuring, 5-3
 InvocationContext, 2-12
 restrictions, 2-11
 signature, 2-12
 transactions, 2-12
 understanding, 2-11
InvocationContext, 2-12
<ior-security-config> element, A-6, A-10, A-23
isCallerInRole method, 22-3
isolation
 transaction levels, 1-24
isolation attribute, A-12, A-21
isolation modes, 1-24

J

JAAS, 22-12
JAR
 archiving command, 2-6
JAR file
 EJB, 2-5
<java> element, 2-6
Java mail
 Session object, 19-10
JDeveloper
 ejb-jar.xml creation, 26-1
JDK
 EJB 2.1, 3-4
 EJB 3.0, 3-1
JMS
 Destination, 23-5
 durable subscriptions, 17-2
 exception queue, 18-7, A-16
 port, 29-2
 retry message dequeue, A-24
<jndi-name> element, 19-5, 19-11, 19-13
JSR250, 22-12

L

lazy loading, 14-11

lazy-loading attribute, 14-12, A-23
lifecycle methods
 entity bean, EJB 2.1, 1-20
 entity, EJB 3.0, 1-15
 message-driven bean, EJB 2.1, 1-34
 message-driven bean, EJB 3.0, 1-34
 session bean, stateless, 1-22
 stateful session bean, EJB 2.1, 1-9, 1-11
 stateful session bean, EJB 3.0, 1-9, 1-10
listener threads, 18-6
listener-threads attribute, 18-6, 18-7, A-15, A-24
ListenerThreads config-property, A-15
load balancing
 clustering, and, 2-22
 DNS, 2-22, 24-4
 replication-based, 2-22, 24-3
 static retrieval, 2-22, 24-3
local access, 29-16
local home interface
 example, 11-8
local interface
 creating, 11-9, 13-20
 EJB 2.1, overview, 1-4
 EJB 3.0, overview, 1-2
 example, 11-9
local-location attribute, A-7, A-12
local-wrapper attribute, A-9, A-13, A-21, A-26
location attribute, A-7, A-12, A-21, A-23, A-25, A-26
locking
 optimistic, 1-25
 pessimistic, 1-25
locking-mode attribute, A-12, A-21
look up
 EJB 2.1, about, 19-24
 EJB 3.0, about, 19-19
 EJB 3.0, using annotations, 19-19, 19-22
 remote interface using ejb-local-ref, 19-21, 19-25
 remote interface using ejb-ref, 19-20, 19-24
 remote interface using local-location, 19-21, 19-26
 remote interface using location, 19-20, 19-25
<lookup-context> element, A-23

M

mail
 Session object, 19-10
managed data sources, 2-13
many-to-many relational mappings
 understanding, 7-11
<map-key-mapping> element, A-23
<mapping> element, 19-5, 19-11, 19-13
mapping, 1-14, 1-20
MaxDeliveryCnt config-property, A-16
max-delivery-count attribute, A-16
max-instances attribute, 31-3, A-7, A-12, A-16, A-21, A-26
max-instances-threshold attribute, A-8, A-26
max-tx-retries attribute, A-7, A-13, A-21, A-26
MDB
 dequeue-retry-count attribute, A-24

- dequeue-retry-interval attribute, A-24
- EJB 2.1, configuration, 18-1
- EJB 3.0, configuration, 10-1
- onMessage method, 2-18
- overview, 1-33
- performance, A-24
- transaction timeout, A-24
- memory-threshold attribute, A-8, A-26
- <message-destination-ref-mapping> element, A-11, A-15
- <message-driven> element, 17-4
- message-driven bean
 - context, 1-35
 - deployment descriptor, A-14
 - EJB 2.1, configuration, 18-1
 - EJB 3.0, configuration, 10-1
 - interceptors, configuring, 10-5
 - lifecycle methods, EJB 2.1, 1-34
 - lifecycle methods, EJB 3.0, 1-34
- message-driven beans
 - listener threads, 18-6
 - transaction timeouts, 21-2
- Message-Driven Beans, see MDB
- message-driven context, 1-35
- <message-driven-deployment> element, A-14, A-23
- message-driven-deployment
 - cache-timeout attribute, A-16
 - connection-factory-location attribute, A-15
 - ConnectionFactoryTimeout
 - config-property, A-15
 - dequeue-retry-count attribute, A-16
 - DequeueRetryCount config-property, A-16
 - dequeue-retry-interval attribute, A-16
 - DequeueRetryInterval config-property, A-16
 - destination-location attribute, A-15
 - DestinationLocation config-property, A-15
 - listener-threads attribute, A-15
 - ListenerThreads config-property, A-15
 - MaxDeliveryCnt config-property, A-16
 - max-delivery-count, A-16
 - max-instances attribute, A-16
 - min-instances attribute, A-16
 - name attribute, A-15
 - resource-adapter attribute, A-15
 - subscription-name attribute, A-15
 - SubscriptionName config-property, A-15
 - transaction-timeout attribute, A-16
 - TransactionTimeout config-property, A-16
- <message-driven-destination> element, 17-5
- messageSelector attribute, 10-3, 10-4
- <method> element, A-17, A-18, A-24
 - defined, 22-5
- <method-intf> element, A-24
- <method-name> element, 16-2, 16-5, A-24
- <method-param> element, A-24
- <method-params> element, A-24
- <method-permission> element, 22-2, 22-3, 22-4, 22-5, 22-6
- middle-tier coordinator, 2-16
- migration

- ejb-jar.xml creation, 26-1
- toplink-ejb-jar.xml creation, 26-2
- migration, TopLink persistence manager, 3-5
- min-instances attribute, 31-3, A-7, A-12, A-16, A-21, A-26
- <module> element, 2-6
- multi-tier environment
 - local accessing, 29-16
 - remote accessing, 29-14

N

- name attribute, A-8, A-13, A-15, A-21, A-24, A-25, A-27
- native data sources, 2-13
- new features, 3-5
- non-batch mode, 28-2
- NoSuchObjectLocalException, 25-5
- NullPointerException, 29-23

O

- OC4J
 - command-line options, 28-2
 - Windows shutdown, 18-4
- OJMS
 - two-phase commit, 23-4
- one-to-many relational mappings
 - understanding, 7-11
- onMessage method, 2-18
- optimistic concurrency mode, A-21
- optimistic locking, 1-25
- OracleAS JMS
 - exception queue, 18-7, A-16
- oracle.j2ee.rmi.loadBalance, 24-3
- oracle.mdb.fastUndeploy property, 18-4
- <orion-ejb-jar> element, A-4, A-25
- orion-ejb-jar.xml
 - about, 2-8
 - configuration, 26-2
 - XSD, 2-8
- orion-ejb-jar.xml file, 17-2
- out of emory error
 - during deployment, 28-1
- out of memory error
 - during execution, 27-2

P

- packaging
 - referenced EJB classes, 27-1, 27-2
- parameters
 - object types, 29-22
 - passing conventions, 29-21
- parent application, 27-1
- parent EJB, 29-5, 29-17
- partial attribute, A-22
- pass by reference, 29-22
- pass by value, 29-22
- passivate-count attribute, A-9, A-27
- passivation

- about, 1-10
- ejbPassivate method, 1-9
- passivation criteria, 1-11 to 1-13
- permissions, 22-1
- persistence
 - container-managed, 1-19, 1-22
 - database schema, BMP, 13-3, 13-8
- persistence manager
 - about, 3-4
 - customization, 3-5
 - deployment descriptor, A-5
 - Orion, 3-4
 - TopLink customization class, 3-5
 - TopLink, migration, 3-5
- persistence-filename attribute, A-8, A-27
- <persistence-manager> element, A-5
- persistence-name attribute, A-19
- persistence-type attribute, A-19
- pessimistic concurrency mode, A-21
- pessimistic locking, 1-25
- pool
 - size, entity beans, 31-3
 - size, session beans, 31-3
 - timeouts, entity beans, 31-4
 - timeouts, session beans, 31-4
- pool-cache-timeout attribute, 31-4, 31-5, A-6, A-13, A-22, A-26
- PostLoad annotation, 1-16
- PostPersist annotation, 1-15
- PostRemove annotation, 1-15
- PostUpdate annotation, 1-15
- prefetch-size attribute, A-23
- PrePersist annotation, 1-9, 1-11, 1-15, 1-34
- PreRemove annotation, 1-15
- PreUpdate annotation, 1-15
- primary key
 - about, 1-16, 1-21
 - automatic generation, 7-3
 - complex class, 13-17
 - complex definition, 13-16
 - composite EJB 2.1, configuring, 14-3
 - composite, about, 1-21
 - creating, 13-15
 - EJB 2.1, configuring, 14-1
 - EJB 3.0, configuring, 7-2
 - overview, 1-23
 - sequencing, 7-3
 - simple definition, 13-16
- <prim-key-class> element, 13-16
- <primkey-mapping> element, A-10, A-25
- <properties> element, A-25
- PropertyPermission, 22-1

Q

- queries
 - about, 1-16, 1-27
 - EntityManager, 1-18
 - finder methods, 1-30
 - select methods, 1-32

- syntax, about, 1-16, 1-27
- syntax, EJB QL, 1-17, 1-27
- syntax, SQL, 1-17, 1-29
- syntax, TopLink, 1-28
- <query> element, 16-2, 16-5
- query attribute, A-23

R

- read-only, 1-25
 - BMP entity bean, 15-1
- read-only concurrency mode, A-21
- relational mappings
 - aggregate object, understanding, 7-12
 - many-to-many, understanding, 7-11
 - one-to-many, understanding, 7-11
- remote access, 29-14
- remote attribute, 29-14
- remote home interface
 - example, 11-7, 13-19, 16-2, 16-5
- remote interface
 - creating, 11-2, 11-4, 11-9, 13-2, 13-6, 13-19
 - EJB 2.1, overview, 1-4
 - EJB 3.0, overview, 1-2
 - example, 11-9, 13-20
- RemoteException, 11-7, 11-8, 11-9, 13-20
- remove method
 - @Remove annotation, 1-3
 - EJBHome interface, 1-5
- replication
 - inherited, 2-21
 - on end of request, 2-21
 - on shutdown, 2-21
- replication attribute, A-9
- <res-auth> element, 19-11, 19-13
- resource injection
 - about, 1-7
 - environment variables, 19-23
- resource-adapter attribute, A-15
- resource-check-interval attribute, A-8, A-27
- <resource-env-ref> element, 19-8, 19-15
- <resource-env-ref-mapping> element, A-6, A-10, A-14, A-25
- <resource-ref> element, 19-8
- <resource-ref-mapping> element, 19-11, 19-13, A-14, A-25
- resources
 - looking up, EJB 2.1, 19-26
 - looking up, EJB 3.0, 19-22
- <resource-provider> element, 23-6
- <resource-ref-mapping> element, A-6, A-10
- <res-ref-name> element, 19-11, 19-13
- <res-type> element, 19-11, 19-13
- <result-type-mapping> element, 16-5
- RMI
 - port, 29-2
- <role-link> element, 22-2, 22-3
- <role-name> element, 22-2, 22-3
- <run-as> element, 22-7
- runAs security identity, 22-7

RuntimeException, 11-7, 11-8
RuntimePermission, 22-1

S

schema manager
 table creation, automatic, 14-5
security, 22-1
 annotations, 22-12
 JAAS, 22-12
 JSR250, 22-12
 permissions, 22-1
 retrieving credentials using JAAS, 22-12
<security-identity> element, 22-7
<security-role> element, 22-2, 22-3
<security-role-ref> element, 22-3
<security-role-mapping> element, 22-8, A-18, A-25
<security-role-ref> element, 22-2, 22-3
select methods
 about, 1-32
sequencing
 configuration, EJB 2.1, 14-4
 configuration, EJB 3.0, 7-3
Serializable interface, 29-22
<service-ref-mapping> element, A-10
session bean
 configuration, EJB 2.1, 12-1
 configuration, EJB 3.0, 5-1
 context, 1-13, 11-10, 12-4
 deployment descriptor, A-5, A-6
 local home interface, 11-8
 remote home interface, 11-7
 removing, 1-9, 1-11, 1-35, 12-3
 stateful, 1-10
 stateless, 1-8
session beans
 transaction timeouts, 21-2
session context, 1-13
Session object, 19-10
SessionBean interface
 EJB, 11-2, 11-5, 12-4, 13-3, 13-7, 17-2
 ejbActivate method, 1-9, 1-11, 12-3
 ejbCreate method, 1-9, 1-11, 1-35, 12-3
 ejbPassivate method, 1-9, 1-11, 12-3
 ejbRemove method, 1-9, 1-11, 1-35, 12-3
 setSessionContext method, 12-4
<session-deployment> element, A-5, A-6, A-26
session-deployment
 call-timeout attribute, A-7
 copy-by-value attribute, A-7
 idletime attribute, A-8
 local-location attribute, A-7
 local-wrapper attribute, A-9
 location attribute, A-7
 max-instances attribute, A-7
 max-instances-threshold attribute, A-8
 max-tx-retries attribute, A-7
 memory-threshold attribute, A-8
 min-instances attribute, A-7
 name attribute, A-8

 passivate-count attribute, A-9
 persistence-filename attribute, A-8
 pool-cache-timeout attribute, A-6
 replication attribute, A-9
 resource-check-interval attribute, A-8
 timeout attribute, A-8
 transaction-timeout attribute, A-8
 tx-retry-wait attribute, A-8
 wrapper attribute, A-9
setEntityContext method, 13-20, 17-6
<set-mapping> element, A-27
setRollbackOnly, 21-5
setRollbackOnly method, 1-6
setSessionContext method, 1-13, 11-10, 12-4, 13-20, 17-6
<sfsb-config> element, 12-2, 12-3
SocketPermission, 22-1
SQL
 queries, about, 1-17, 1-29
SQRT, 16-8
stateful session bean
 implementing, EJB 3.0, 4-2
 lifecycle methods, EJB 2.1, 1-9, 1-11
 lifecycle methods, EJB 3.0, 1-9, 1-10
 overview, 1-10
stateful session beans
 state replication, 2-21
stateless session bean
 implementing, EJB 2.1, 11-1, 11-4, 13-1, 13-6, 17-1
 implementing, EJB 3.0, 4-1
 overview, 1-8
<subscription-durability> element, 17-5
subscription-name attribute, A-15, A-24
SubscriptionName config-property, A-15
system properties
 KeepWrapperCode, 31-6
 oracle.j2ee.rmi.loadBalance, 24-3
 WrapperCodeDir, 31-6
 KeepWrapperCode, 31-6

T

table attribute, A-13, A-22
Time, 16-8
TimeoutException, A-7, A-11
timeout attribute, 31-4, A-8, A-27
TimeoutExpiredException, A-7, A-26
timeouts
 bean instance pool, entity beans, 31-4
 bean instance pool, session beans, 31-4
 transactions, 21-1
timers, 2-23
 callback method implementation, 25-2
 cancel, 25-5
 creating, 25-2
 ejbTimeout method, 25-2
 executing within a transaction, 25-5
 NoSuchObjectLocalException, 25-5
 persistence, 25-5
 retrieving information, 25-5

- retrieving timer service, 25-2
- TimerService object, 25-2
- Timestamp, 16-8
- TopLink
 - ejb3-toplink-sessions.xml, about, 2-9
 - ejb3-toplink-sessions.xml, XSD, 2-9
 - queries, about, 1-28
 - toplink-ejb-jar.xml File, A-4
 - toplink-ejb-jar.xml, about, 2-8
 - toplink-ejb-jar.xml, XSD, 2-9
- TopLink migration tool, 3-5
- TopLink Workbench
 - toplink-ejb-jar.xml creation, 26-2
- toplink-ejb-jar.xml
 - about, 2-8
 - creating at migration time, 26-2
 - creating with TopLink Workbench, 26-2
 - XSD, 2-9
- toplink-ejb-jar.xml File, A-4
- transaction
 - commit, 1-6
 - context propagation, 1-6
 - retrieve status, 1-6
 - rollback, 1-6
- transaction isolation, 1-24
- transactions
 - about, 2-14
 - bean-managed, about, 2-15
 - beginning, about, 2-15
 - committing, about, 2-15
 - container-managed, about, 2-15
 - global, about, 2-16
 - interceptors, 2-12
 - isolation levels, 1-24
 - management, about, 2-15
 - rollback, 21-5
 - timeouts, configuring, 21-1
 - timeouts, global, 21-1
 - timeouts, message-driven beans, 21-2
 - timeouts, session beans, 21-2
 - two-phase commit, about, 2-16
 - two-phase commit, OJMS, 23-4
- transaction-timeout attribute, 21-2, 21-3, A-8, A-16, A-24
- TransactionTimeout config-property, A-16
- <transaction-type> element, 17-5
- troubleshooting, 27-2, 28-1
- two-phase commit, 2-16
- tx-retry-wait attribute, A-8, A-21, A-26
- type attribute, A-23, A-27

U

- <unchecked> element, 22-7
 - defined, 22-6
- unsetEntityContext method, 13-20
- update-changed-fields-only attribute, A-13, A-22
- <use-caller-identity> element, 22-8
- <user> element, A-27

V

- validating XML, 31-5
- validity-timeout attribute, A-13, A-22
- <value-mapping> element, A-27

W

- <web> element, 2-6
- Windows
 - shutdown, 18-4
- wrapper attribute, A-9, A-13, A-22, A-27
- wrapper code
 - debugging, 31-6

X

- XML validation, 31-5
- XSD
 - ejb3-toplink-sessions.xml, 2-9
 - ejb-jar.xml, EJB 2.1, 2-8
 - ejb-jar.xml, EJB 3.0, 2-8
 - orion-ejb-jar.xml, 2-8, A-1
 - toplink-ejb-jar.xml, 2-9, A-4