Oracle9*i*AS Containers for J2EE

Enterprise JavaBeans Developer's Guide

Release 2 (9.0.3)

August 2002

Part No. A97677-01

ORACLE®

Oracle9*i*AS Containers for J2EE Enterprise JavaBeans Developer's Guide, Release 2 (9.0.3)

Copyright © 2000, 2002 Oracle Corporation. All rights reserved.

Primary Author: Sheryl Maring

The Programs (which include both the software and documentation) contain proprietary information of Oracle Corporation; 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. Oracle Corporation does not warrant that this document is 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, without the express written permission of Oracle Corporation.

If the Programs are delivered to the U.S. Government or anyone licensing or using the programs on behalf of the U.S. Government, the following notice is applicable:

Restricted Rights Notice Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs, including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the Programs shall be subject to the restrictions 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 Oracle Corporation disclaims liability for any damages caused by such use of the Programs.

Oracle is a registered trademark, and Oracle9*i* is a trademark or registered trademark of Oracle Corporation. Other names may be trademarks of their respective owners.

Portions of this software are copyrighted by Data Direct Technologies, 1991-2001.

Contents

Send Us Your Comments xi

Preface xiii

1 EJB Overview

New Features of EJB 2.0 1-2
Local Interface Support 1-2
Home Interface Business Methods 1-3
Message-Driven Beans 1-4
Enterprise JavaBeans Query Language (EJB QL) 1-4
CMP Relationships 1-5
CORBA Support - RMI-over-IIOP 1-6
Invoking Enterprise JavaBeans 1-7
Implementing an EJB 1-9
Bean Implementation 1-10
Parameter Passing 1-10
Parameter Objects 1-10
Types of EJBs 1-11
Session Beans 1-11
Entity Beans 1-16
Message-Driven Beans 1-23
Difference Between Session and Entity Beans 1-25

2 An EJB Primer For OC4J

Develop EJBs 2-2
Create the Development Directory 2-2
Implement the EJB 2-4
Create the Deployment Descriptor 2-11
Archive the EJB Application 2-12
Prepare the EJB Application for Assembly 2-13
Modify the Application.XML File 2-13
Create the EAR File 2-14
Deploy the Enterprise Application to OC4J 2-15
Access the EJB 2-15

3 CMP Entity Beans

Entity Bean Overview 3-2
Creating Entity Beans 3-3
Home Interface 3-4
Component Interfaces 3-5
Entity Bean Class 3-6
Primary Key 3-9
Defining the Primary Key in a Class 3-10
Defining an Auto-Generated Primary Key 3-11
Persistence Fields 3-12
Default Mapping of Persistent Fields to the Database 3-14
Explicit Mapping of Persistent Fields to the Database 3-15
CMP Types 3-17
Simple Data Types 3-18
Serializable Classes 3-19
Other Entity Beans or Collections 3-19

4 Entity Relationship Mapping

Defining Entity-To-Entity Relationships 4-2
Choosing Cardinality and Direction 4-2
Defining Relationships 4-3
Mapping Relationship Fields to the Database 4-10

Default Mapping of Relationship Fields to the Database 4-11
Explicit Mapping of Relationship Fields to the Database 4-19

5 EJB Query Language

EJB QL Overview 5-2
Query Methods Overview 5-2
Finder Methods 5-2
Select Methods 5-3
Deployment Descriptor Semantics 5-4
Finder Method Example 5-5
Select Method Example 5-7

6 BMP Entity Beans

Creating BMP Entity Beans 6-2
Component and Home Interfaces 6-3
BMP Entity Bean Implementation 6-3
The ejbCreate Implementation 6-3
The ejbFindByPrimaryKey Implementation 6-7
Other Finder Methods 6-7
The ejbStore Implementation 6-8
The ejbLoad Implementation 6-9
The ejbPassivate Implementation 6-9
The ejbActivate Implementation 6-10
The ejbRemove Implementation 6-10
Modify XML Deployment Descriptors 6-11
Create Database Table and Columns for Entity Data 6-12

7 Message-Driven Beans

MDB Overview 7-2
Creating MDBs 7-3
Install And Configure The Resource Provider 7-4
Bean Class Implementation 7-8
Configure Deployment Descriptors 7-10
Deploy the Entity Bean 7-13

Accessing MDBs 7-13
Using Logical Names in the JMS JNDI Lookup 7-15

8 Advanced EJB Subjects

Accessing EJBs 8-2
Client Installation of OC4J.JAR 8-3
EJB Reference Information 8-3
Setting JNDI Properties 8-4
Configuring RMI or JMS Port for Standalone EJB Clients 8-6
Using the Initial Context Factory Classes 8-6
Accessing an EJB in a Remote Server 8-8
Packaging and Sharing Classes 8-8
Entity Bean Concurrency and Database Isolation Modes 8-10
Database Isolation Modes 8-10
Entity Bean Concurrency Modes 8-11
Exclusive Write Access to the Database 8-12
Effects of the Combination of Isolation and Concurrency Modes 8-13
Affects of Concurrency Modes on Clustering 8-13
Configuring Pool Sizes For Entity Beans 8-13
Techniques for Updating Persistence 8-14
Configuring Environment References 8-15
Environment variables 8-15
Environment References To Other Enterprise JavaBeans 8-16
Environment References To Resource Manager Connection Factory References 8-20
Configuring Security 8-26
Granting Permissions in Browser 8-27
Authenticating and Authorizing EJB Applications 8-27
Specifying Credentials in EJB Clients 8-36
Setting Performance Options 8-38
Performance Command-Line Options 8-38
Thread Pool Settings 8-39
Statement Caching 8-42
Task Manager Granularity 8-42
Using DNS for Load Balancing 8-43
Common Errors 8-44

NamingException Thrown 8-44
Deadlock Conditions 8-44
ClassCastException 8-44

9 EJB Clustering

EJB Clustering Overview 9-2
Stateless Session Bean Clustering 9-4
Stateful Session Bean Clustering 9-4
Entity Bean Clustering 9-4
Combination of HTTP and EJB Clustering 9-5
Enabling Clustering For EJBs 9-5
Configure Nodes With Multicast Address and Identifier 9-5
EJB Replication Configuration 9-7
Deploy EJB Application To All Nodes 9-8
Application Client Retrieval Of Clustered Nodes 9-8
Load Balancing Options 9-9

10 Active Components for Java

Future Needs of Business Applications 10-2
Architectures 10-3
Remote Procedure Call Model 10-3
Database Transactional Queueing Model 10-5
AC4J Solution 10-6
AC4J Architecture 10-8
Introduction to AC4J Components 10-8
Active EJBs 10-10
Interaction 10-11
Processes 10-12
Reactions 10-13
Data Tokens 10-16
Data Bus 10-17
Configuring Oracle Databases to Support AC4J 10-20
AC4J Data Bus XML Configuration 10-21
AC4J Example 10-23
Asynchronous Request to An Active EJB 10-25

Active EJB Processes the Client's Request 10-30
Asynchronous Response to the Requesting Active EJB 10-34
Asynchronous Response to the Client 10-34
Response from the Client 10-35
AC4J Active EJB Deployment 10-38
Administering AC4J 10-40
Administering Oracle Databases to Support AC4J 10-40
Description of the JEM PL/SQL package 10-40
Description of the createDatabusTpc Package Public Method 10-41
Description of the dropDatabusTpc Package Public Method 10-41
Description of the JEM Schema Objects 10-42

A EJB 1.1 CMP Entity Beans

Creating Entity Beans A-2
Home Interface A-3
Remote Interface A-4
Entity Bean Class A-4
Persistent Data A-7
Primary Key A-8
Deploying the Entity Bean A-10
Advanced CMP Entity Beans A-11
EJB 1.1 Advanced Finder Methods A-11
EJB 1.1 Object-Relational Mapping of Persistent Fields A-13

B OC4J-Specific DTD Reference

OC4J-Specific Deployment Descriptor for EJBs B-3
Enterprise Beans Section B-3
Assembly Descriptor Section B-20
Element Description B-21

C Third Party Licenses

Apache HTTP Server C-2
The Apache Software License C-2
Apache JServ C-4

Apache JServ Public License C-4

Index

Send Us Your Comments

Oracle9*i*AS Containers for J2EE Enterprise JavaBeans Developer's Guide, Release 2 (9.0.3)

Part No. A97677-01

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

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

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

- Electronic mail — jpgreader_us@oracle.com
- FAX - 650-506-7225. Attn: Java Platform Group, Information Development Manager
- Postal service:
  Oracle Corporation
  Information Development Manager
  500 Oracle Parkway, Mailstop 4op978
  Redwood Shores, CA 94065
  USA

Please indicate if you would like a reply.

If you have problems with the software, please contact your local Oracle World Wide Support Center.

Preface

This guide gets you started building Enterprise JavaBeans for OC4J. It includes code examples to help you develop your application.

Who Should Read This Guide?

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. To use this guide effectively, you must have a working knowledge of J2EE.

Prerequisite Reading

Before consulting this Guide, you should read the following:

- Any J2EE book that enables you to understand the basics of J2EE programming.
- The *Oracle9iAS Containers for J2EE User's Guide*. This guide helps you to understand the minimum requirements for a J2EE application in the OC4J environment.
- The Sun Microsystems EJB 2.0 specification as a supplement to this guide. This guide assumes that you already have a base understanding of the EJB 2.0 specification details.

Suggested Reading

Books

- *Professional Java Server Programming, J2EE Edition*, Wrox Press Ltd, 2000.
- *Mastering Enterprise JavaBeans and the Java2 Platform Enterprise Edition*, by Ed Roman. Wily Computer Publishing, 1999.
- *Designing Enterprise Applications with the Java2 Platform, Enterprise Edition*, Addison-Wesley, 2000.
- *Core Java* by Cornell & Horstmann, second edition, Volume II (Prentice-Hall, 1997) demonstrates several Java concepts relevant to EJBs.
- The *Developer's Guide to Understanding Enterprise JavaBeans,* an overview of EJBs, is available at `http://www.Nova-Labs.com.`

Online Sources

There are many useful online sources of information about Java. For example, you can view or download guides and tutorials from the Sun Microsystems home page on the Web:

`http://www.sun.com`

The current 2.0 EJB specification is available at:

`http://java.sun.com/products/ejb/docs.html`

Another popular Java Web site is:

`http://www.gamelan.com`

For Java API documentation, see:

`http://www.javasoft.com`

How This Guide Is Organized

This guide consists of the following:

Chapter 1, "EJB Overview", presents a brief overview of EJBs.

Chapter 2, "An EJB Primer For OC4J", discusses a stateless session bean development for the OC4J server.

Chapter 3, "CMP Entity Beans", discusses a CMP entity bean and advanced issues connected with CMP entity beans.

Chapter 4, "Entity Relationship Mapping", discusses container-managed relationships (CMR) within the entity bean for OC4J.

Chapter 5, "EJB Query Language", provides an overview and examples of setting up query methods that use EJB QL.

Chapter 6, "BMP Entity Beans", discusses a BMP entity bean.

Chapter 7, "Message-Driven Beans", discusses an MDB entity bean.

Chapter 8, "Advanced EJB Subjects", discusses advanced issues for EJBs.

Chapter 9, "EJB Clustering", discusses how to cluster EJBs across OC4J nodes.

Chapter 10, "Active Components for Java", introduces a new methodology to merge the advantages of both asynchronous and request/response communication.

Appendix A, "EJB 1.1 CMP Entity Beans" contains the EJB 1.1 CMP entity bean methodology.

Appendix B, "OC4J-Specific DTD Reference" describes the OC4J-specific deployment descriptor.

Documentation Accessibility

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

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

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

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

Notational Conventions

This guide follows these conventions:

Italic	Italic font denotes terms being defined for the first time, words being emphasized, error messages, and book titles.
`Courier`	Courier font denotes Java program names, file names, path names, and Internet addresses.

Java code examples follow these conventions:

`{ }`	Braces enclose a block of statements.
`//`	A double slash begins a single-line comment, which extends to the end of a line.
`/* */`	A slash-asterisk and an asterisk-slash delimit a multi-line comment, which can span multiple lines.
`...`	An ellipsis shows that statements or clauses irrelevant to the discussion were left out.
`lower case`	Lower case is used for keywords and for one-word names of variables, methods, and packages.
`UPPER CASE`	Upper case is used for names of constants (static final variables) and for names of supplied classes that map to built-in SQL datatypes.
`Mixed Case`	Mixed case is used for names of classes and interfaces and for multi-word names of variables, methods, and packages. The names of classes and interfaces begin with an upper-case letter. In all multi-word names, the second and succeeding words also begin with an upper-case letter.

1 EJB Overview

This chapter discusses EJB concepts that are specified fully in the J2EE specification. The remainder of the chapters in this book show only the tasks necessary to develop your EJBs.

For more details and examples of the concepts presented in this chapter, refer to books written by Sun Microsystems that discuss EJBs and J2EE Blueprint Architecture recommendations.

This chapter includes the following topics:

- New Features of EJB 2.0
- Invoking Enterprise JavaBeans
- Implementing an EJB
- Types of EJBs
- Difference Between Session and Entity Beans

New Features of EJB 2.0

The following sections describe the new features to EJB 2.0:

- Local Interface Support
- Home Interface Business Methods
- Message-Driven Beans
- Enterprise JavaBeans Query Language (EJB QL)
- CMP Relationships
- CORBA Support - RMI-over-IIOP

Local Interface Support

Oracle9*i*AS provides complete support for local interfaces.

A client may access a session or an entity bean only through the methods defined in the bean's interfaces which define the client's view of a bean. All other aspects of the bean - method implementations, deployment descriptor settings, abstract schemas, database access calls - are hidden from the client providing modularity and encapsulation. Well designed interfaces simplify the development and maintenance of J2EE applications by shielding clients from any complexities in the business logic and also allowing the EJBs to change internally without affecting the clients. EJBs support two types of client access - remote or local.

Remote Access

A remote client of an enterprise bean has the following traits:

1. It may run on a different machine and a different Java Virtual Machine (JVM) than the enterprise bean it accesses.
2. It can be a web component, a J2EE application client, or another enterprise bean.
3. To a remote client, the location of the enterprise bean is transparent. To create an enterprise bean with remote access, you must code a remote interface and a home interface. The remote interface defines the business methods that are specific to the bean.

Local Access

A local client has these characteristics:

1. It must run in the same JVM as the enterprise bean it accesses.
2. It may be a web component or another enterprise bean.
3. To the local client, the location of the enterprise bean it accesses is not transparent.
4. It is often an entity bean that has a container-managed relationship with another entity bean. To build an enterprise bean that allows local access, you must code a local interface and a local home interface. The local interface defines the bean's business methods and the local home interface defines its life-cycle and finder methods.

Local Interfaces and Container-Managed Relationships

If an entity bean is the target of a container-managed relationship, then it must have local interfaces. Further, if the relationship between the EJBs is bi-directional, both beans must have local interfaces. Moreover, since they require local access, entity beans that participate in a container-managed relationship must reside in the same EJB container. The primary benefit of this locality is increased performance - local calls are usually faster than remote calls.

Local Compared to Remote Access

The decision on whether to allow local or remote access depends on the following factors:

1. Container-Managed Relationships - If an entity bean is the target of a container-managed relationship, it must use local access.
2. Tight or Loose Coupling of Related Beans - tightly coupled beans depend on one another. For example, a completed sales order must have one or more line items, which cannot exist without the order to which they belong. The `OrderEJB` and `LineItemEJB` beans that model this relationship are tightly coupled. Tightly coupled beans are good candidates for local access. Since they fit together as a logical unit, they probably call each other often and would benefit from the increased performance that is possible with local access.

Home Interface Business Methods

Home interface business methods are used for public usage of methods that do not use entity bean persistent data. If you want to supply methods that perform duties for you that are not associated with any specific bean, a home interface business method allows you to publicize this method.

Message-Driven Beans

You can implement EJB 2.0 message-driven beans with Oracle JMS. A full example is provided in Chapter 7, "Message-Driven Beans".

Enterprise JavaBeans Query Language (EJB QL)

EJB QL defines the queries for the finder and select methods of an entity bean with container-managed persistence. A subset of SQL92, EJB QL has 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 datastore. 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`). The EJB QL query determines the query that is executed by the EJB container when the finder method is invoked.

Oracle9*i*AS 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 Oracle9*i*AS, 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, Oracle9*i*AS makes several optimizations such as the use of bulk SQL, batched statement dispatch, etc. to make database access efficient.
- Support for Oracle and Non-Oracle Databases: Further, Oracle9*i*AS provides the ability to execute EJB QL against any database - Oracle, MS SQL-Server, IBM DB/2, Informix, and Sybase.
- CMP with Relationships: Oracle9*i*AS 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.

CMP Relationships

The EJB 2.0 specification enables the specification of relationships between entity beans. An entity bean can be defined so as to have a relationship with other entity beans. For example, in a project management application the `ProjectEJB` and `TaskEJB` beans would be related because a project is made up of a set of tasks. You implement relationships differently for entity beans with bean-managed-persistence than those entity beans that utilize container-managed-persistence. With bean-managed persistence, the code that you write implements the relationships. With container-managed persistence, the EJB container takes care of the relationships for you. For this reason, relationships in entity beans with container-managed persistence are often referred to as container-managed relationships.

- Relationship Fields - A relationship field in an EJB identifies a related bean. A relationship field is virtual and is defined in the enterprise bean class with access methods. Unlike a persistent field, a relationship field does not represent the bean's state.
- Multiplicity in Container-Managed Relationships - There are four types of multiplicities all of which are supported by Oracle9*i*AS:
  - One-to-One - Each entity bean instance is related to a single instance of another entity bean.
  - One-to-Many - An entity bean instance is related to multiple instances of the other entity bean.
  - Many-to-One - Multiple instances of an entity bean may be related to a single instance of the other entity bean. This multiplicity is the opposite of one-to-many.
  - Many-to-Many - The entity bean instances may be related to multiple instances of each other.
- Direction in Container-Managed Relationships - 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. Oracle9*i*AS supports both unidirectional and bi-directional relationships between EJBs.

- EJBQL and CMP With Relationships - EJB QL queries often navigate across relationships. The direction of a relationship determines whether a query can navigate from one bean to another. With Oracle9*i*AS, EJBQL queries can traverse CMP Relationships with any type of multiplicity and with both unidirectional and bi-directional relationships.

Oracle9*i*AS Object-Relational Mapping

Oracle9*i*AS furnishes, out of the box, its own persistence manager for entity beans, which supplies both simple (1:1) mapping and complex relationship (1:n, m:n) mapping. Oracle9*i*AS provides complete support for the EJB 2.0 O-R mapping specification.

Third Party O-R Mappings - TopLink Integration

Oracle9*i*AS integrates leading third party O-R mapping solutions including TopLink for Java, with the EJB container. TopLink provides developers with the flexibility to map objects and Enterprise Java Beans to a relational database schema with minimal impact. TopLink for Java provides advanced mapping capabilities such as bean/object identity mapping, type and value transformation, relationship mapping (1:1, 1:n and m:n), object caching and locking, batch writing, and advanced and dynamic query capabilities. TopLink offers a GUI mapping tool - the TopLink Mapping Workbench - which simplifies the process of mapping J2EE components to database objects. TopLink provides EJB 2.0 support, automatic or developer-configured bi-directional relationship maintenance, automatic or developer-configured cache synchronization session management via XML, and optimistic read locking. Oracle9*i*AS is also integrated with other leading O-R mapping solutions in the market.

CORBA Support - RMI-over-IIOP

RMI over IIOP is part of the J2EE 1.3 Specification and provides two important benefits:

- RMI over IIOP provides the ability to write CORBA applications for the Java platform without learning CORBA Interface Definition Language (IDL).
- IIOP eases legacy application and platform integration by allowing applications written in C++, Smalltalk, and other CORBA supported languages to communicate with J2EE components.

Oracle9*i*AS supports RMI-over-IIOP providing the following important facilities:

- Automatic IDL Stub and Helper Class Generation - To work with CORBA applications in other languages, IDL, CORBA stubs and skeletons can be generated:
  1. Automatically by Oracle9*i*AS when the J2EE Application is deployed to it.
  2. IDL can also be generated from J2EE interfaces using the rmic compiler with the `-idl` option. Further, developers can use the `rmic` compiler with the `-iiop` option to generate IIOP stub and tie classes, rather than Java Remote Messaging Protocol (JRMP) stub and skeleton classes.
- Objects-By-Value - The Oracle9*i*AS RMI-IIOP implementation provides flexibility by allowing developers to pass any serializable Java object (Objects By Value) between application components.
- POA Support - The Portable Object Adapter (POA) is designed to provide an object adapter that can be used with multiple ORB implementations with a minimum of rewriting needed to deal with different vendors' implementations. The POA is also intended to allow persistent objects -- at least, from the client's perspective. That is, as far as the client is concerned, these objects are always alive, and maintain data values stored in them, even though physically, the server may have been restarted many times, or the implementation may be provided by many different object implementations. Oracle9*i*AS provides complete POA support.
- Interoperating with Other ORBs - The Oracle9*i*AS RMI-IIOP implementation will interoperate with other ORBs that support the CORBA 2.3 specification. It will not interoperate with older ORBs, because these are unable to handle the IIOP encodings for Objects By Value. This support is needed to send RMI value classes (including strings) over IIOP. Oracle9*i*AS also provides complete support for the Interoperable Naming, Security, and Transactions elements in the J2EE 1.3 specification allowing developers to build J2EE applications and interoperate them with J2EE applications on other Application Servers and with legacy systems through CORBA.

See the RMI/Interoperability chapter in the *Oracle9iAS Containers for J2EE Services Guide* for more information.

Invoking Enterprise JavaBeans

Enterprise JavaBeans (EJBs) can be one of three types: session beans, entity beans, or message-driven beans.

- Session beans can be stateful or stateless and are used for business logic functionality.

- Stateless session beans are used for business services. They do not retain client state across calls.
- Stateful session beans do maintain state across client calls. Thus, these beans manage business functions for a specific client for the life of that client.

- Entity beans are normally used for managing persistent data.
- Message-driven beans are used for receiving messages from a JMS queue or topic.

An EJB has two client interfaces:

- Remote interface—The remote interface specifies the business methods that the clients of the object can invoke.
- Home interface—The home interface defines EJB life cycle methods, such as a method to create and retrieve a reference to the bean object.

The client uses both of these interfaces when invoking a method on a bean.

Figure 1–1 Events In A Stateless Session Bean

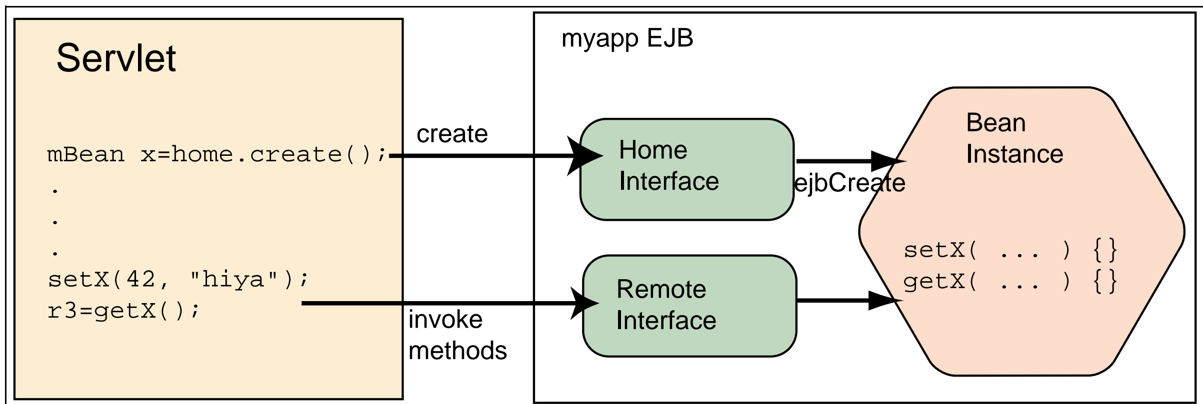


Figure 1–1 demonstrates a stateless session bean and corresponds to the following steps:

1. The client, which can be a standalone Java client, servlet, JSP, or an applet, 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 remote interface of the bean.
3. The client invokes a method defined in the remote 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 remote interface. Some beans, such as stateless session beans, cannot call the `remove` method. In this case, the container removes the bean.

Implementing an EJB

You must create the following four major components to develop an EJB:

- the *home interface*
- the *remote interface*
- the *implementation* of the bean
- a *deployment descriptor* for each EJB

Component	Description
The home interface	Specifies the interface to an object that the container itself implements: the *home object*. The home interface contains the life cycle methods, such as the `create()` methods that specify how a bean is created.
The remote interface	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.
The bean implementation	Contains the Java code that implements the methods defined in the home interface (life cycle methods), remote interface (business methods), and the required container methods (container callback functions).
The deployment descriptor	Specifies attributes of the bean for deployment. These designate configuration specifics, such as environment, interface names, transactional support, type of EJB, and persistence information.

Bean Implementation

Your bean implements the methods within either the SessionBean, EntityBean, or MessageDrivenBean interface. The implementation contains logic for lifecycle methods defined in the home interface, business methods defined in the remote interface, and container callback functions defined in the SessionBean, EntityBean, or MessageDrivenBean interface.

Parameter Passing

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.

Parameter Objects

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.

Types of EJBs

There are three types of EJBs: ***session beans***, ***entity beans***, and ***message-driven beans***.

- Session Beans
- Entity Beans
- Message-Driven Beans

Session Beans

A session bean implements one or more business tasks. A session bean might contain methods that query and update data in a relational table. Session beans are often used to implement services. For example, an application developer might implement one or several session beans that retrieve and update inventory data in a database.

Session beans are transient because they do not survive a server crash or a network failure. If, after a crash, you instantiate a bean that had previously existed, the state of the previous instance is not restored. State can be restored only to entity beans.

A session bean implements the `javax.ejb.SessionBean` interface, which has the following definition:

```
public interface javax.ejb.SessionBean extends javax.ejb.EnterpriseBean {
   public abstract void ejbActivate();
   public abstract void ejbPassivate();
```

```
    public abstract void ejbRemove();
    public abstract void setSessionContext(SessionContext ctx);
}
```

At a minimum, an EJB must implement the following methods, as specified in the `javax.ejb.SessionBean` interface:

`ejbCreate()`	The container invokes this method right before it creates the bean. Stateless session beans must do nothing in this method. Stateful session beans can initiate state in this method.
`ejbActivate()`	The container invokes this method right after it reactivates the bean.
`ejbPassivate()`	The container invokes this method right before it passivates the bean.
`ejbRemove()`	A container invokes this method before it ends the life of the session object. This method performs any required clean-up—for example, closing external resources such as file handles.
`setSessionContext (SessionContext ctx)`	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.

Using setSessionContext

You use this method to obtain a reference to the context of the bean. Session beans have session contexts that the container maintains and makes available to the beans. 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. The following example shows the bean saving the session context in the **`sessctx`** variable.

```
import javax.ejb.*;
import oracle.oas.ejb.*;

public class myBean implements SessionBean {
   SessionContext sessctx;

   void setSessionContext(SessionContext ctx) {
      sessctx = ctx;   // session context is stored in
                       // instance variable
   }
   // other methods in the bean
}
```

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

```
public interface SessionContext extends javax.ejb.EJBContext {
    public abstract EJBObject getEJBObject();
}
```

And 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 session context when it wants to perform the operations listed in Table 1–1.

Table 1–1 SessionContext Operations

Method	**Description**
`getEnvironment()`	Get the values of properties for the bean.
`getUserTransaction()`	Get a transaction context, which allows you to demarcate transactions programmatically. This is valid only for beans that have been designated transactional.
`setRollbackOnly()`	Set the current transaction so that it cannot be committed.
`getRollbackOnly()`	Check whether the current transaction has been marked for rollback only.
`getEJBHome()`	Retrieve the object reference to the corresponding `EJBHome` (home interface) of the bean.

There are two types of session beans:

- Stateless Session Beans—Stateless session beans do not share state or identity between method invocations. They are useful mainly in middle-tier application servers that provide a pool of beans to process frequent and brief requests.
- Stateful Session Beans—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.

Stateless Session Beans

A stateless session bean does not maintain any state for the client. It 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.

The EJB container 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. There is implemented only a single `create/ejbCreate` with no parameters—containing no initialization for the bean within these methods. There is no need to implement any actions within the `remove/ejbRemove`, `ejbPassivate`, `ejbActivate`, and `setSessionContext` methods. In addition, there is no need for the intended use

for these methods in a stateless session bean. Instead, these methods are used mostly for EJBs with state—for stateful session beans and entity beans. Thus, these methods should be empty or extremely simple.

Implementation	Methods
Home Interface	Extends `javax.ejb.EJBHome` and requires a single `create()` factory method, with no arguments, and a single `remove()` method.
Remote Interface	Extends `javax.ejb.EJBObject` and defines the business logic methods, which are implemented in the bean implementation.
Bean implementation	Implements `SessionBean`. This class must be declared as public, contain a public, empty, default constructor, no `finalize`() method, and implements the methods defined in the remote interface. Must contain a single `ejbCreate` method, with no arguments, to match the `create`() method in the home interface. Contains empty implementations for the container service methods, such as `ejbRemove`, and so on.

Stateful Session Beans

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. The state of this type of bean is maintained across several calls through serialization of its state, called passivation. This is why the state that you passivate must be serializable. However, this information does not survive system crashes.

To maintain state for several stateful beans in a pool, it serializes the conversational state of the least recently used stateful bean to a secondary storage. When the bean instance is requested again by its client, the state is activated to a bean within the pool. Thus, all resources are used performantly, and the state is not lost.

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.

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.

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

Entity Beans

An entity bean is a complex business entity. 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 necessary tasks.

An entity bean is a remote object that manages persistent data, performs complex business logic, potentially uses several dependent Java objects, and can be uniquely identified by a primary key. Entity beans are normally coarse-grained persistent objects, because they utilize persistent data stored within several fine-grained persistent Java objects.

Entity beans are persistent because they do survive a server crash or a network failure. When an entity bean is re-instantiated, the state of previous instances is automatically restored.

Uniquely Identified by a Primary Key

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.

Managing Persistent Data

The persistence for entity bean data is provided both for saving state when the bean is passivated and for recovering the state when a failover has occurred. Entity beans are able to survive because the data is stored persistently by the container in some form of data storage system, such as a database. Entity beans persist business data using one of the two following methods:

- Automatically by the container using a container-managed persistent (CMP) entity bean.
- Programmatically through methods implemented in a bean-managed persistent (BMP) entity bean. These methods use JDBC or SQLJ to manage persistence.

An entity bean manages its data persistence through callback methods, which are defined in the `javax.ejb.EntityBean` interface. When you implement the `EntityBean` interface in your bean class, you develop each of the callback functions as designated by the type of persistence that you choose: bean-managed persistence or container-managed persistence. The container invokes the callback functions at designated times.

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

```
public interface javax.ejb.EntityBean extends javax.ejb.EnterpriseBean {
   public abstract void ejbActivate();
   public abstract void ejbLoad();
   public abstract void ejbPassivate();
   public abstract void ejbRemove();
   public abstract void ejbStore();
   public abstract void setEntityContext(EntityContext ctx);
   public abstract voic unsetEntityContext();
}
```

The container expects these methods to have the following functionality:

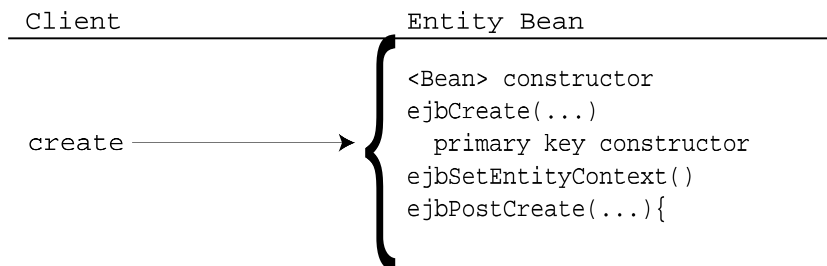
- `ejbCreate` — You must implement an `ejbCreate` method corresponding to each `create` method declared in the home interface. When the client invokes the `create` method, the container first invokes the constructor to instantiate the object, then it invokes the corresponding `ejbCreate` method. The `ejbCreate` method performs the following:
  - creates any persistent storage for its data, such as database rows
  - intializes a unique primary key and returns it
- `ejbPostCreate` — The container invokes this method after the environment is set. For each `ejbCreate` method, an `ejbPostCreate` method must exist with the same arguments. This method can be used to initialize parameters within or from the entity context.
- `ejbRemove` — The container invokes this method before it ends the life of the session object. This method can perform any required clean-up, for example closing external resources such as file handles.
- `ejbStore` — The container invokes this method right before a transaction commits. It saves the persistent data to an outside resource, such as a database.
- `ejbLoad` — The container invokes this method when the data should be reinitialized from the database. This normally occurs after activation of an entity bean.
- `setEntityContext` — Associates the bean instance with 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.

  You can also allocate any resources that will exist for the lifetime of the bean within this method. You should release these resources in `unsetEntityContext`.

- `unsetEntityContext` Unset the associated entity context and release any resources allocated in `setEntityContext`.
- `ejbActivate` The container calls this method directly before it activates an object that was previously passivated. Perform any necessary reaquisition of resources in this method.
- `ejbPassivate` The container calls this method before it passivates the object. Release any resources that can be easily re-created in `ejbActivate`, and save storage space. Normally, you want to free resources that cannot be passivated, such as sockets or database connections. Retrieve these resources in the `ejbActivate` method.

Using ejbCreate and ejbPostCreate An entity bean is similar to a session bean because certain callback methods, such as `ejbCreate`, are invoked at specified times. Entity beans use callback functions for managing its persistent data, primary key, and context information. The following diagram shows what methods are called when an entity bean is created.

Figure 1–2 Creating the Entity Bean



Using setEntityContext 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.

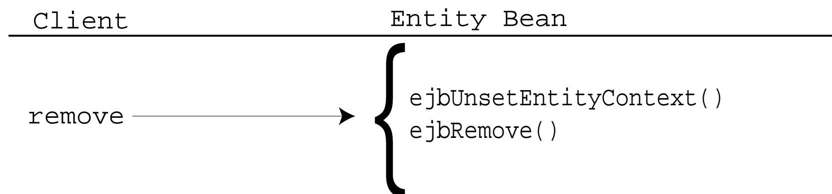
Note: You can also use the `setEntityContext` and `unsetEntityContext` methods to allocate and destroy any resources that will exist for the lifetime of the instance.

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.

```
public void setEntityContext(EntityContext ctx) { this.ctx = ctx; }
```

Using ejbRemove When the client invokes the `remove` method, the container invokes the methods shown in Figure 1–3.

Figure 1–3 Removing the Entity Bean



Using ejbStore and ejbLoad In addition, the `ejbStore` and `ejbLoad` methods are called for managing your persistent data. These are the most important callback methods—for bean-managed persistent beans. Container-managed persistent beans can leave these methods empty, because the persistence is managed by the container.

- 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 `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.

Container-Managed Persistence

You can choose to have the container manage your persistent data for the bean. You do not have to implement some of the callback methods to manage persistence for your bean's data, 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. In addition, you do not have to provide management for the primary key: the container provides this key for the bean.

- Callback methods—The container still invokes the callback methods, so you can add logic for other purposes. At the least, you must provide an empty implementation for all callback methods.
- Primary key—The primary key fields in a CMP bean must be declared as container-managed persistent fields in the deployment descriptor. All fields within the primary key are restricted to be either primitive, serializable, and types that can be mapped to SQL types.

The following table details the implementation requirements for the callback functions of the bean class:

Callback Method	Functionality Required
`ejbCreate`	You must initialize all container-managed persistent fields, including the primary key.
`ejbPostCreate`	You have the option to provide any additional initialization, which can involve the entity context.
`ejbRemove`	No functionality for removing the persistent data from the outside resource is required. You must at least provide an empty implementation for the callback, which means that you can add logic for performing any cleanup functionality you require.
`ejbFindByPrimaryKey`	No functionality is required for returning the primary key to the container. The container manages the primary key—after it is initialized by the `ejbCreate` method. You still must provide an empty implementation for this method.
`ejbStore`	No functionaltiy is required for saving persistent data within this method. The persistent manager saves all persistent data to the database for you. However, you must provide at least an empty implementation.

Callback Method	Functionality Required
`ejbLoad`	No functionality is required for restoring persistent data within this method. The persistence manager restores all persistent data for you. However, you must provide at least an empty implementation.
`setEntityContext`	Associates the bean instance with 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. You can also allocate any resources that will exist for the lifetime of the bean within this method. You should release these resources in `unsetEntityContext`.
`unsetEntityContext`	Unset the associated entity context and release any resources allocated in `setEntityContext`.

Differences Between Bean and Container-Managed Persistence

There are two methods for managing the persistent data within an entity bean: bean-managed (BMP) and container-managed persistence (CMP). The main difference between BMP and CMP beans is defined by who manages the persistence of the entity bean's data. With CMP beans, the container manages the persistence—the bean deployment descriptor specifies how to map the data and where the data is stored. With BMP beans, the logic for saving the data and where it is saved is programmed within designated methods. These methods are invoked by the container at the appropriate moments.

In practical terms, the following table provides a definition for both types, and a summary of the programmatic and declarative differences between them:

	Bean-Managed Persistence	Container-Managed Persistence
Persistence management	You are required to implement the persistence management within the ejbStore, ejbLoad, ejbCreate, and ejbRemove EntityBean methods. These methods must contain logic for saving and restoring the persistent data. For example, the ejbStore method must have logic in it to store the entity bean's data to the appropriate database. If it does not, the data can be lost.	The management of the persistent data is done for you. That is, the container invokes a persistence manager on behalf of your bean. You use ejbStore and ejbLoad for preparing the data before the commit or for manipulating the data after it is refreshed from the database. The container always invokes the ejbStore method right before the commit. In addition, it always invokes the ejbLoad method right after reinstating CMP data from the database.
Finder methods allowed	The findByPrimaryKey method and other finder methods are allowed.	The findByPrimaryKey method and other finder methods clause are allowed.
Defining 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. See the next section for a description of a persistence manager.

Message-Driven Beans

Message-Driven Beans (MDB) provide an easier method to implement asychronous communication than using straight JMS. MDBs were created to receive asynchronous JMS messages. The container handles much of the setup required for JMS queues and topics. It sends all messages to the interested MDB.

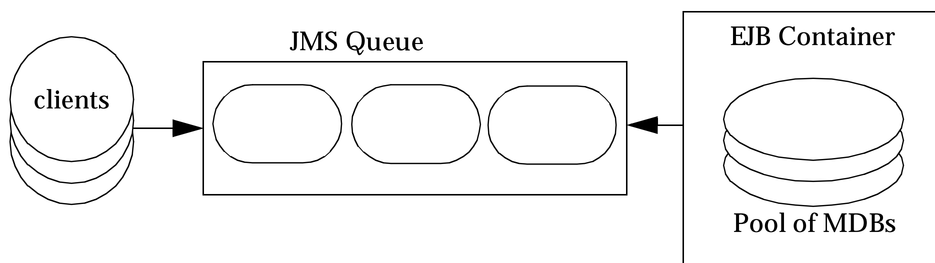
Previously, EJBs could not send or receive JMS messages. It took creating MDBs for an EJB-type object to receive JMS messages. This provides all of the asynchronous and publish/subscribe abilities to an enterprise object that is able to be synchronous with other Java objects.

The purpose of an MDB is to exist within a pool and to receive and process incoming messages from a JMS queue. 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 1–4 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 1–4 Message Driven Beans



MDBs implement the `javax.ejb.MessageDrivenBean` interface, which also inherits the `javax.jms.MessageListener` methods. Within these interfaces, the following methods must be implemented:

Method	Description
`onMessage(msg)`	The container dequeues a message from the JMS queue associated with this MDB and gives it to this instance by invoking this method. This method must have an implementation for handling the message appropriately.
`setMessageDrivenContext(ctx)`	After the bean is created, the `setMessageDrivenContext` method is invoked. This method is similar to the EJB `setSessionContext` and `setEntityContext` methods.
`ejbCreate()`	This method is used just like the stateless session bean `ejbCreate` method. No initialization should be done in this method. However, any resources that you allocate within this method will exist for this object.

Method	Description
`ejbRemove()`	Delete any resources allocated within the `ejbCreate` method.

The container handles JMS message retrieval and acknowledgment. Your MDB does not have to worry about JMS specifics. The MDB is associated with an existing JMS queue. Once associated, the container handles dequeuing messages and sending acknowledgments. The container communicates the JMS message through the `onMessage` method.

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. The following table 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.

	Entity Bean	Session Bean
Remote interface	Extends `javax.ejb.EJBObject`	Extends `javax.ejb.EJBObject`
Home interface	Extends `javax.ejb.EJBHome`	Extends `javax.ejb.EJBHome`
Bean class	Extends `javax.ejb.EntityBean`	Extends `javax.ejb.SessionBean`
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.

2

An EJB Primer For OC4J

After you have installed Oracle9*i*AS Containers for J2EE (OC4J) and configured the base server and default Web site, you can start developing J2EE applications. This chapter assumes that you have a working familiarity with simple J2EE concepts and a basic understanding for EJB development.

This chapter demonstrates simple EJB development with a basic OC4J-specific configuration and deployment. Download the stateless session bean example (`stateless.jar`) from the OC4J sample code page at `http://otn.oracle.com/sample_code/tech/java/oc4j/htdocs/oc4jsamplecode/oc4j-demo-ejb.html` on the OTN Web site.

To develop and deploy EJB applications with OC4J, do the following:

- Develop EJBs—Developing and testing an EJB module within the standard J2EE specification.
- Prepare the EJB Application for Assembly—Before deploying, you must modify an XML file that acts as a manifest file for the enterprise application.
- Deploy the Enterprise Application to OC4J—Archive the enterprise Java application into an Enterprise ARchive (EAR) file and deploy it to OC4J.
- Access the EJB—Develop the client to access the bean through the remote or local interface.

Develop EJBs

You develop EJB components for the OC4J environment in the same way as in any other standard J2EE environment. Here are the steps to develop EJBs:

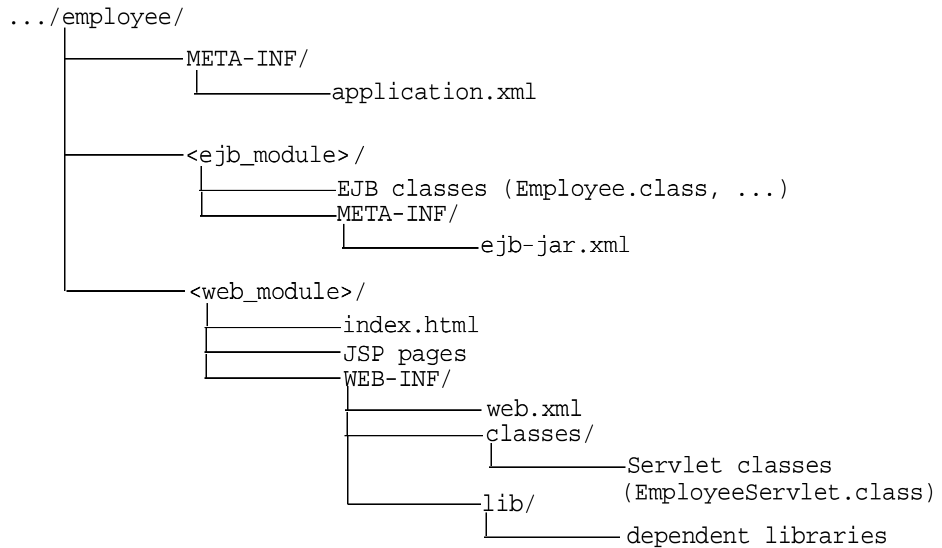
1. Create the Development Directory—Create a development directory for the enterprise application (as Figure 2–1 shows).
2. Implement the EJB—Develop your EJB with its home interfaces, component interfaces, and bean implementation.
3. Create the Deployment Descriptor—Create the standard J2EE EJB deployment descriptor for all beans in your EJB application.
4. Archive the EJB Application—Archive your EJB files into a JAR file.

Create the Development Directory

Although you can develop your application in any manner, we encourage you to use consistent naming for locating 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.

Our employee example was developed using the directory structure mentioned in the *Oracle9iAS Containers for J2EE User's Guide*. Notice in Figure 2–1 that the EJB and Web modules exist under the `employee` application parent directory and are developed separately in their own directory.

Figure 2–1 Employee 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.Employee.class`' is expected to be located in "`...employee/ejb_module/myapp/Employee.class`".

Implement the EJB

When you implement an EJB, create the following:

1. The home interfaces for the bean. The home interface defines the `create` method for your bean. If the bean is an entity bean, it also defines the finder method(s) for that bean.
   a. The remote home interface extends `javax.ejb.EJBHome`.
   b. The local home interface extends `javax.ejb.EJBLocalHome`.
2. The component interfaces for the bean.
   a. The remote interface declares the methods that a client can invoke remotely. It extends `javax.ejb.EJBObject`.
   b. The local interface declares the methods that a collocated bean can invoke locally. It extends `javax.ejb.EJBLocalObject`.
3. The bean implementation includes the following:
   a. The implementation of the business methods that are declared in the component interfaces.
   b. The container callback methods that are inherited from either the `javax.ejb.SessionBean` or `javax.ejb.EntityBean` interfaces.
   c. The `ejbCreate` and `ejbPostCreate` methods with parameters matching those of the `create` method as defined in the home interfaces.

Creating the Home Interfaces

The home interfaces (remote and local) are used to create the bean instance; thus, they define the `create` method for your bean. Each type of EJB can define the `create` method in the following ways:

EJB Type	Create Parameters
Stateless Session Bean	Can have only a single `create` method, with no parameters.
Stateful Session Bean	Can have one or more `create` methods, each with its own defined parameters.
Entity Bean	Can have zero or more `create` methods, each with its own defined parameters. All entity beans must define one or more finder methods, where at least one is a `findByPrimaryKey` method.

For each `create` method, a corresponding `ejbCreate` method is defined in the bean implementation.

Remote Invocation Any 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. You can use the parameter arguments to initialize the state of the new EJB object.

1. The remote home interface must extend the `javax.ejb.EJBHome` interface.
2. All `create` methods must throw the following exceptions:
   - `javax.ejb.CreateException`
   - either `java.rmi.RemoteException` or `javax.ejb.EJBException`

Example 2–1 Remote Home Interface for Session Bean

The following code sample illustrates a remote home interface for a session bean called `EmployeeHome`.

```
package employee;

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

public interface EmployeeHome extends EJBHome
{
  public Employee create()
    throws CreateException, RemoteException;
}
```

Local Invocation 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. You can use the parameter arguments to initialize the state of the new EJB object.

1. The local home interface must extend the `javax.ejb.EJBLocalHome` interface.
2. All `create` methods must throw the following exceptions:
   - `javax.ejb.CreateException`
   - `javax.ejb.EJBException`

Example 2–2 Local Home Interface for Session Bean

The following code sample shows a local home interface for a session bean called `EmployeeLocalHome`.

```
package employee;

import javax.ejb.*;

public interface EmployeeLocalHome extends EJBLocalHome
{
    public EmployeeLocal create() throws CreateException, EJBException;
}
```

Creating the Component Interfaces

The component interfaces define the business methods of the bean that a client can invoke.

Creating the Remote Interface The remote interface defines the business methods that a remote client can invoke. Here are the requirements for developing the remote interface:

1. The remote interface of the bean must extend the `javax.ejb.EJBObject` interface, and its methods must throw the `java.rmi.RemoteException` exception.
2. You must declare the remote interface and its methods as `public` for remote clients.
3. The remote 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.

4. 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 2–3 Remote Interface Example for Employee Session Bean

The following code sample shows a remote interface called **Employee** with its defined methods, each of which will be implemented in the stateless session bean.

```
package employee;

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

public interface Employee extends EJBObject
{
  public Collection getEmployees()
    throws RemoteException;

  public EmpRecord getEmployee(Integer empNo)
    throws RemoteException;

  public void setEmployee(Integer empNo, String empName, Float salary)
    throws RemoteException;

  public EmpRecord addEmployee(Integer empNo, String empName,
                               Float salary)
    throws RemoteException;

  public void removeEmployee(Integer empNo)
    throws RemoteException;
}
```

Creating the Local Interface The local interface defines the business methods of the bean that a local (collocated) client can invoke.

1. The local interface of the bean must extend the `javax.ejb.EJBLocalObject` interface.
2. You declare the local interface and its methods as `public`.

Example 2–4 Local Interface for Employee Session Bean

The following code sample contains a local interface called `EmployeeLocal` with its defined methods, each of which will be implemented in the stateless session bean.

```
package employee;

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

public interface EmployeeLocal extends EJBLocalObject
{
  public Collection getEmployees() throws EJBException;

  public EmpRecord getEmployee(Integer empNo)
    throws FinderException, EJBException;

  public void setEmployee(Integer empNo, String empName, Float salary)
    throws FinderException, EJBException;

  public EmpRecord addEmployee(Integer empNo, String empName,
     Float salary) throws CreateException, EJBException;

  public void removeEmployee(Integer empNo)
    throws RemoveException, EJBException;
}
```

Implementing the Bean

The bean contains the business logic for your application. It implements the following methods:

1. The signature for each of these methods must match the signature in the remote or local interface.

   The bean in the example application consists of one class, `EmployeeBean`, that retrieves an employee's information.

2. The methods defined in the home interfaces are inherited from the `SessionBean` or `EntityBean` interface. The container uses these methods for controlling the life cycle of the bean. These include the `ejb<Action>` methods, such as `ejbActivate`, `ejbPassivate`, and so on.

3. The `ejbCreate` methods that correspond to the `create` method(s) that are declared in the home interfaces. The container invokes the appropriate `ejbCreate` method when the client invokes the corresponding `create` method.
4. 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.

Example 2–5 Employee Session Bean Implementation

The following code shows the bean implementation for the employee example. To compact this example, the try blocks for error processing are removed. See the full example on `http://otn.oracle.com`.

```
package employee;

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

public class EmployeeBean extends Object implements SessionBean
{
  public SessionContext ctx;
  public EmployeeLocal empLocal;

  public EmployeeBean() {}

  public EmpRecord addEmployee(Integer empNo, String empName,
           Float salary) throws CreateException
  {
      return empLocal.addEmployee(empNo, empName, salary);
  }

  public Collection getEmployees()
  {
      return empLocal.getEmployees();
  }

  public EmpRecord getEmployee(Integer empNo) throws FinderException
  {
```

```
        return empLocal.getEmployee(empNo);
    }

  public void setEmployee(Integer empNo, String empName, Float salary)
      throws FinderException
    {
        empLocal.setEmployee(empNo, empName, salary);
    }

    public void removeEmployee(Integer empNo) throws RemoveException
    {
        empLocal.removeEmployee(empNo);
    }

    public void ejbCreate() throws CreateException
    {
      // stateless bean has create method with no args. This
      // causes one bean instance to which multiple employees cling.
    }

    public void ejbRemove()
    {
        empLocal = null;
    }

    public void ejbActivate()  {   }

    public void ejbPassivate() {   }

    public void setSessionContext(SessionContext ctx) throws EJBException
    {
       this.ctx = ctx;
       Context context = new InitialContext();

       /*Lookup the EmployeeLocalHome object. The reference is retrieved
         from the application-local context (java:comp/env). The variable
         is specified in the assembly descriptor (META-INF/ejb-jar.xml).
       */
      Object homeObject =
                 context.lookup("java:comp/env/EmployeeLocalBean");
```

```
        // Narrow the reference to EmployeeHome.
        EmployeeLocalHome home = (EmployeeLocalHome) homeObject;

        // Create remote object and narrow the reference to Employee.
        empLocal = (EmployeeLocal) home.create();
    }

    public void unsetSessionContext()
    {
        this.ctx = null;
    }
}
```

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 mandated in the DTD file, which is provided at " `http://java.sun.com/dtd/ejb-jar_2_0.dtd`".

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 for more information.

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

Example 2–6 XML Deployment Descriptor for Employee Bean

The following is the deployment descriptor for a version of the employee example that uses a stateless session bean.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>
   <enterprise-beans>
      <session>
         <description>Session Bean Employee Example</description>
         <ejb-name>EmployeeBean</ejb-name>
         <home>employee.EmployeeHome</home>
         <remote>employee.Employee</remote>
         <local-home>employee.EmployeeLocalHome</local-home>
         <local>employee.EmployeeLocal</local>
         <ejb-class>employee.EmployeeBean</ejb-class>
         <session-type>Stateless</session-type>
         <transaction-type>Bean</transaction-type>
      </session>
   </enterprise-beans>
</ejb-jar>
```

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 *Oracle9iAS Containers for J2EE User's Guide*.

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

```
% jar cvf Employee-ejb.jar .
```

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

Prepare the EJB Application for Assembly

Before deploying, perform the following:

1. Modify the `application.xml` file with the modules of the enterprise Java application.
2. Archive all elements of the application into an EAR file.

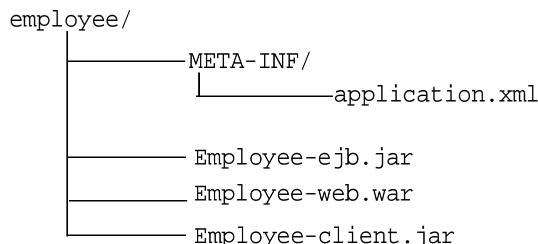
Modify the Application.XML File

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. Each module describes one of three things: EJB JAR, Web WAR, or any client files. Respectively, designate the `<ejb>`, `<web>`, and `<java>` elements in separate `<module>` elements.

- The `<ejb>` element specifies the EJB JAR filename.
- The `<web>` element specifies the Web WAR filename in the `<web-uri>` element, and its context in the `<context>` element.
- The `<java>` element specifies the client JAR filename, if any.

As Figure 2–2 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.

Figure 2–2 Archive Directory Format



For example, the following example modifies the `<ejb>`, `<web>`, and `<java>` module elements within `application.xml` for the Employee EJB application that also contains a servlet that interacts with the EJB.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE
Application 1.3//EN" "http://java.sun.com/dtd/application_1_3.dtd">
<application>
 <module>
    <ejb>Employee-ejb.jar</ejb>
 </module>
 <module>
  <web>
    <web-uri>Employee-web.war</web-uri>
    <context-root>/employee</context-root>
  </web>
 </module>
 <module>
    <java>Employee-client.jar</java>
 </module>
</application>
```

Create the 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 `Employee.EAR` file, execute the following in the employee directory contained in Figure 2–2:

```
% jar cvf Employee.ear .
```

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

Deploy the Enterprise Application to OC4J

After archiving your application into an EAR file, deploy the application to OC4J. See the *Oracle9iAS Containers for J2EE User's Guide* for information on how to deploy your application.

Access the EJB

All EJB clients—including standalone clients, servlets, JSPs, and JavaBeans—perform the following steps to instantiate a bean, invoke its methods, and destroy the bean:

1. Look up the home interface through a JNDI lookup, which is used for the life cycle management. Follow JNDI conventions for retrieving the bean reference, including setting up JNDI properties if the bean is remote to the client.
2. Narrow the returned object from the JNDI lookup to the home interface, as follows:
   a. When accessing the remote interface, use the `PortableRemoteObject.narrow` method to narrow the returned object.
   b. When accessing the local interface, cast the returned object with the local home interface type.
3. Create instances of the bean in the server through the returned object. Invoking the `create` method on the home interface causes a new bean to be instantiated and returns a bean reference.

   Note: For entity beans that are already instantiated, you can retrieve the bean reference through one of its finder methods.

4. Invoke business methods, which are defined in the component (remote or local) interface.
5. After you are finished, invoke the `remove` method. This will either remove the bean instance or return it to a pool. The container controls how to act on the `remove` method.

Example 2–7 A Servlet Acting as a Remote Client

The following example is executed from a servlet that acts as a remote client. Any remote client must set up JNDI properties before retrieving the object, using a JNDI lookup.

Note: The JNDI name is specified in the `<ejb-ref>` element in the client's `application-client.xml` file—as follows:

```
<ejb-ref>
    <ejb-ref-name>EmployeeBean</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>employee.EmployeeHome</home>
    <remote>employee.Employee</remote>
 </ejb-ref>
```

This code should be executed within a TRY block for catching errors, but the TRY block was removed to show the logic clearly. See the example for the full exception coverage.

```
public class EmployeeServlet extends HttpServlet
{
  EmployeeHome home;
  Employee empBean;

  public void init() throws ServletException
  {
   /* initialize JNDI context by setting factory, url, and credentials
       in a hashtable */
   Hashtable env = new Hashtable();
   env.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.evermind.server.rmi.ApplicationClientInitialContextFactory");
   env.put(Context.PROVIDER_URL, "ormi://myhost/employee");
   env.put(Context.SECURITY_PRINCIPAL, "admin");
   env.put(Context.SECURITY_CREDENTIALS, "welcome");

   /*1. Retrieve remote interface using a JNDI lookup*/
   Context context = new InitialContext();

   /**
   * Lookup the EmployeeHome object. The reference is retrieved from the
   * application-local context (java:comp/env). The variable is
```

```
    * specified in the application-client.xml).
    */
    Object homeObject = context.lookup("java:comp/env/EmployeeBean");

    //2. Narrow the reference to EmployeeHome. Since this is a remote
    //   object, use the PortableRemoteObject.narrow method.
    EmployeeHome home = (EmployeeHome)
           PortableRemoteObject.narrow(homeObject, EmployeeHome.class);

    //3. Create the remote object and narrow the reference to Employee.
    Employee empBean = (Employee)
          PortableRemoteObject.narrow(home.create(), Employee.class);
 }

  public void doGet(HttpServletRequest request,
                        HttpServletResponse response)
          throws ServletException, IOException
  {
    response.setContentType("text/html");
    ServletOutputStream out = response.getOutputStream();

    //4. Invoke a business method on the remote interface reference.
    Collection emps = empBean.getEmployees();

    out.println("<html>");
    out.println("<head><title>Employee Bean</title></head>");
    out.println("<body>");
    out.println("<table border='2'>");
    out.println("<tr><td>" + "<b>EmployeeNo</b>"
                + "</td><td>"    + "<b>EmployeeName</b>"
                + "</td><td>"    + "<b>Salary</b>"
                + "</td></tr>");

    Iterator iterator = emps.iterator();

    while(iterator.hasNext()) {
        EmpRecord emp = (EmpRecord)iterator.next();
        out.println("<tr><td>" + emp.getEmpNo()
                    + "</td><td>"    + emp.getEmpName()
                    + "</td><td>"    + emp.getSalary()
                    + "</td></tr>");
```

```
    }

    out.println("</table>");
    out.println("</body>");
    out.println("</html>");
    out.close();
 }
}
```

Example 2–8 A Session Bean Acting as a Local Client

The following example is executed from a session bean that is collocated with the Employee bean. Thus, the session bean uses the local interface, and the JNDI lookup does not require JNDI properties.

Note: The JNDI name is specified in the `<ejb-ref>` element in the session bean EJB deployment descriptor as follows:

```
 <ejb-local-ref>
   <ejb-ref-name>EmployeeLocalBean
   </ejb-ref-name>
   <ejb-ref-type>Session</ejb-ref-type>
   <local-home>employee.EmployeeLocalHome
   </local-home>
   <local>employee.EmployeeLocal</local>
</ejb-local-ref>
```

This code should be executed within a TRY block for catching errors, but the TRY block was removed to show the logic clearly. See the example for the full exception coverage.

```
// 1. Retreive the Home Interface using a JNDI Lookup
//Retrieve the initial context for JNDI. No properties needed when local
Context context = new InitialContext();

//Retrieve the home interface using a JNDI lookup using
// the java:comp/env bean environment variable specified in web.xml
Object homeObject = context.lookup("java:comp/env/EmployeeLocalBean");

//2. Narrow the returned object to be an EmployeeHome object. Since
//   the client is local, cast it to the correct object type.
```

```
EmployeeLocalHome home = (EmployeeLocalHome) homeObject;

//3. Create the local Employee bean instance, return the reference
Employee empBean = (Employee) home.create();

//4. Invoke a business method on the local interface reference.
Collection emps = empBean.getEmployees();
...
```

3

CMP Entity Beans

This chapter demonstrates simple Container Managed Persistence (CMP) EJB development with a basic configuration and deployment. Download the CMP entity bean example (`cmpapp.jar`) from the OC4J sample code page at `http://otn.oracle.com/sample_code/tech/java/oc4j/htdocs/oc4jsamplecode/oc4j-demo-ejb.html` on the OTN Web site.

This chapter demonstrates the following:

- Entity Bean Overview
- Creating Entity Beans
- Primary Key
- Persistence Fields
- CMP Types

See Chapter 6, "BMP Entity Beans", for an example of how to create a simple bean-managed persistent entity bean. For a description of persisting object relationships between EJBs, see Chapter 4, "Entity Relationship Mapping".

Entity Bean Overview

With EJB 2.0 and the local interface support, most developers agree that entity beans should be paired with a session bean, servlet, or JSP that acts as the client interface. The entity bean is a coarse-grain bean that encapsulates functionality and represents data and dependent 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, servlet, or JSP 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 under the umbrella of transactional, security, and concurrency support through the container. This and the following chapters focus on how to use the persistence functionality of the entity bean.

An entity bean manages persistent data in one of two ways: container-managed persistence (CMP) and bean-managed persistence (BMP). The primary difference between the two is as follows:

- Container-managed persistence—The EJB container manages data by saving it to a designated resource, which is normally a database. For this to occur, you must define the data that the container is to manage within the deployment descriptors. The container manages the data by saving it to the database.
- Bean-managed persistence—The bean implementation manages the data within callback methods. All the logic for storing data to your persistent storage must be included in the `ejbStore` method and reloaded from your storage in the `ejbLoad` method. The container invokes these methods when necessary.

Creating Entity Beans

To create an entity bean, perform the following steps:

1. Create the component interfaces for the bean. The component interfaces declare the methods that a client can invoke.
   a. The local component interface extends `javax.ejb.EJBLocalObject`.
   b. The remote component interface extends `javax.ejb.EJBObject`.
2. Create the home interfaces for the bean. The home interface defines the `create` and finder methods, including `findByPrimaryKey`, for your bean.
   a. The local home interface extends `javax.ejb.EJBLocalHome`.
   b. The remote home interface extends `javax.ejb.EJBHome`.
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 bean. This includes the following:
   a. The implementation for the methods that are declared in your component interfaces.
   b. The methods that are defined in the `javax.ejb.EntityBean` interface.
   c. The methods that match the methods that are declared in your home interfaces, which include the following:
      * The `ejbCreate` and `ejbPostCreate` methods with parameters matching the associated `create` method defined in the home interface.
      * Finder methods, other than `ejbFindByPrimaryKey` and `ejbFindAll`, that are defined in the home interface. The container generates the `ejbFindByPrimaryKey` and `ejbFindAll` method implementations—although you must still provide an empty method for each of these.
5. Create the bean deployment descriptor. The deployment descriptor specifies properties for the bean through XML elements. This step is where you identify the data within the bean that is to be managed by the container. These relationships could also entail other objects, which Chapter 4, "Entity Relationship Mapping" discusses.

6. If the persistent data is saved to or restored from a database and you are not using the defaults provided by the container, then you must ensure that the correct tables exist for the bean. In the default scenario, the container creates the table and columns for your data based on deployment descriptor and datasource information.
7. Create an EJB JAR file containing the bean, component interface, home interface, and the deployment descriptors. Once created, configure the `application.xml` file, create an EAR file, and deploy the EJB to OC4J.

The following sections demonstrate a simple CMP entity bean. This example continues the use of the employee example, as in other chapters—without adding complexity.

Home Interface

The home interface is primarily used for retrieving the bean reference, on which the client can request business methods.

- The local home interface extends `javax.ejb.EJBLocalHome`.
- The remote home interface extends `javax.ejb.EJBHome`.

The home interface must contain a `create` method, which the client invokes to create the bean instance. Each `create` method can have a different signature. For an entity bean, you must develop a `findByPrimaryKey` method. Optionally, you can develop other finder methods, which are named `find<name>`, for the bean.

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.

Our employee example provides 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 inquiry against the database for all employees.

Example 3–1 Entity Bean Employee Home Interface

The employee home interface provides a method to create the component interface. It also provides two finder methods: one to find a specific employee by an employee number and one that finds all employees. Last, it supplies a home interface business method, `calcSalary`, to calculate how much all employees cost the business.

The home interface is required to extend `javax.ejb.EJBHome` and define the `create` and `findByPrimaryKey` methods.

```
package employee;

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

public interface EmployeeLocalHome extends EJBLocalHome
{

  public EmployeeLocal create(Integer empNo) throws CreateException;

  // Find an existing employee
  public EmployeeLocal findByPrimaryKey (Integer empNo) throws FinderException;

  //Find all employees
  public Collection findAll() throws FinderException;

  //Calculate the Salaries of all employees
  public float calcSalary() throws Exception;
}
```

Component Interfaces

The entity bean component interfaces are the interfaces that the customer sees and invokes methods upon. The component interface defines the business logic methods for the entity bean instance.

- The local component interface extends `javax.ejb.EJBLocalObject`.
- The remote component interface extends `javax.ejb.EJBObject`.

The employee entity bean example exposes the local component interface, which contains methods for retrieving and updating employee information.

```
package employee;

import javax.ejb.*;
```

```
public interface EmployeeLocal extends EJBLocalObject
{
  public Integer getEmpNo();
  public void setEmpNo(Integer empNo);

  public String getEmpName();
  public void setEmpName(String empName);

  public Float getSalary();
  public void setSalary(Float salary);
}
```

Entity Bean Class

The entity bean class implements the following methods:

- The target methods for the methods that are declared in the home interface, which include the following:
  - the `ejbCreate` method
  - any finder methods, including `ejbFindByPrimaryKey`
  - any home interface business methods, which are prepended with `ejbHome` in the bean implementation. For example, the `calcSalary` method is implemented in the `ejbHomeCalcSalary` method.
- The business logic methods that are declared in the component interfaces.
- The methods that are inherited from the `EntityBean` interface.

However, with container-managed persistence, the container manages most of the target methods and the data objects, thereby leaving little for you to implement.

```
package employee;

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

public abstract class EmployeeBean implements EntityBean
{

  private EntityContext ctx;

  // Each CMP field has a get and set method as 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()
{
  // Constructor. Do not initialize anything in this method.
  // All initialization should be performed in the ejbCreate method.
  // The passivate() method may destroy these attributes when pooling
}

public float ejbHomeCalcSalary() throws Exception
{
  Collection c = null;
  try {
     c = ((EmployeeLocalHome)this.ctx.getEJBLocalHome()).findAll();

     Iterator i = c.iterator();
     float totalSalary = 0;
     while (i.hasNext())
     {
       EmployeeLocal e = (EmployeeLocal)i.next();
       totalSalary = totalSalary + e.getSalary().floatValue();
     }
     return totalSalary;
  }
  catch (FinderException e) {
    System.out.println("Got finder Exception "+e.getMessage());
    throw new Exception(e.getMessage());
  }
}

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
  {
    // Called just after bean created; container takes care of implementation
  }

  public void ejbStore()
  {
    // Called when bean persisted; container takes care of implementation
  }

  public void ejbLoad()
  {
     // Called when bean loaded; container takes care of implementation
  }

  public void ejbRemove() throws RemoveException
  {
     // Called when bean removed; container takes care of implementation
  }

  public void ejbActivate()
  {
    // Called when bean activated; container takes care of implementation.
    // If you need resources, retrieve them here.
  }

  public void ejbPassivate()
  {
    // Called when bean deactivated; container takes care of implementation.
    // if you set resources in ejbActivate, remove them here.
  }

  public void setEntityContext(EntityContext ctx)
  {
    this.ctx = ctx;
  }

  public void unsetEntityContext()
  {
    this.ctx = null;
  }
}
```

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 either primitive, serializable, or types that can be mapped to SQL types. You can define your primary key in one of two ways:

- Define the type of the primary key to be a well-known type. The type is defined in the `<prim-key-class>` in the deployment descriptor. The data field that is identified as the persistent primary key is identified in the `<primkey-field>` element in the deployment descriptor. 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. This class is declared in the `<prim-key-class>` element in the deployment descriptor. This is an advanced method for defining a primary key and is discussed in "Defining the Primary Key in a Class" on page 3-10.
- Specify an auto-generated primary key: If you specify a `java.lang.Object` as the primary key class type in `<prim-key-class>`, but do not specify the primary key name in `<primkey-field>`, then the primary key is auto-generated by the container. See Defining an Auto-Generated Primary Key on page 3-11 for more information.

For a simple CMP, you can define your primary key to be a well-known type by defining the data type of the primary key within the deployment descriptor.

The employee example defines its primary key as a `java.lang.Integer` and uses the employee number (`empNo`) as its primary key.

```
<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>
```

Once defined, the container creates a column in the entity bean table for the primary key and maps the primary key defined in the deployment descriptor to this column.

Defining the Primary Key in a Class

If your primary key is more complex than a simple data type, your primary key must be a class that is serializable of the name <*name*>PK. You define the primary key class within the `<prim-key-class>` element in the deployment descriptor.

The primary key variables must adhere to the following:

- Be defined within a `<cmp-field><field-name>` element in the deployment descriptor. This enables the container to manage the primary key fields.
- Be declared within the bean class as `public` and restricted to be either primitive, serializable, or types that can be mapped to SQL types.
- The names of the variables that make up the primary key must be the same in both the `<cmp-field><field-name>` elements and in the primary key class.

Within the primary key class, you implement a constructor for creating a primary key instance. Once the primary key class is defined in this manner, the container manages the class.

The following example places the employee number within a primary key class.

```
package employee;

public class EmployeePK implements java.io.Serializable
{
  public Integer empNo;

  public EmployeePK()
  {
    this.empNo = null;
  }

  public EmployeePK(Integer empNo)
```

```
  {
    this.empNo = empNo;
  }
}
```

The primary key class is declared within the `<prim-key-class>` element, and each of its variables are declared within a `<cmp-field><field-name>` element in the XML deployment descriptor, as follows:

```
<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 creates a column in the entity bean table for the primary key and maps the primary key class defined in the deployment descriptor to this column.

Defining an Auto-Generated Primary Key

If you specify a `java.lang.Object` as the primary key class type in `<prim-key-class>`, but do not specify the primary key name in `<primkey-field>`, then the primary key is auto-generated by the container.

The employee example defines its primary key as a `java.lang.Object`. Thus, the container auto-generates the primary key.

```
<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 called `autoid` in the entity bean table for the primary key of type `LONG`. The container uses random numbers for the primary key values.

Persistence Fields

The persistent data in your CMP bean can be one of the following:

- Persistence field—Simple data type that is persisted to a database table. This field is a direct attribute of the bean.
- Relationship field—Relationship to another bean.

Each type results in its own complex rules of how to configure. This section discusses persistence fields. For information on relationship fields, see Chapter 4, "Entity Relationship Mapping".

In CMP entity beans, you define the persistent data both in the bean instance and in the deployment descriptor.

- Get/Set methods in the bean instance: For each persistence and relationship field, both a get and a set method is created. For persistence fields, the data type of the parameter returned from the get method and passed into the set method defines the simple data type of the field. The name of the field is designated by the name of the get and set methods.

  The following XML shows the get and set methods for the employee name persistence field. A `String` is passed back from the get method and into the set method. Thus, the `String` is the simple data type of the field. If you remove the "get" and "set" from the method names and then lower the case of the first

letter, you have the persistence field name. In this case, `empName` is the persistence field name.

```
public abstract String getEmpName() throws RemoteException;
public abstract void setEmpName(String empName) throws RemoteException;
```

- The deployment descriptor defines these fields as persistent. Each field name must be defined in a `<cmp-field><field-name>` element in the EJB deployment descriptor. In the employee example, three persistence data fields are defined in the data accessor methods: `empNo`, `empName`, and `salary`.

  These fields are defined as persistent fields in the `ejb-jar.xml` deployment descriptor within the `<cmp-field><field-name>` element, as follows:

```
<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>
```

For these fields to be mapped to a database, you can do one of the following:

- Accept the defaults for these fields and avoid more deployment descriptor configuration. See "Default Mapping of Persistent Fields to the Database" on page 3-14 on how the default mapping occurs.
- Map the persistent data fields to columns in a table that exists in a designated database. The persistent data mapping is configured within the `orion-ejb-jar.xml` file. See "Explicit Mapping of Persistent Fields to the Database" on page 3-15 for more information.

Default Mapping of Persistent Fields to the Database

If you simply define the persistent fields in the `ejb-jar.xml` file, then OC4J provides the following mappings of these fields to the database:

- Database—The default database as set up in your OC4J instance configuration. For the JNDI name, use the `<location>` element for emulated data sources and `<ejb-location>` element for non-emulated data sources.

  Upon installation, the default database is a locally installed Oracle database that must be listening on port 5521 with a SID of `ORACLE`.

  > **Note:** You must change the "default" database configuration in the `data-sources.xml` file to coordinate with the default installation for an Oracle database. The default port and SID for an Oracle database are 1521 and ORCL, respectively.

  To customize the default database, change the first configured database to point to your database.

- Table—The container creates a default table where the name of the table is guaranteed to be unique. For all future redeployments, copy the generated `orion-ejb-jar.xml` file with this table name into the same directory as your `ejb-jar.xml` file. Thus, all future redeployments have the same table names as first generated. If you do not copy this file over, different table names may be generated.

  The table name is constructed with the following names, where each is separated by an underscore (_):

  – EJB name defined in `<ejb-name>` in the deployment descriptor.

  – JAR file name, including the `.jar` extension. However, all dashes (-) and periods (.) are converted to underscores (_) to follow SQL conventions. For example, if the name of your JAR file is `employee.jar`, then `employee_jar` is appended to the name.

  – Application name: This is the name of the application name, which you define during deployment.

  If the constructed name is greater than thirty characters, the name is truncated at twenty-four characters. Then six characters made up of an alphanumeric hash code is appended to the name.

For example, if the EJB name is `EmpBean`, the JAR file is `empl.jar`, and the application name is `employee`, then the default table name is `EmpBean_empl_jar_employee`.

- Column names—The columns in the entity bean table each have the same name as the `<cmp-field>` elements in the designated database. The data types for the database, translating Java data types to database data types, are defined in the specific database XML file, such as `oracle.xml`.

Explicit Mapping of Persistent Fields to the Database

As "Default Mapping of Persistent Fields to the Database" on page 3-14 discusses, your persistent data can be automatically mapped to a database table by the container. However, if the data represented by your bean is more complex or you do not want to accept the defaults that OC4J provides for you, then you can map the persistent data to an existing database table and its columns within the `orion-ejb-jar.xml` file. Once the fields are mapped, the container provides the persistence storage of the persistent data to the indicated table and rows.

For explicit mapping, Oracle recommends that you do the following:

1. Deploy your application with only the `ejb-jar.xml` elements configured.

   OC4J creates an `orion-ejb-jar.xml` file for you with the default mappings in them. It is easier to modify these fields than to create them from scratch. This provides you a method for choosing all or part of the modifications that are discussed in this section.

2. Modify the `<entity-deployment>` element in the `orion-ejb-jar.xml` file to use the database table and columns you specify.

Once you define persistent fields, each within its own `<cmp-field>` element, you can map each to a specific database table and column. Thus, you can map CMP fields to existing database tables. The mapping occurs with the OC4J-specific deployment descriptor: `orion-ejb-jar.xml`.

The explicit mapping of CMP fields is completed within an `<entity-deployment>` element. This element contains all mapping for an entity bean. However, the attributes and elements that are specific to CMP field mapping is as follows:

```
<entity-deployment name="..." location="..."
  table="..." data-source="...">
  <primkey-mapping>
    <cmp-field-mapping name="..." persistence-name="..." />
```

```
    </primkey-mapping>
    <cmp-field-mapping name="..." persistence-name="..." />
...
</entity-deployment>
```

Element or Attribute Name	Description
name	Bean name, which is defined in the ejb-jar.xml file in the <ejb-name> element.
location	JNDI location
table	Database table name
data-source	Data source for the database where the table resides
primkey-mapping	Definition of how the primary key is mapped to the table.
cmp-field-mapping	The name attribute specifies the <cmp-field> in the deployment descriptor, which is mapped to a table column in the persistence-name attribute.

You can configure the following within the orion-ejb-jar.xml file:

1. Configure the <entity-deployment> element for every entity bean that contains CMP fields that will be mapped within it.
2. Configure a <cmp-field-mapping> element for every field within the bean that is mapped. Each <cmp-field-mapping> element must contain the name of the field to be persisted.
   a. Configure the primary key in the <primkey-mapping> element contained within its own <cmp-field-mapping> element.
   b. Configure simple data types (such as a primitive, simple object, or serializable object) that are mapped to a single field within a single <cmp-field-mapping> element. The name and database field are fully defined within the element attributes.

Example 3–2 Mapping Persistent Fields to a Specific Database Table

The following example demonstrates how to map persistent data fields in your bean instance to database tables and columns by mapping the employee persistence data fields to the Oracle database table EMP.

- The bean is identified in the <entity-deployment> name attribute. The JNDI name for this bean is defined in the location attribute.

- The database table name is defined in the `table` attribute. And the database is specified in the `data-source` attribute, which should be identical to the `<ejb-location>` name of a `DataSource` defined in the `data-sources.xml` file.
- The bean primary key, `empNo`, is mapped to the database table column, `EMPNO`, within the `<primkey-mapping>` element.
- The bean persistent data fields, `empName` and `salary`, are mapped to the database table columns `ENAME` and `SAL` within the `<cmp-field-mapping>` element.

```
<entity-deployment name="EmpBean" location="emp/EmpBean"
  wrapper="EmpHome_EntityHomeWrapper2" max-tx-retries="3"
  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="ename" />
  <cmp-field-mapping name="salary" persistence-name="sal" />
  ...
</entity-deployment>
```

After deployment, OC4J maps the element values to the following:

Bean	Database
emp/EmpBean	EMP table, located at `jdbc/OracleDS` in the `data-sources.xml` file
empNo	`EMPNO` column as primary key
empName	`ENAME` column
salary	`SAL` column

CMP Types

In defining the container-managed persistent fields in the `<cmp-field>` and the primary key types, you can define simple data types and Java user classes that are serializable.

- Simple Data Types
- Serializable Classes
- Other Entity Beans or Collections

Simple Data Types

The following table provides a list of simple data types and the mapping of these types to SQL types and to Oracle database types.

Table 3–1 Simple Data Types

Known Type (native)	SQL type	Oracle type
java.lang.String	VARCHAR(255)	VARCHAR(255)
java.lang.Integer(int)	INTEGER	NUMBER(20,0)
java.lang.Long(long)	INTEGER	NUMBER(20,0)
java.lang.Short(short)	INTEGER	NUMBER(10,0)
java.lang.Double(double)	DOUBLE PRECISION	NUMBER(30,0)
java.lang.Float(float)	FLOAT	NUMBER(20,5)
java.lang.Byte(byte)	SMALLINT	NUMBER(10,0)
java.lang.Character(char)	CHAR	CHAR(1)
java.lang.Boolean(boolean)	BIT	NUMBER(1,0)
java.util.Date	DATETIME	DATE
java.util.Locale	VARCHAR(5)	VARCHAR(5)
java.sql.Date	DATE	DATE
java.sql.Clob	CLOB	CLOB
java.sql.Blob	BLOB	BLOB
java.sql.Timestamp	TIMESTAMP	TIMESTAMP
javax.mail.internet.InterenetAddress	VARCHAR(127)	VARCHAR(127)
java.math.BigInteger	VARCHAR(100)	VARCHAR(100)
java.io.Serializable	LONGVARBINARY	BLOB

You can modify the mapping of these data types in the `config/database-schema/<db>.xml` XML configuration files.

Serializable Classes

In addition to simple data types, you can define user classes that implement `Serializable`. These classes are stored in a BLOB in the database.

Other Entity Beans or Collections

You should not define other entity beans or `Collections` as a CMP type. Instead, these are relationships and should be defined within a CMR field.

- A relationship to another entity bean is always defined in a `<cmr-field>` relationship.
- `Collections` promote a "many" relationship and should be configured within a `<cmr-field>` relationship. Other types, such as `Lists`, are sub-interfaces of `Collections`. Oracle recommends that you use `Collections`.

4

Entity Relationship Mapping

This chapter discusses how to develop entity-to-entity relationships. As a developer, you can approach entity relationships from either the EJB development or database development viewpoint.

- EJB development—You can use UML diagrams to design the entity beans, and the cardinality and direction of the relationship between each bean, from the perspective of the EJB objects.
- Database development—You can use ERD diagrams to design the database tables, complete with the cardinality and direction designated by primary and foreign keys, that support the entity beans. The focus is on how the database maps each entity bean and the relationships between them.

This chapter starts by discussing entity relationships from the EJB development viewpoint. Next, it demonstrates how the deployment descriptor maps to database tables. If you want to design with the database development viewpoint, skip to "Mapping Relationship Fields to the Database" on page 4-10.

This chapter covers the following topics:

- Defining Entity-To-Entity Relationships
- Mapping Relationship Fields to the Database

Defining Entity-To-Entity Relationships

The following sections describe what an entity bean relationship can be and how to define them.

- Choosing Cardinality and Direction
- Defining Relationships

Choosing Cardinality and Direction

Cardinality refers to the number of entity objects on each side of the relationship. Thus, you can define the following types of relationship between EJBs:

- one-to-one
- one-to-many or many-to-one (dependent on the direction)
- many-to-many

In addition, each relationship can be one-way or two-way. This is referred to as the direction of the relationship. The one-way relationship is unidirectional; the two-way relationship is bidirectional. For example, a unidirectional relationship can be from an employee to an address. With the employee information, you can retrieve an address. However, with an address, you cannot retrieve the employee. An example of a bidirectional relationship is with a employee/projects example. Given a project number, you can retrieve the employees working on the project. Given an employee number, you can retrieve all projects that the employee is working on. Thus, the relationship is valid in both directions.

Normally, you use a unidirectional relationship when you want to reuse the target from multiple entities.

You define the cardinality and direction of the relationship between two beans in the deployment descriptor.

One-To-One Relationship Overview

A one-to-one relationship is the simplest relationship between two beans. One entity bean relates only to one other entity bean. If our company office contains only cubicles, and only a single employee can sit in each cubicle, then you have a one-to-one relationship: one employee in one designated cubicle. You define a unidirectional definition for this relationship as follows:

```
employee -> cubicle
```

However, if you have a cubicle number and want to determine who is assigned to it, you can assign a bidirectional relationship. This would enable you to retrieve the employee and find what cubicle he/she sits in. In addition, you could retrieve the cubicle number and determine who sits there. You define this bidirectional one-to-one relationship as follows:

```
employee <—> cubicle
```

One-To-Many or Many-To-One Relationship Overview

In a one-to-many relationship, one object can reference several instances of another. A many-to-one relationship is when many objects reference a single object. For example, an employee can have multiple addresses: a home address and an office address. If you define these relationships as unidirectional from the perspective of the employee, then you can look up the employee and see all of his/her addresses, but you cannot look up an address to see who lives there. However, if you define this relationship as bidirectional, then you can look up any address and see who lives there.

Many-To-Many Relationship Overview

A many-to-many relationship is complex. For example, each employee can be working on several projects. And each projects has multiple employees working on it. Thus, you have a many-to-many cardinality. The direction does not matter in this instance. You have the following cardinality:

```
employees <—> projects
```

In a many-to-many relationship, many objects can reference many objects. This cardinality is the most difficult to manage.

Defining Relationships

Here are the restrictions imposed on defining your relationships:

- You can define relationships only between CMP 2.0 entity beans.
- You must declare both EJBs in the relationship within the same deployment descriptor.
- Each relationship can use only the local interface of the target EJB.

The following are the requirements to define each cardinality type and its direction:

1. Define the abstract accessor methods (`get/set` methods) for each relationship field. The naming follows the same rules as for the persistence field abstract

accessor methods. For example, getAddress and setAddress methods are abstract accessor methods for retrieving and setting an address.

2. Define each relationship—its cardinality and direction—in the deployment descriptor. The relationship field name is defined in the `<cmr-field-name>` element. This name must be the same as the abstract accessor methods, without the `get/set` and the first letter in lower case. For example, the `<cmr-field-name>` would be `address` to compliment the `getAddress/setAddress` abstract accessor methods.
3. Declare if you want the cascade delete option for the one-to-one, one-to-many, and many-to-one relationships. The cascade delete is always specified on the "one" side of the relationship.

Define the Get/Set Methods for Each Relationship Field

Each relationship field must have the abstract accessor methods defined for it. In a relationship that sets or retrieves only a single entity, the object type passed back and forth must be the local interface of the target entity bean. In a relationship that sets or retrieves multiple objects, the object type passed back and forth is a `Set` or `Collection` containing local interface objects.

Example 4–1 Definition of Abstract Accessor Methods for the Employee Example

In this example, the employee can have only a single address, and you can retrieve the address only through the employee. This defines a one-to-one relationship that is unidirectional from the perspective of the employee. Then the abstract accessor methods for the employee bean are as follows:

```
public AddressLocal getAddress();
public void setAddress(AddressLocal address);
```

Because the cardinality is one-to-one, the local interface of the address entity bean is the object type that is passed back and forth in the abstract accessor methods.

The cardinality and direction of the relationship are defined in the deployment descriptor.

Example 4–2 Definition of One-To-Many Abstract Accessor Methods

If the employee example included a one-to-many relationship, the abstract accessor methods would pass back and forth a `Set` or `Collection` of objects, each of which contains target bean local interface objects. When you have a "many" relationship, multiple records are being passed back and forth.

A department contains many employees. In this one-to-many example, the abstract accessor methods for the department retrieves multiple employees. Thus, the abstract accessor methods pass a `Collection` or a `Set` of employees, as follows:

```
public Collection getDeptEmployees();
public void setDeptEmployees(Collection deptEmpl);
```

Declare the Relationships in the Deployment Descriptor

You define the relationships between entity beans in the same deployment descriptor the entity beans are declared. All entity-to-entity relationships are defined within the `<relationships>` element and you can define multiple relationships within this element. Each specific entity-to-entity relationship is defined within an `<ejb-relation>` element. The following XML demonstrates two entity-to-entity relationships defined within an application:

```
<relationships>
      <ejb-relation>
      ...
      </ejb-relation>
      <ejb-relation>
      ...
      </ejb-relation>
</relationships>
```

The following XML shows the full element structure for relationships:

```
<relationships>
  <ejb-relation>
    <ejb-relation-name> </ejb-relation-name>
    <ejb-relationship-role>
      <ejb-relationship-role-name> </ejb-relationship-role-name>
      <multiplicity> </multiplicity>
      <relationship-role-source>
         <ejb-name> </ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name> </cmr-field-name>
        <cmr-field-type> </cmr-field-type>
      </cmr-field>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
```

Table 4–1 describes the usage for each of these elements.

Table 4–1 Description of Relationship Elements of the Deployment Descriptor

Deployment Descriptor Element	Description
`<ejb-relation>`	Each entity-to-entity relationship is described in a single `<ejb-relation>` element.
`<ejb-relation-name>`	A user-defined name for the entity-to-entity relationship.
`<ejb-relationship-role>`	Each entity within the relationship is described within its own `<ejb-relationship-role>`. Thus, there are always two `<ejb-relationship-role>` entities within the `<ejb-relation>`.
`<ejb-relationship-role-name>`	A user-defined name to describe the role or involvement of the entity bean in the relationship.
`<multiplicity>`	The declaration of the cardinality for this entity. The value is "one" or "many."
`<relationship-role-source><ejb-name>`	The name of the entity bean. This must equal an EJB name defined in an `<entity><ejb-name>` element.
`<cmr-field><cmr-field-name>`	A user-defined name to represent the target bean reference. This name must match the abstract accessor methods. For example, if the abstract accessor fields are `getAddress()` and `setAddress()`, the CMR field must be `address`.
`<cmr-field><cmr-field-type>`	Optional. If "many", this type should be a `Collection` or `Set`. This is only specified for the "many" side to inform if a `Collection` or a `Set` is returned.

These relationships can be one-to-one, one-to-many, or many-to-many. The cardinality is defined within the `<multiplicity>` element. Each bean defines its cardinality within its own relationship. For example,

- One-to-one: For one employee to have a relationship with one address, the employee bean is declared with a `<multiplicity>` of one, and the address bean is declared with a `<multiplicity>` of one.
- One-to-many, many-to-one: For one department to have a relationship with multiple employees, the department bean is declared with a `<multiplicity>` of one, and the employee bean is declared with a `<multiplicity>` of many. For many employees to belong to a department, you define the same `<multiplicity>`.

- Many-to-many: For each employee to have a relationship with multiple projects and each project to have multiple employees working on it, the employee bean is declared with a `<multiplicity>` of many, and the project is declared with a `<multiplicity>` of many.

The direction of the relationship is defined by the presence of the `<cmr-field>` element. The reference to the target entity is defined within the `<cmr-field>`. If you are unidirectional, then only one entity within the relationship contains a reference to a target. In this case, the `<cmr-field>` is declared in the source entity and contains the target bean reference. If bidirectional, both entities should declare each other's target bean references within a `<cmr-field>` element.

The following demonstrates how to declare direction in the one-to-one employee and address example:

- Unidirectional: Define the `<cmr-field>` element within the employee bean section that references the address bean. Do not define a `<cmr-field>` element in the address bean section of the relationship.
- Bidirectional: Define a `<cmr-field>` element in the employee bean section that references the address bean. In addition, define a `<cmr-field>` element in the address bean section that references the employee bean.

Once you understand how to declare the cardinality and direction of the entity relationships, configuring the deployment descriptor for each relationship is simple.

Example 4–3 One-To-One Relationship Example

The employee example defines a one-to-one unidirectional relationship in which each employee has only one address. This relationship is unidirectional because you can retrieve the address from the employee, but you cannot retrieve the employee from the address. Thus, the employee object has a relationship to the address object.

In the deployment descriptor, you configure the following:

- The `<entity>` elements within the `<enterprise-beans>` section for each of the entity beans involved in the relationship. For this example, these include an `<entity>` element for the employee with an `<ejb-name>` of `EmpBean` and an `<entity>` element for the address with an `<ejb-name>` of `AddressBean.`
- An `<ejb-relationship>` element within the `<relationships>` section for the one-to-one relationship that contains the following:
  – An `<ejb-relationship-role>` element for the employee bean that defines its cardinality as "one" in its `<multiplicity>` element. The

<relationship-role-source> element defines the <ejb-name> as EmpBean, which is the same name in the <entity> element.

- An <ejb-relationship-role> element for the address bean that defines its cardinality as "one" in its <multiplicity> element. The <relationship-role-source> element defines the <ejb-name> as AddressBean, which is the same name in the <entity> element.

- A <cmr-field> element in the EmpBean relationship that points to the AddressBean. The <cmr-field> element defines address as the AddressBean reference. This element name matches the get and set method names, which are named getAddress and setAddress. These methods identify the local interface of the address entity bean as the data type that is returned from the get method and passed in on the set method.

```
<enterprise-beans>
  <entity>
    ...
    <ejb-name>EmpBean</ejb-name>
    <local-home>employee.EmpHome</local-home>
    <local>employee.Emp</local>
    <ejb-class>employee.EmpBean</ejb-class>
    ...
  </entity>
  <entity>
    ...
    <ejb-name>AddressBean</ejb-name>
    <local-home>employee.AddressHome</local-home>
    <local>employee.Address</local>
    <ejb-class>employee.AddressBean</ejb-class>
    ...
  </entity>
</enterprise-beans>
...
<relationships>
   <ejb-relation>
         <ejb-relation-name>Emp-Address</ejb-relation-name>
         <ejb-relationship-role>
            <ejb-relationship-role-name>Emp-has-Address
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source><ejb-name>EmpBean</ejb-name>
            </relationship-role-source>
            <cmr-field>
               <cmr-field-name>address</cmr-field-name>
```

```
            </cmr-field>
         </ejb-relationship-role>
         <ejb-relationship-role>
            <ejb-relationship-role-name>Address-has-Emp
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source><ejb-name>AddressBean</ejb-name>
            </relationship-role-source>
         </ejb-relationship-role>
      </ejb-relation>
</relationships>
```

Decide Whether to Use the Cascade Delete Option

When you have relationships between entity beans and the master entity bean is deleted, what happens to the slave beans? This question is answered by the cascade delete option. If you specify cascade delete to happen, the deletion of a master entity causes the deletion of all its slave relationship entity beans.

The cascade delete is defined in the object that is deleted automatically.

For example, an employee has a relationship with an address object. The employee object specifies cascade delete. When the employee, as master in this relationship, is deleted, the address, the slave, is also deleted.

In some instances, you do not want a cascade delete to occur. If you have a department that has a relationship with multiple employees within the department, you do not want all employees to be deleted when you delete the department.

You can only specify a cascade delete on a relationship if the master entity bean has a `<multiplicity>` of one. Thus, in a one-to-one, the master is obviously a "one". You can specify a cascade delete in a one-to-many relationship, but not in a many-to-one or many-to-many relationship.

The cascade delete is specified in the slave entity bean of the one-to-one or one-to-many relationship. Thus, when the master entity bean is deleted, the slave entity beans are deleted.

Example 4–4 Cascade Delete Requested in the Employee Example

The following deployment descriptor shows the definition of a one-to-one relationship with the employee and his/her address. When the employee is deleted, the slave entity bean—the address—is automatically deleted. You ensure the deletion by specifying the `<cascade-delete/>` element in the slave entity bean of

the relationship. In this case, specify the `<cascade-delete/>` element in the `AddressBean` definition.

```
<relationships>
   <ejb-relation>
         <ejb-relation-name>Emp-Address</ejb-relation-name>
         <ejb-relationship-role>
            <ejb-relationship-role-name>Emp-has-Address
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source><ejb-name>EmpBean</ejb-name>
            </relationship-role-source>
            <cmr-field>
               <cmr-field-name>address</cmr-field-name>
            </cmr-field>
         </ejb-relationship-role>
         <ejb-relationship-role>
            <ejb-relationship-role-name>Address-has-Emp
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
           <cascade-delete/>
            <relationship-role-source><ejb-name>AddressBean</ejb-name>
            </relationship-role-source>
         </ejb-relationship-role>
      </ejb-relation>
</relationships>
```

Mapping Relationship Fields to the Database

Each entity bean maps to a table in the database. Each of its persistent and relationship fields are saved within a database table in columns. For these fields to be mapped to a database, do one of the following:

- Accept the defaults for these fields and avoid more deployment descriptor configuration. See "Default Mapping of Relationship Fields to the Database" on page 4-11 to learn how the default mapping occurs.
- Map the fields to columns in a table that already exists in a designated database. The persistent data mapping is configured within the `orion-ejb-jar.xml` file. See "Explicit Mapping of Relationship Fields to the Database" on page 4-19 for more information.

Default Mapping of Relationship Fields to the Database

Note: This section discusses how OC4J maps relationship fields to the database. Chapter 3, "CMP Entity Beans" discusses persistent field mapping.

If you declare relationship fields only in the `ejb-jar.xml` file, then OC4J provides default mappings of these fields to the database. The default mapping is the same as for the persistent fields, as described in "Default Mapping of Persistent Fields to the Database" on page 3-14. describes.

Note: For all future redeployments, copy the generated `orion-ejb-jar.xml` file with this table name into the same directory as your `ejb-jar.xml` file. Thus, all future redeployments have the same table names as first generated. If you do not copy this file over, different table names may be generated.

In summary, these defaults include:

- Database—The default database as set up in your OC4J instance configuration.
- Default table—Each entity bean in the relationship represents data in its own database table. The name of the entity bean table is guaranteed to be unique, and so it is constructed with the following names, where each is separated by an underscore (_):
  - EJB name defined in `<ejb-name>` in the deployment descriptor.
  - JAR file name, including the `.jar` extension. However, all dashes (-) and periods (.) are converted to underscores (_) to follow SQL conventions. For example, if the name of your JAR file is `employee.jar`, then `employee_jar` is appended to the name.
  - Application name: This is the name of the application name, which you define during deployment.

  If the constructed name is greater than thirty characters, the name is truncated at twenty-four characters. An underscore and then five characters made up of an alphanumeric hash code is appended to the name for uniqueness.

For example, if the EJB name is `EmpBean`, the JAR file is `empl.jar`, and the application name is `employee`, then the default table name is `EmpBean_empl_jar_employee`.

- Column names in each table—The container generates columns in each table based on the `<cmp-field>` and `<cmr-field>` elements declared in the deployment descriptor. Each `<cmp-field>` is a column that relates to the entity bean data. Each `<cmr-field>` element represents a relationship. To establish a unidirectional relationship, only a single entity in the relationship defines a `<cmr-field>` in the deployment descriptor. To define a bidirectional relationship, both entities in the relationship define a `<cmr-field>`.

  For each `<cmr-field>` element, the container creates a foreign key that points to the primary key of the relevant object, as follows:

  – In a one-to-one relationship, the foreign key is created in the database table for the source EJB and is directed to the primary key of the target database table. For example, if one employee has one address, then the foreign key is created within the employee table that points to the primary key of the address table.

  – In one-to-many, many-to-one, and many-to-many relationships, an association table (third table) is created. The association table contains two foreign keys, where each points to the primary key of one of the entity tables.

  The translation rules for converting Java data types to database data types are defined in the specific database XML file located in `j2ee/home/config/database-schemas`, such as `oracle.xml`.

- Primary key generation—Both entity tables contain a primary key. The primary key can be defined or auto-generated. See "Primary Key" on page 3-9 for a full description.

  – Defined primary key: The primary key is generated as designated in the as a simple data type or a class.

  – Auto-generated primary key: If you specify a `java.lang.Object` as the primary key class type in `<prim-key-class>`, but do not specify the primary key name in `<primkey-field>`, then the primary key is auto-generated by the container.

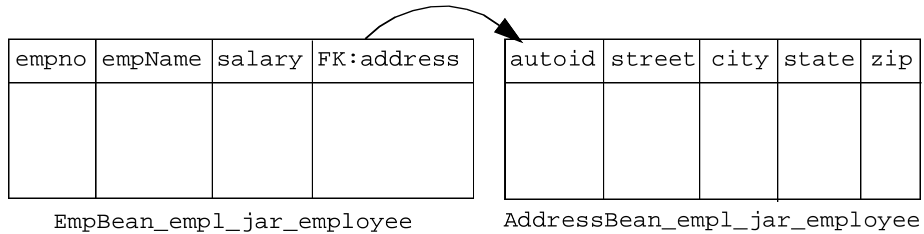
Default Mapping of the One-To-One Relationship

The one-to-one entity relationship is managed between the entity tables with a foreign key. Figure 4–1 demonstrates a one-to-one unidirectional relationship between the employee and address bean.

Note: Normally, you use a unidirectional relationship when you want to reuse the target for multiple entities. To reuse a table in the database, the target table must have the same definition for all tables using it. The target table does not normally have a foreign key pointing back to any of the source tables. For this reason, when you reuse a table, it is normally the target of a unidirectional relationship.

- The container generates the table names based on the entity bean names, the JAR file the beans are archived in, and the application name that they are deployed under. If the JAR filename is `empl.jar` and the application name is `employee`, then the table names are `EmpBean_empl_jar_employee` and `AddressBean_empl_jar_employee`.
- The container generates columns in each table based on the `<cmp-field>` and `<cmr-field>` elements declared in the deployment descriptor.
  - The columns for the `EmpBean` table are `empno`, `empname`, and `salary`. A foreign key is created called `address`, from the `<cmr-field>` declaration, that points to the primary key column of the `AddrBean` table.
  - The columns for the `AddressBean` table are an auto-generated long primary key and columns for stree, city, state, and zip.
- The primary key for the employee table is designated in the deployment descriptor as `empno`. The `AddressBean` is configured for an auto-generated primary key by specifying only `<primkey-class>` of `java.lang.Object`.

Figure 4–1 One-To-One Employee Relationship Example

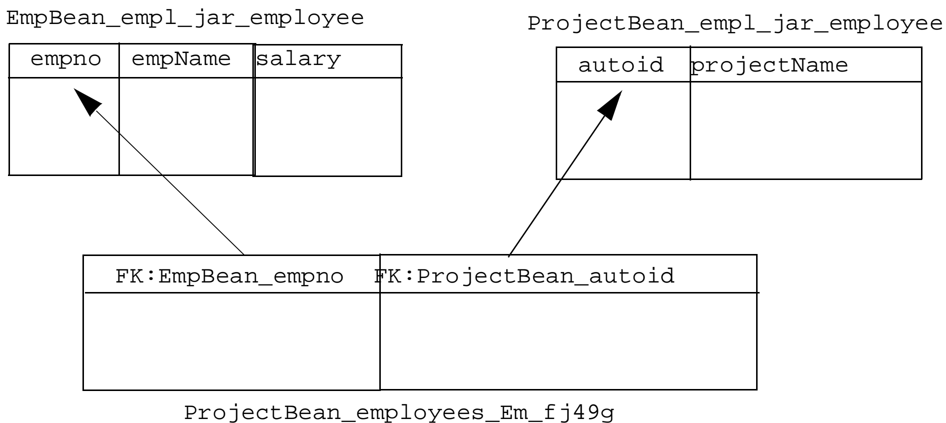


Default Mapping of One-To-Many and Many-To-Many Relationships

You cannot facilitate the one-to-many and many-to-many relationships using only a primary key and foreign key in the entity tables. To facilitate these relationships, the container creates an association table. The association table contains two columns, where each contains a foreign key to each of the entity tables in the relationship.

Figure 4–2 shows the tables that are created for the employee/project relationship. Each project can have multiple employees, and each employee can belong to several projects. Thus, the employee and project relationship is a many-to-many relationship. The container creates three tables to manage this relationship: the employee table, the project table, and the association table for both of these tables.

Figure 4–2 Many-To-Many Employee Relationship Example



The association table contains a foreign key column that points to the employee table and a foreign key column that points to the project table. The column names of

the association table are a concatenation of the entity bean name in `<ejb-name>` and its primary key name. If the primary key for the bean is auto-generated, then "`autoid`" is appended as the primary key name. For example, the foreign key that points to the employee table is the bean name of `EmpBean`, followed by the primary key name of `empno`, which results in the column name `EmpBean_empno`. The foreign key that points to the address table is the bean name of `ProjectBean` concatenated with `autoid`, because the primary key is auto-generated, which results in the column name `ProjectBean_autoid`.

The following is a demonstration of the association table for the employee/projects relationship. Employee 1 is assigned to projects a, b, and c. Project a involves employees 1, 2, and 3. The association table contains the following:

EmpBean_empno	ProjectBean_autoid
1	a
1	b
1	c
2	a
3	a

The association table details all relationships between the two entity beans.

Example 4–5 Deployment Descriptor for a Many-To-Many Relationship

To configure the employee/project many-to-many relationship in the deployment description, create an `<ejb-relation>` in which each bean defines its `<multiplicity>` as `many` and defines a `<cmr-field>` to the other bean of type `Collection` or `Set`.

```
<enterprise-beans>
  <entity>
    ...
    <ejb-name>EmpBean</ejb-name>
    <local-home>employee.EmpHome</local-home>
    <local>employee.Emp</local>
    ...
    <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>
```

```
    ...
  </entity>
  <entity>
    ...
    <ejb-name>ProjectBean</ejb-name>
    <local-home>employee.ProjectHome</local-home>
    <local>employee.Project</local>
    ...
    <cmp-field><field-name>projectName</field-name></cmp-field>
    <prim-key-class>java.lang.Object</prim-key-class>
    ...
  </entity>
</enterprise-beans>
<relationships>
  <ejb-relation>
    <ejb-relation-name>Emps-Projects</ejb-relation-name>
    <ejb-relationship-role>
      <ejb-relationship-role-name>Project-has-Emps</ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>ProjectBean</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>employees</cmr-field-name>
        <cmr-field-type>java.util.Collection</cmr-field-type>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <ejb-relationship-role-name>Emp-has-Projects</ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>EmpBean</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>projects</cmr-field-name>
        <cmr-field-type>java.util.Collection</cmr-field-type>
      </cmr-field>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
```

The container maps this definition to the following:

- The container generates the entity tables based on the entity bean names, the JAR file the beans are archived in, and the application name that they are

deployed under. If the JAR filename is empl.jar and the application name is employee, then the table names are EmpBean_empl_jar_employee and ProjectBean_empl_jar_employee.

- The container generates columns in each entity table based on the <cmp-field> elements declared in the deployment descriptor.
  - The columns for the EmpBean table are empno, empname, and salary. The primary key is designated as the empno field.
  - The columns for the ProjectBean table are autoid for an auto-generated primary key and a projectName column. The primary key is auto-generated because the <prim-key-class> is defined as java.lang.Object, and no <primkey-field> element is defined.
- The container generates an association table in the same manner as the entity table.
  - The association table name is created to include the two <cmr-field> definitions for each of the entity beans in the relationship. The format for the association table name consists of the following, separated by underscores: first bean name, its <cmr-field> to the second bean, second bean name, its <cmr-field> to the first bean, JAR file name, and application name. The rule of thirty characters also applies to this table name, as to the entity tables. Thus, the association table name for the employee/projects relationship is ProjectBean_employees_EmpBean_projects_empl_jar_employee. Because this name is over thirty characters, it is truncated to twenty-four characters, and then an underscore plus five characters of a hash code are added. Thus, the official association table would be something like ProjectBean_employees_Em_fj49g
  - Two columns in the association table are created. Each column name is a concatenation of the bean name and the primary key (or autoid if auto-generated). In our example, the column names would be EmpBean_empno and ProjectBean_autoid. These columns are foreign keys to the entity tables that are involved in the relationship. The EmpBean_empno foreign key points to the employee table; the ProjectBean_autoid foreign key points to the projects table.

Example 4–6 Deployment Descriptor for One-To-Many Bidirectional Relationship

The following XML demonstrates how to configure a single employee who can have multiple phone numbers. You can add another source of the phone numbers table, such as department phone numbers, so that the department entity bean has a one-to-many relationship with the phone number entity bean. This is why the employee to phone numbers relationship is unidirectional.

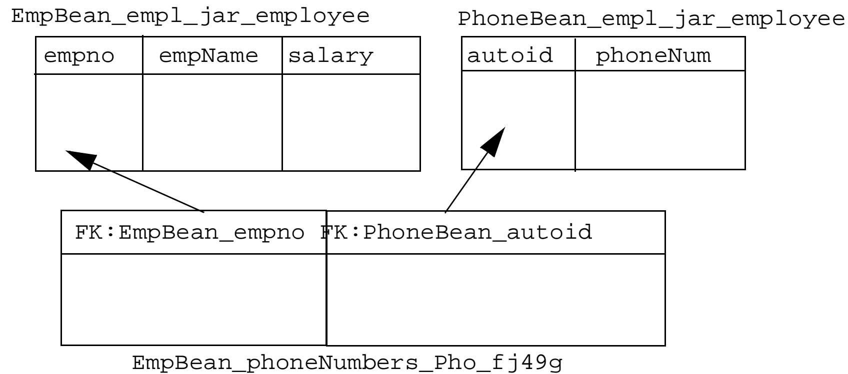
The employee entity bean, `EmpBean`, defines a `<cmr-field>` element designating a `Collection` of `phoneNumbers` within the `PhoneBean`.

```
<relationships>
  <ejb-relation>
   <ejb-relation-name>Emp-Phone</ejb-relation-name>
   <ejb-relationship-role>
     <ejb-relationship-role-name>Emp-PhoneNumbers</ejb-relationship-role-name>
     <multiplicity>One</multiplicity>
     <relationship-role-source>
        <ejb-name>EmpBean</ejb-name>
     </relationship-role-source>
     <cmr-field>
        <cmr-field-name>phoneNumbers</cmr-field-name>
        <cmr-field-type>java.util.Collection</cmr-field-type>
     </cmr-field>
   </ejb-relationship-role>
   <ejb-relationship-role>
     <ejb-relationship-role-name>Phone-has-Emp</ejb-relationship-role-name>
     <multiplicity>Many</multiplicity>
     <relationship-role-source>
        <ejb-name>PhoneBean</ejb-name>
     </relationship-role-source>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
```

The container maps this definition to the following:

- The container creates the entity tables, its primary keys, and columns in the same manner as the many-to-many relationship.
- The container creates the association table in the same manner as the many-to-many.

Figure 4–3 One-To-Many Relationship Employee Example



Explicit Mapping of Relationship Fields to the Database

As "Default Mapping of Relationship Fields to the Database" on page 4-11 discusses, your relationship fields can be automatically mapped to the database tables by the container. However, if you do not want to accept the defaults that OC4J provides for you, then you can map the relationships between entity beans within an existing database table and its columns in the `orion-ejb-jar.xml` file.

Note: "Explicit Mapping of Persistent Fields to the Database" on page 3-15 discusses how to explicitly map persistent fields. This section builds on that information and shows how the relationship mapping occurs.

For explicit mapping, Oracle recommends that you perform the following steps:

1. Deploy your application with only the `ejb-jar.xml` elements configured.

   OC4J creates an `orion-ejb-jar.xml` file for you, with the default mappings in it. It is easier to modify these fields than to create them from scratch. This provides you with a method for choosing all or part of the modifications that this discusses.

2. Copy the container-created `orion-ejb-jar.xml` file to your development environment.

3. Modify the `<entity-deployment>` element in the `orion-ejb-jar.xml` file to use the database table and columns you specify.

4. Rearchive and redeploy the application.

How you map relationship fields is dependent on the type of relationship:

- One-To-One Relationship Explicit Mapping: the source table contains a foreign key that points to the primary key of the target table. Thus, explicit mapping of this relationship field requires modifying the column name of the foreign key.
- One-To-Many and Many-To-Many Relationship Explicit Mapping: an association table is created that contains two columns, where each column is a foreign key that points to the primary key of the source and target tables. Thus, explicit mapping of this relationship requires modifying the association table name and its column names.
- Option for the One-To-Many Explicit Bidirectional Relationship: you can forego the association table and have the "many" table contain a foreign key that points to the "one" table. Thus, explicit mapping of this relationship requires modifying the "many" table and adding a foreign key that points to the primary key of the "one" table.

Modify elements and attributes of the `<entity-deployment>` element in the `orion-ejb-jar.xml` file to explicitly map relationship fields.

The following XML shows the relevant elements and attributes for explicit mapping of a one-to-one relationship:

```
<entity-deployment name=" " location=" " table=" " data-source=" ">
  <cmp-field-mapping name=" ">
    <entity-ref home=" ">
      <cmp-field-mapping name=" " persistence-name=" " />
    </entity-ref>
  </cmp-field-mapping>
</entity-deployment>
```

The following XML illustrates the relevant elements and attributes for explicitly identifying the association table for one-to-many or many-to-many:

```
<entity-deployment name=" " location=" " table=" " data-source=" ">
  <cmp-field-mapping name=" ">
    <collection-mapping table=" ">
      <primkey-mapping>
        <cmp-field-mapping name=" " persistence-name=" " />
      </primkey-mapping>
      <value-mapping type=" ">
        <cmp-field-mapping>
          <entity-ref home=" ">
            <cmp-field-mapping name=" " persistence-name=" "/>
```

```
        </entity-ref>
      </cmp-field-mapping>
    </value-mapping>
  </collection-mapping>
 </cmp-field-mapping>
</entity-deployment>
```

Element or Attribute	Description
<entity-deployment>	■ The name attribute identifies the <ejb-name> of the bean. ■ The location attribute identities the JNDI name of the bean. ■ The table attribute identifies the database table to which this entity bean is mapped. ■ The data-source attribute identifies the database in which the table resides.
<cmp-field-mapping>	Use this element to map a persistent field or a relationship field. For relationship fields, it will contain either an <entity-ref> for a one-to-one mapping or a <collection-mapping> for a one-to-many, many-to-one, or many-to-many relationship. ■ The name attribute identifies the <cmp-field> or <cmr-field> that is to be mapped. ■ The persistence-name attribute identifies the database column, which defaults to the <ejb-name> concatenated with the primary key of that entity bean.
<entity-ref>	Use this element to identify the primary key to which the foreign key points. The target bean and its primary key are identified in this element. The container uses this information to create a foreign key in the source table to point to the target table. ■ The name attribute identifies the bean name defined in <ejb-name>. ■ The <cmp-field-mapping> within this element identifies the target table column name.

Element or Attribute	Description
<collection-mapping>	Use this element to explicitly map the "many" side of a relationship. ■ The `table` attribute identifies the association table. In a one-to-many bidirectional relationship, you can specify the table of the "many" in this field to avoid the association table. This element defines two elements, one for each column in the association table: ■ <primkey-mapping> identifies the first foreign key in the association table. ■ <value-mapping> identifies the second foreign key.
<primkey-mapping>	Within the <collection-mapping>, use this element to identify the first foreign key.
<value-mapping>	Use this element to specify the second foreign key.

Note: This section first describes in detail how logical names defined in the `ejb-jar.xml` file relate to those in the `orion-ejb-jar.xml` file, and then how those logical variables defined in the `orion-ejb-jar.xml` file relate to the database table and column names. This document specifically chooses different names so that you can see which names must be the same. However, for efficiency and ease, you can make all these names the same. For example, a <cmr-field> defined in the `ejb-jar.xml` file relates to a `persistence-name` attribute in the `orion-ejb-jar.xml` file, which is then translated to a column name. **Your configuration is easier if all these names are the same.**

One-To-One Relationship Explicit Mapping

Figure 4–1 shows a one-to-one unidirectional relationship between an employee and an address. The employee table has a foreign key that points to the primary key of the employee. A one-to-one bidirectional relationship would add a foreign key to the address table that points to the employee.

```
<enterprise-beans>
  <entity>
    ...
    <ejb-name>EmpBean</ejb-name>
```

```
    <local-home>employee.EmpHome</local-home>
    <local>employee.Emp</local>
    <ejb-class>employee.EmpBean</ejb-class>
    ...
    <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>
    ...
  </entity>
  <entity>
    ...
    <ejb-name>AddressBean</ejb-name>
    <local-home>employee.AddressHome</local-home>
    <local>employee.Address</local>
    <ejb-class>employee.AddressBean</ejb-class>
    ...
    <cmp-field><field-name>addressPK</field-name></cmp-field>
    <cmp-field><field-name>addressDescription</field-name></cmp-field>
    <primkey-field>addressPK</primkey-field>
    <prim-key-class>java.lang.Integer</prim-key-class>
    ...
  </entity>
</enterprise-beans>
<relationshipe>
   <ejb-relation>
         <ejb-relation-name>Emp-Address</ejb-relation-name>
         <ejb-relationship-role>
            <ejb-relationship-role-name>Emp-has-Address
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source><ejb-name>EmpBean</ejb-name>
            </relationship-role-source>
            <cmr-field>
               <cmr-field-name>address</cmr-field-name>
            </cmr-field>
         </ejb-relationship-role>
         <ejb-relationship-role>
            <ejb-relationship-role-name>Address-has-Emp
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source><ejb-name>AddressBean</ejb-name>
            </relationship-role-source>
       </ejb-relationship-role>
```

```
      </ejb-relation>
</relationships
```

The EmpBean requires a foreign key to the AddressBean. Thus, the container modifies the <entity-deployment> element for the EmpBean to include a foreign key to the primary key of the AddressBean. The following mapping for this relationship is located in the orion-ejb-jar.xml file:

```
<entity-deployment name="EmpBean" location="emp/EmpBean"
    wrapper="EmpHome_EntityHomeWrapper2" max-tx-retries="3" 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="ename" />
       <cmp-field-mapping name="salary" persistence-name="sal" />
       <cmp-field-mapping name="address">
           <entity-ref home="AddressBean">
              <cmp-field-mapping name="address"
                                    persistence-name="addressPK" />
            </entity-ref>
       </cmp-field-mapping>
       ...
    </entity-deployment>
```

This mapping specifies:

- The <entity-deployment> attributes define the following:
  - name attribute: The name of the source bean is EmpBean.
  - location attribute: The JNDI location is emp/EmpBean.
  - table attribute: The database table in which the persistent data for this entity bean is stored is emp.
  - data-source attribute: The database in which this table resides is defined by the data source jdbc/OracleDS.
- The <cmp-field-mapping> elements identify the table columns and the persistent data to be stored in each: The columns in this table are empno, ename, sal, and address. The empno column contains the primary key, as defined in the EmpBean as empNo. The empName and salary CMP data are saved in the ename and sal columns. The address column is a foreign key that points to the primary key of the AddressBean table.

- The `address` foreign key points to the primary key of the `AddressBean`. The `<cmp-field-mapping>` for `address` describes this, as follows:

```
<cmp-field-mapping name="address">
      <entity-ref home="AddressBean">
         <cmp-field-mapping name="address"
                              persistence-name="addressPK" />
       </entity-ref>
</cmp-field-mapping>
```

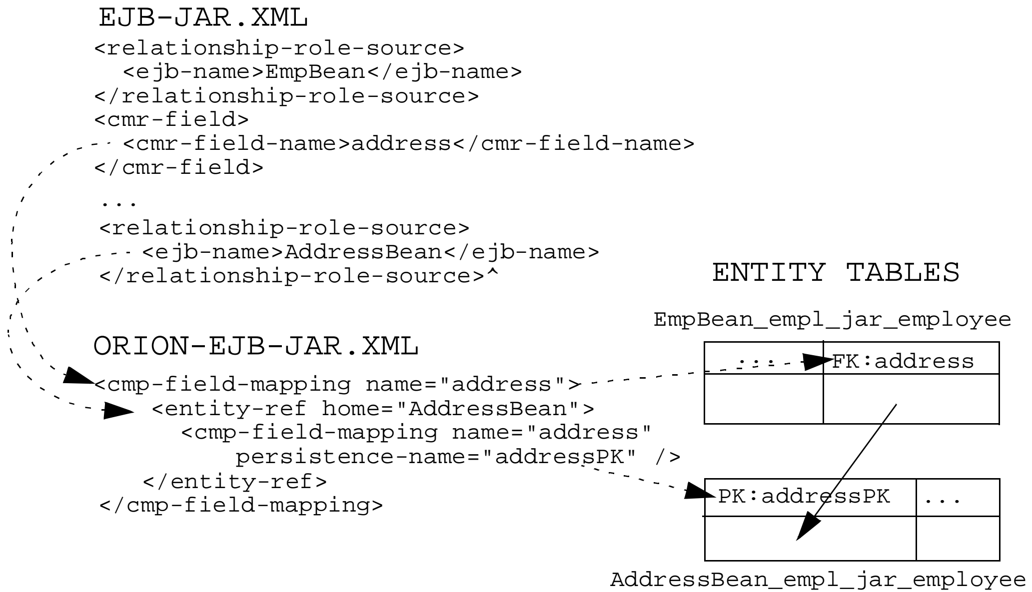
  - The first `<cmp-field-mapping>` `name` attribute identifies the `<cmr-field>` that was defined in the source bean. This is the column name for the foreign key called `address` in the `emp` table. However, it is not mapped to the primary key of the target table here.
  - The `<entity-ref>` `home` attribute identifies the `<ejb-name>` of the target bean. The target in this example is the `AddressBean`. The container understands the entity table in which `AddressBean` is stored.
  - The second `<cmp-field-mapping>` `name` attribute maps the source foreign key to the primary key of the target table. Thus, this second `name` attribute is identical to the first—`address`, which is the foreign key of the `EmpBean` table. The `persistence-name` attribute identifies the primary key column name of the target bean. In this example, the primary key of the `AddressBean` table is the `addressPK` column.

    In the original example, the container auto-generates a primary key for the target `AddressBean` table. For the auto-generated example, the second `<cmp-field>` mapping is mapped to an auto-generated primary key, known as `autoid`, as follows:

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

Figure 4–4 displays the relationship mapping of the `EmpBean` `address` foreign key to the `AddressBean` `addressPK` primary key.

Figure 4–4 Demonstration of Explicit Mapping for a One-To-One Relationship



In summary, an `address` column in the `EmpBean_empl_jar_employee` table is a foreign key that points to the primary key, `addressPK`, in the `AddressBean_empl_jar_employee` table. For the example in which the `AddressBean` has an auto-generated primary key, an `address` column in the `EmpBean_empl_jar_employee` table is a foreign key that points to the primary key, `autoid`, in the `AddressBean_empl_jar_employee` table.

One-To-Many and Many-To-Many Relationship Explicit Mapping

Figure 4–3 shows a one-to-many unidirectional relationship between an employee and his/her phone numbers. Because this involves a "many" in the relationship, an association table is created. The association table is the same whether this is a unidirectional or bidirectional, or one-to-many or many-to-many relationship.

In the `ejb-jar.xml` file, the cardinality is defined in the `<relationships>` element. The container knows from this definition whether the relationship is one-to-many or many-to-many. In the `orion-ejb-jar.xml` file, the mapping of this relationship to an association table is described in a `<collection-mapping>` element. Because the cardinality is already known, only one entity in the relationship defines the `<collection-mapping>` element.

- In a one-to-many relationship, the "one" entity bean defines the `<collection-mapping>` element as it receives back a `Collection` or `Set` of the target.
- In a many-to-many relationship, only one of the entity beans in the relationship fully defines the `<collection-mapping>` element with the association table specifications. The other entity bean has an empty `<collection-mapping>` element.

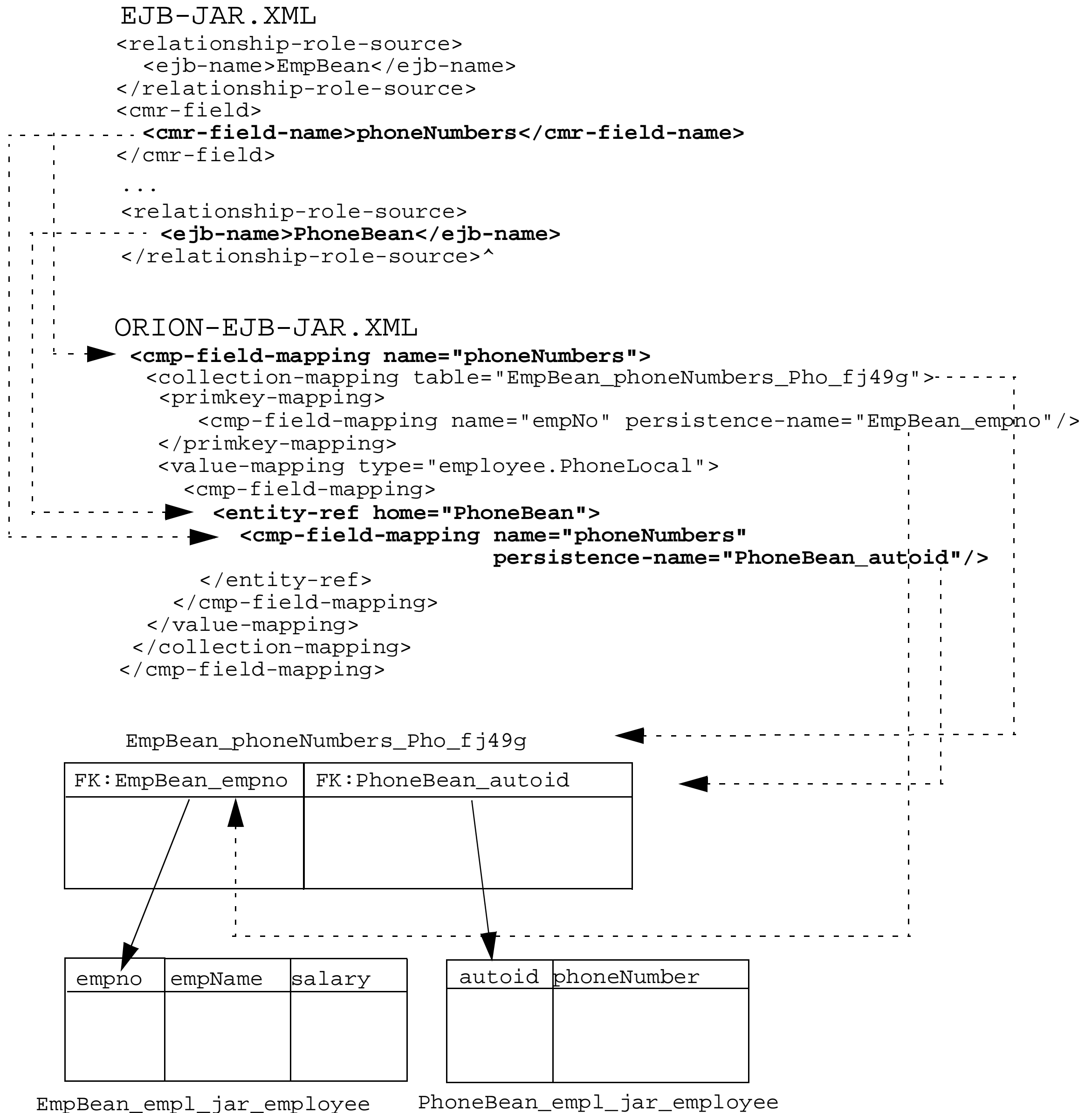
In the `orion-ejb-jar.xml` file for the employee example, the `EmpBean` `<entity-deployment>` element defines the `<collection-mapping>` element to designate a `Collection` of phone numbers. The `<collection-mapping>` element defines the association table.

```
<entity-deployment name="EmpBean" location="emp/EmpBean"
wrapper="EmpHome_EntityHomeWrapper2" max-tx-retries="3"
table="EmpBean_phoneNumbers_Pho8fj49g" data-source="jdbc/OracleDS">
   <primkey-mapping>
        <cmp-field-mapping name="empNo" persistence-name="empno" />
   </primkey-mapping>
   <cmp-field-mapping name="empName" persistence-name="ename" />
   <cmp-field-mapping name="salary" persistence-name="sal" />
   <cmp-field-mapping name="phoneNumbers">
     <collection-mapping table="EmpBean_phoneNumbers_Pho8fj49g">
       <primkey-mapping>
           <cmp-field-mapping name="empNo" persistence-name="EmpBean_empno" />
       </primkey-mapping>
       <value-mapping type="employee.PhoneLocal">
           <cmp-field-mapping>
             <entity-ref home="PhoneBean">
              <cmp-field-mapping name="phoneNumbers"
                                         persistence-name="PhoneBean_autoid"/>
             </entity-ref>
            </cmp-field-mapping>
        </value-mapping>
      </collection-mapping>
    </cmp-field-mapping>
    ...
    </entity-deployment>
```

- The `<collection-mapping>` element is contained within the `<cmp-field-mapping>` for `phoneNumbers`, which is the `EmpBean` `<cmr-field>` element definition. It defines the association table name in the `table` attribute, which currently defines the association table name as `EmpBean_phoneNumbers_Pho8fj49g`.

- Both primary keys of the entity beans are defined in the `<primkey-mapping>` and `<value-mapping>` elements respectively.
  - The `<primkey-mapping>` element defines the association table foreign key of the current entity bean, which is `EmpBean_empno`.
  - The `<value-mapping>` element defines the association table foreign key of the target bean, which is `PhoneBean_autoid`.
- The column names of the association table are defined by a concatenation of the entity bean name and the primary key, separated by an underscore (_). Thus, the column names for this example are `EmpBean_empno` and `PhoneBean_autoid`.
- The `<value-mapping>` element specifies the target entity bean.
  - The `type` attribute of the `<value-mapping>` element defines the target bean local interface that is returned to the source entity bean.
  - The `<ejb-name>` of the target entity bean is defined in the `<entity-ref> home` attribute.

Figure 4–5 Demonstration of Explicit Mapping for a One-To-Many Relationship

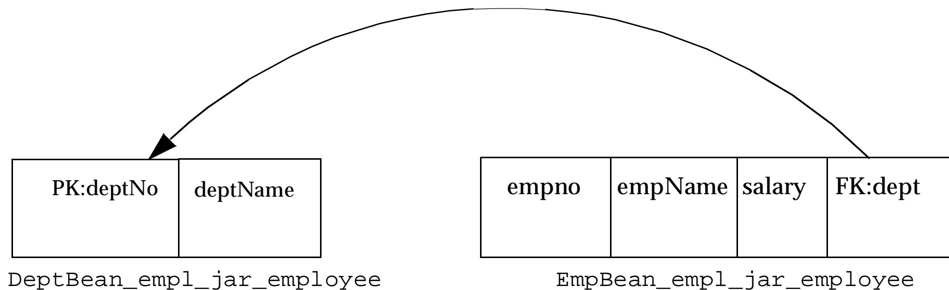


Option for the One-To-Many Explicit Bidirectional Relationship

You can bypass an association table in the one-to-many bidirectional entity relationship. The "one" relationship has a primary key that points to the "many"; the "many" has a foreign key that points back. With both tables maintaining primary keys, and the "many" table maintaining a foreign key back to the "one" table, there is no need for an association table.

Figure 4–6 shows the department<->employee example, where each employee belongs to only one department and each department can contain multiple employees. The department table has a primary key. The employee table has a primary key to identify each employee and a foreign key to point back to the employee's department. If you want to find the department for a single employee, a simple SQL statement retrieves the department information from the foreign key. To find all employees in a department, the container performs a JOIN statement on both the department and employee tables and retrieves all employees with the designated department number.

Figure 4–6 One-To-Many Bidirectional Relationship Option



This is not the default behavior. To have this type of relationship, do one of the following:

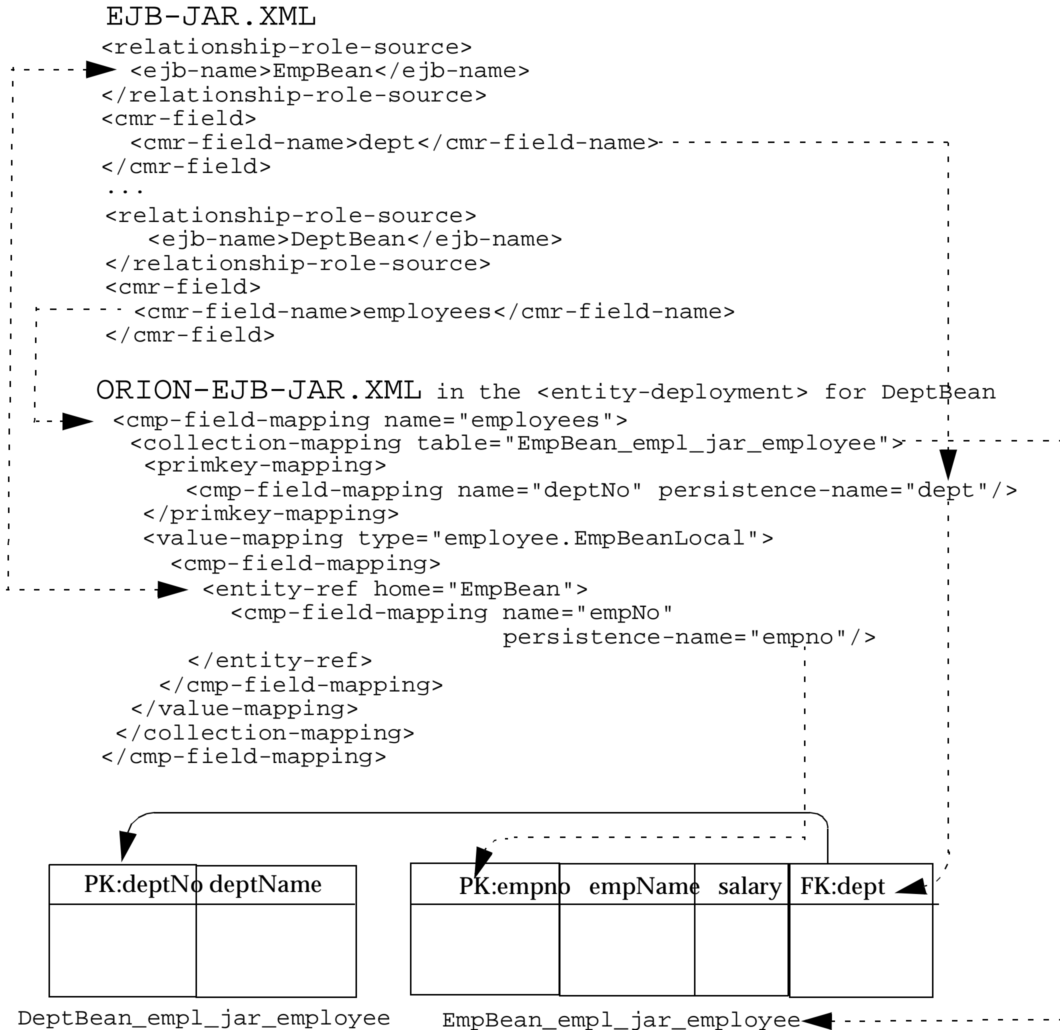
- Specify `-DassociateUsingThirdTable=false` on the OC4J.JAR startup options before deployment. Restart the OC4J instance.
- Manipulate the `<collection-mapping>` element in the `orion-ejb-jar.xml` file.

To manipulate the `<collection-mapping>` element in the `orion-ejb-jar.xml` file, you modify the `<entity-deployment>` element for the "one" entity bean, which contains the `Collection`, as follows:

1. Change the association table in the `<collection-mapping> table` attribute to be the "many" table. In this example, you would modify this attribute to be the `EmpBean_empl_jar_employee` table.
2. Modify the column names in the `persistence-name` attribute for each entity bean in the `<collection-mapping>` table as follows:
   - The `persistence-name` of the `<primkey-field>` element should be the primary key of the source entity bean and the database column name of the foreign key in the target entity bean.
   - The `<value-mapping>` element provides a pointer to the target bean—the "many" bean. Thus, the `<entity-ref> home` attribute should be the target entity bean name. In the second `<cmp-field-mapping>` element, the `name` attribute contains the `<cmr-field>` defined in the "one" entity bean in the `orion-ejb-jar.xml` file. The `persistence-name` attribute defines the primary key column of the "one" entity bean table.

Figure 4–7 demonstrates how the department/employee one-to-many bidirectional example is mapped without the use of an association table.

Figure 4–7 Explicit Mapping for One-To-Many Bidirectional Relationship Example



5 EJB Query Language

In EJB 2.0, you can specify query methods using the standardized query language, EJB Query Language (EJB QL).

The EJB 2.0 specification and various off-the-shelf books document EJB QL extensively. This chapter briefly overviews the development rules for these methods, but does not describe the EJB QL syntax in detail.

Refer to the EJB 2.0 specification and the following books for detailed syntax:

- *Enterprise JavaBeans, 3rd Edition* by Richard Monson-Haefel, O'Reilly Publishers
- *Special Edition Using Enterprise JavaBeans 2.0* by Chuck Cavaness and Brian Keeton, Que Publishers

This chapter covers the following subjects:

- EJB QL Overview
- Query Methods Overview
- Deployment Descriptor Semantics
- Finder Method Example
- Select Method Example

EJB QL Overview

EJB QL is a query language that is similar to SQL. In fact, your knowledge of SQL is beneficial in using EJB QL. SQL applies queries against tables, using column headings. EJB QL applies queries against entity beans, using the entity bean name and its CMP and CMR fields 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, primary keys, foreign keys, and column names. EJB QL is portable to all databases supported by your container.

Query Methods Overview

Query methods can be finder or select methods:

- Finder Methods: Use finder methods to retrieve entity bean references.
- Select Methods: Select methods are for internal use for the entity bean only. Use them to retrieve either entity bean references or CMP values.

Both query methods must throw the `FinderException`.

Finder Methods

Finder methods are used to retrieve entity bean references. The `findByPrimaryKey` finder method is always defined in both home interfaces (local and remote) to retrieve the entity reference for this bean using a primary key. You can define other finder methods in either or both the home interfaces to retrieve one or several entity bean references.

Do the following to define finder methods:

1. Define the `find<name>` method in the desired home interface. You can specify different finder methods in the remote or the local home interface. If you define the same finder method in both home interfaces, it maps to the same bean class definition. The container returns the appropriate home interface type.
2. Define the conditional statement for the finder method in the deployment descriptor. An EJB QL statement is created for each finder method in its own `<query>` element. The container uses this statement to translate the condition on how to retrieve the entity bean references into the relevant SQL statements.

If you retrieve only a single entity bean reference, the container returns the same type as returned in the find<name> method. If you request multiple entity bean references, you must define the return type of the find<name> method to return a Collection. If you want to ensure that no duplicates are returned, specify the DISTINCT keyword in the EJB QL statement. An empty Collection is returned if no matches are found.

Backward Compatibility for Finder Methods

In Release 2 (9.0.2) and previous releases, OC4J had its own methodology for finder methods. These finder methods were configured in the orion-ejb-jar.xml file in a <finder-method> element. Each <finder-method> element specified a partial or full SQL statement in its query attribute, as follows:

```
<finder-method query="">
OR
<finder-method query="$empname = $1">
```

If you have a <finder-method> with a query attribute from a previous release, it overrides any EJB QL modifications to the same method in the ejb-jar.xml file. The orion-ejb-jar.xml configured <finder-method> query attribute definition has higher priority.

To have the previous finder method modified with EJB QL, erase the query attribute of the <finder-method> in the orion-ejb-jar.xml file and redeploy the application. OC4J notes that the query attribute is not present and places the EJB QL equivalent in the <finder-method> element.

Select Methods

Select methods are for internal use within the bean. These methods cannot be called from a client. Thus, you do not define them in the home interfaces. Select methods are used to retrieve entity bean references or the value of a CMP field.

Do the following to define select methods:

1. Define an ejbSelect<name> method in the bean class for each select method. Each method is defined as public abstract. The SQL that is necessary for this method is not included in the implementation.
2. Define the conditional statement for the select method in the deployment descriptor. An EJB QL statement is created for each select method in its own <query> element. The container uses this statement to translate the condition into the relevant SQL statements.

Return Objects

Here are the rules for defining return types for the select method:

- Single object: If you retrieve only a single item, the container returns the same type as returned in the `ejbSelect<name>` method.
- Multiple objects: If you request multiple items, you must define the return type of the `ejbSelect<name>` method as either a `Set` or `Collection`. A `Set` eliminates duplicates. A `Collection` may include duplicates. For example, if you want to retrieve all zip codes of all customers, use a `Set` to eliminate duplicates. To retrieve all customer names, use a `Collection` to retrieve the full list. An empty `Collection` or `Set` is returned if no matches are found.
  - Bean interface: If you return the bean interface, the default interface type returned within the `Set` or `Collection` is the local bean interface. You can change this to the remote bean interface in the `<result-type-mapping>` element, as follows:

    ```
    <result-type-mapping>Remote</result-type-mapping>
    ```

  - CMP values: If you return a `Set` or `Collection` of CMP values, the container determines the object type from the EJB QL select statement.

Deployment Descriptor Semantics

The structure required for defining both types of query methods is the same in the deployment descriptor.

1. You must define the `<abstract-schema-name>` element for each entity bean referred to in the EJB QL statement. This element defines the name that identifies the entity bean in the EJB QL statement. Thus, if you define your `<abstract-schema-name>` as `Employee`, then the EJB QL uses `Employee` in its EJB QL to refer to the `EmpBean` entity bean.
2. You must define the `<query>` element for each query method (finder and select), except for the `findByPrimaryKey` finder method. The `<query>` element has two main elements:
   - The `<method-name>` element identifies the finder or select method. The finder method is the same name as defined in the component home interfaces. The select method is the same name as defined in the bean class.
   - The `<ejb-ql>` element contains the EJB QL statement for this method.

Example 5–1 Employee FindAll Deployment Descriptor Definition

The following example shows the `EmpBean` entity bean definition.

- The `<entity>` element defines its `<abstract-schema-name>` as `Employee`.
- A `<query>` element defines a finder method, `findAll`, in which the EJB QL statement refers to the `Employee` name.

```
<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></description>
      <query-method>
        <method-name>findAll</method-name>
        <method-params />
      </query-method>
    <ejb-ql>Select OBJECT(e) From Employee e</ejb-ql>
  </query>
  ...
</entity>
```

The EJB QL statement for the `findAll` method is simple. It selects objects, identified by the variable `e`, from the `Employee` entity beans. Thus, it selects all `Employee` entity bean objects.

Finder Method Example

To define finder methods in a CMP entity bean, do the following:

1. Define the finder method in one or both of the home interfaces.
2. Define the finder method definition in the deployment descriptor.

Define the Finder Method in the Home Interface

You must add the finder method to the home interface. For example, if you want to retrieve all employees, define the `findAll` method in the home interface (local home interface for this example), as follows:

```
public Collection findAll() throws FinderException;
```

To retrieve data for a single employee, define the `findByEmpNo` in the home interface, as follows:

```
public EmployeeLocal findByEmpNo(Integer empNo)
     throws FinderException;
```

The returned bean interface is the local interface, `EmployeeLocal`. The input parameter is an employee number, `empNo`, which is substituted in the EJB QL `?1` parameter.

Define the Finder Method Definition in the Deployment Descriptor

Each finder method is defined in the deployment descriptor in a `<query>` element. Example 5–1 contains the EJB QL statement for the `findAll` method. The following example shows the deployment descriptor for the `findByEmpNo` method:

```
<query>
  <description></description>
  <query-method>
    <method-name>findByEmpNo</method-name>
    <method-params>
      <method-param>java.lang.Integer</method-param>
    </method-params>
  </query-method>
  <ejb-ql>SELECT OBJECT(e) FROM Employee e WHERE e.empNo = ?1
  </ejb-ql>
</query>
```

The EJB QL statement for the `findByEmpName` method selects the `Employee` object where the employee number is substituted in the EJB QL `?1` parameter. The `?` symbol denotes a place holder for the method parameters. Thus, the `findByEmpNo` is required to supply at least one parameter. The `empNo` passed in on the `findByEmpNo` method is substituted in the ?1 position here. The variable, `e`, identifies the `Employee` object in the `WHERE` condition.

Relationship Finder Example

For the EJB QL statement that involves a relationship between entity beans, both entity beans are referenced within the EJB QL statement. The following example shows the `findByDeptNo` method. This finder method is defined within the employee bean, which references the department entity bean. This method retrieves all employees that belong to a department.

```
<query>
  <description></description>
  <query-method>
    <method-name>findByDeptNo</method-name>
    <method-params>
      <method-param>java.lang.Integer</method-param>
    </method-params>
  </query-method>
  <ejb-ql>SELECT OBJECT(e) From Employee e, IN (e.dept)
         AS d WHERE d.deptNo = ?1
  </ejb-ql>
</query>
```

The `<abstract-schema-name>` element for the employee bean is `Employee`. The employee bean defines a relationship with the department bean through a CMR field, called `dept`. Thus, the department bean is referenced in the EJB QL through the `dept` CMR field. The department primary key is `deptNo`. The department number that the query is executed with is given in the input parameter and substituted in `?1`.

Select Method Example

To define select methods in a CMP entity bean, do the following:

1. Define the select method in the bean class as `ejbSelect<name>`.
2. Define the select method definition in the deployment descriptor.

Define the Select Method in the Bean Class

Add the select method in the bean class. For example, if you want to retrieve all employees whose salary falls within a range, define the `ejbSelectBySalaryRange` method in the bean class, as follows:

```
public abstract Collection ejbSelectBySalaryRange(Float s1, Float s2)
                          throws FinderException;
```

Because the select method retrieves multiple employees, a Collection is returned. The low and high end of the salary range are input parameters, which are substituted in the EJB QL ?1 and ?2 parameters. The order of the declared method parameters is the same as the order of the ?1, ?2, ... ?n EJB QL parameters.

Define the Select Method Definition in the Deployment Descriptor

Each select method is defined in the deployment descriptor in a <query> element. The following example shows the deployment descriptor for the ejbSelectBySalaryRange method:

```
<query>
  <description></description>
  <query-method>
    <method-name>ejbSelectBySalaryRange</method-name>
    <method-params>
      <method-param>java.lang.Float</method-param>
      <method-param>java.lang.Float</method-param>
    </method-params>
  </query-method>
  <ejb-ql>SELECT DISTINCT OBJECT(e) From Employee e
           WHERE e.salary BETWEEN ?1 AND ?2
  </ejb-ql>
</query>
```

The ejbSelectBySalaryRange method provides two input parameters, both of type float. The types of these expected input parameters are defined in the <method-param> elements.

The EJB QL is defined in the <ejb-ql> element. This select method evaluates the CMP field of salary. It is designated within the EJB QL statement by the e.salary. The e represents the Employee objects; the salary represents the CMP field within that object. Separating it with a period shows the relationship between the entity bean and its CMP field.

The two input parameters designate the low and high salary ranges and are substituted in the ?1 and ?2 positions respectively.

The DISTINCT keyword ensures that no duplicate records are returned.

6

BMP Entity Beans

You must implement the storing and reloading of data in a bean-managed persistent (BMP) bean. The bean implementation manages the data within callback methods. All the logic for storing data to your persistent storage is included in the `ejbStore` method, and reloaded from your storage in the `ejbLoad` method. The container invokes these methods when necessary.

This chapter demonstrates simple BMP EJB development with a basic configuration and deployment. Download the BMP entity bean example (`bmpapp.jar`) from the OC4J sample code page at `http://otn.oracle.com/sample_code/tech/java/oc4j/htdocs/oc4jsamplecode/oc4j-demo-ejb.html` on the OTN site.

The following sections discuss how to implement data persistence:

- Creating BMP Entity Beans
- Component and Home Interfaces
- BMP Entity Bean Implementation
- Create Database Table and Columns for Entity Data

Creating BMP Entity Beans

As Chapter 3, "CMP Entity Beans" indicates, the steps for creating an entity bean are as follows:

1. Create the component interfaces for the bean. The component interfaces declare the methods that a client can invoke.
   a. The local component interface extends `javax.ejb.EJBLocalObject`.
   b. The remote component interface extends `javax.ejb.EJBObject`.
2. Create the home interfaces for the bean. The home interface defines the `create` and finder methods, including `findByPrimaryKey`, for your bean.
   a. The local home interface extends `javax.ejb.EJBLocalHome`.
   b. The remote home interface extends `javax.ejb.EJBHome`.
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 bean. This includes the following:
   a. The implementation for the methods that are declared in your component interfaces.
   b. The methods that are defined in the `javax.ejb.EntityBean` interface.
   c. The methods that match the methods that are declared in your home interface, which include the following:
      * The `ejbCreate` and `ejbPostCreate` methods with parameters matching the associated `create` method defined in the home interface.
      * Finder methods that are defined in the home interface. The `ejbFindByPrimaryKey` corresponds to the `findByPrimaryKey` method in the home interface. It retrieves the primary key and validates that it exists. Any other finder methods defined in the home interface must also be implemented in the bean implementation.
   d. The methods defined in the `javax.ejb.EntityBean` interface. The `ejbCreate`, `ejbPostCreate`, and `ejbFindByPrimaryKey` are already mentioned above. The other methods are as follows:
      * Persistent saving of the data within the `ejbStore` method.

* Restoring the persistent data to the bean within your implementation of the `ejbLoad` method.
* Passivation of the bean instance within the `ejbPassivate` method.
* Activation of the passivated bean instance within the `ejbActivate` method.

5. If the persistent data is saved to or restored from a database, you must ensure that the correct tables exist for the bean.
6. Create the bean deployment descriptor. The deployment descriptor specifies properties for the bean through XML elements.
7. Create an EJB JAR file containing the bean, component interface, home interface, and the deployment descriptors. Once created, configure the `application.xml` file, create an EAR file, and deploy the EJB to OC4J.

Component and Home Interfaces

The BMP entity bean definition of the component and home interfaces are identical to the CMP entity bean. For examples of how the component and home interfaces are implemented, see "Creating Entity Beans" on page 3-3.

BMP Entity Bean Implementation

Because the container is not managing the primary key nor the saving of the persistent data, the bean callback functions must include the implementation logic for these functions. The container invokes the `ejbCreate`, `ejbFindByPrimaryKey`, other finder methods, `ejbStore`, and `ejbLoad` methods where it is appropriate.

The ejbCreate Implementation

The `ejbCreate` method is responsible primarily for the creation of the primary key. This includes creating the primary key, creating the persistent data representation for the key, initializing the key to a unique value, and 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.

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
{
 this.empNo = empNo;
 this.empName = empName;
 this.salary = salary;
 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 "Modify XML Deployment Descriptors" on page 6-11.

```
<enterprise-beans>
   <entity>
      <display-name>EmployeeBean</display-name>
      <ejb-name>EmployeeBean</ejb-name>
      <local-home>employee.EmployeeHome</local-home>
      <local>employee.Employee</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;

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.EmployeeHome</local-home>
      <local>employee.Employee</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 be to 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.close();
  return pk;
}
```

The ejbFindByPrimaryKey Implementation

The `ejbFindByPrimaryKey` implementation is a requirement for all BMP entity beans. Its primary responsibility is to ensure that the primary key is valid. Once it is validated, it returns the primary key to the container, which uses the key to return the component interface 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 Integer ejbFindByPrimaryKey(Integer empNoPK)
    throws FinderException
{
  if (empNoPK == null) {
       throw new FinderException("Primary key cannot be null");
  }

  ps = conn.prepareStatement(SELECT EMPNO FROM EMPLOYEEBEAN
               WHERE EMPNO = ?);
  ps.setInt(1, empNoPK.intValue());
  ps.executeQuery();
  ResultSet rs = ps.getResultSet();
  if (rs.next()) {
   /*PK is validated because it exists already*/
  } else {
  throw new FinderException("Failed to select this PK");
 }

 ps.close();

 return empNoPK;
}
```

Other Finder Methods

You can create other finder methods beyond 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.

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
{
  Vector recs = new Vector();

  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;
}
```

The ejbStore Implementation

The container invokes the `ejbStore` method when the persistent data should be saved to the database. This includes whenever the primary key is "dirtied", or 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()
{
 //Container invokes this method to instruct the instance to
 //synchronize its state by storing it to the underlying database
 ps = conn.prepareStatement(UPDATE EMPLOYEEBEAN SET EMPNAME=?,
             SALARY=? WHERE EMPNO=?);
 ps.setString(1, this.empName);
 ps.setFloat(2, this.salary.floatValue());
 ps.setInt(3, this.empNo.intValue());
 if (ps.executeUpdate() != 1) {
```

```
        throw new EJBException("Failed to update record");
  }
  ps.close();
}
```

The ejbLoad Implementation

The container invokes the `ejbLoad` method after activating the bean instance. 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()
{
  //Container invokes this method to instruct the instance to
  //synchronize its state by loading it from the underlying database
  this.empNo = ctx.getPrimaryKey();
  ps = conn.prepareStatement(SELECT EMP_NO, EMP_NAME, SALARY WHERE EMPNAME=?");
  ps.setInt(1, this.empNo.intValue());
  ps.executeQuery();
  ResultSet rs = ps.getResultSet();
  if (rs.next()) {
    this.empNo = new Integer(rs.getInt(1));
    this.empName = new String(rs.getString(2));
    this.salary = new Float(rs.getFloat(3));
  } else {
    throw new FinderException("Failed to select this PK");
  }
  ps.close();
}
```

The ejbPassivate Implementation

The `ejbPassivate` method is invoked directly before the bean instance is serialized for future use. Normally, this is invoked when the instance has not been used in a while. 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();
}
```

The ejbActivate Implementation

As the `ejbPassivate` method section states, 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.

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);
}
```

The ejbRemove Implementation

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—both by the instance being destroyed and 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 befor it removes the EJB object
 //that is currently associated with the instance
 ps = conn.prepareStatement(DELETE FROM EMPLOYEEBEAN WHERE EMPNO=?);
 ps.setInt(1, this.empNo.intValue());
 if (ps.executeUpdate() != 1) {
       throw new RemoveException("Failed to delete record");
```

```
  }
 ps.close();
 conn.close();
}
```

Modify XML Deployment Descriptors

In addition to the configuration described in "Creating Entity Beans" on page 3-3, you must modify and add the following to your `ejb-jar.xml` deployment descriptor:

1. Configure the persistence type to be "`Bean`" in the `<persistence-type>` element.
2. Configure an resource reference for the database persistence storage in the `<resource-ref>` element.

   Our employee used the database environment element of "`jdbc/OracleDS`". This is configured in the `<resource-ref>` element as follows:

   ```
   <resource-ref>
    <res-ref-name>jdbc/OracleDS</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Application</res-auth>
   </resource-ref>
   ```

The database specified in the `<res-ref-name>` element maps to a `<ejb-location>` element in the `data-sources.xml` file. Our "`jdbc/OracleDS`" database is configured in the `data-sources.xml` file, as shown below:

```
<data-source
  class="com.evermind.sql.DriverManagerDataSource"
  name="Oracle"
  location="jdbc/OracleCoreDS"
  pooled-location="jdbc/pool/OraclePoolDS"
  ejb-location="jdbc/OracleDS"
  xa-location="jdbc/xa/OracleXADS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  url="jdbc:oracle:thin:@localhost:5521:orcl"
  username="scott"
  password="tiger"
  max-connections="300"
  min-connections="5"
  max-connect-attempts="10"
```

```
  connection-retry-interval="1"
  inactivity-timeout="30"
  wait-timeout="30"
/>
```

Create Database Table and Columns for Entity Data

If your entity bean stores its persistent data within a database, you need to create the appropriate table with the proper columns for the entity bean. This table must be created before the bean is loaded into the database. The container will not create this table for BMP beans, but it will create it automatically for CMP beans.

In our employee example, you must create the following table in the database defined in the `data-sources.xml` file:

Table	Columns
EMPLOYEEBEAN	■ employee number (EMPNO) ■ employee name (EMPNAME) ■ salary (SALARY)

The following shows the SQL commands that create these fields.

```
CREATE TABLE EMPLOYEEBEAN (
 EMPNO NUMBER NOT NULL,
 EMPNAME VARCHAR2(255) NOT NULL,
 SALARY FLOAT NOT NULL,
 CONSTRAINT EMPNO PRIMARY KEY
)
```

7

Message-Driven Beans

A Message-Driven Bean (MDB) is a Java Messaging Service (JMS) message listener that can reliably consume messages from a queue or a subscription of a topic. The advantage of using an MDB instead of a JMS message listener is that you can use the asynchronous nature of a JMS listener with the benefit of the EJB container performing the following:

- The consumer is created for the listener. That is, the appropriate `QueueReceiver` or `TopicSubscriber` is created by the container.
- The MDB is registered with the consumer. The container registers the MDB with the `QueueReceiver` or `TopicSubscriber` and its factory at deployment time.
- The message acknowledgment mode is specified.

An MDB is an easy method for creating a JMS message listener.

The following sections discuss the tasks in creating an MDB in Oracle9*i*AS Containers for J2EE (OC4J) and demonstrate MDB development with a basic configuration to use Oracle JMS as the resource provider.

- MDB Overview
- Creating MDBs
- Accessing MDBs

Download the MDB example from the OC4J sample code page at `http://otn.oracle.com/sample_code/tech/java/oc4j/htdocs/oc4jsamplecode/oc4j-demo-ejb.html` on the OTN web site.

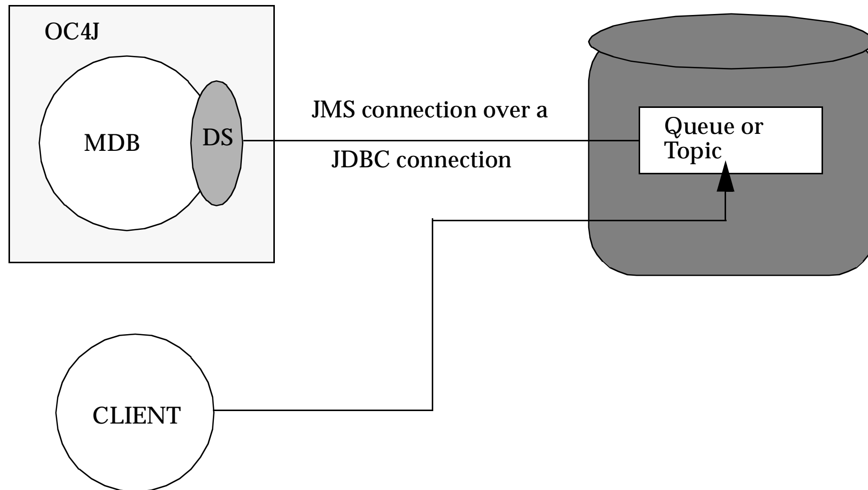
MDB Overview

An MDB is a unique EJB whose function is to read or write JMS messages from a JMS `Destination` (topic or queue).

The OC4J MDB interacts with Oracle JMS, which must be installed and configured appropriately. Oracle JMS is installed and configured on an Oracle database. Within this database, the appropriate queue or table is created.

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 resource 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 from the queue or topic. Other clients may have access to the same queue or topic to put on messages for the MDB.

Figure 7–1 Demonstration of an MDB Interacting with an Oracle JMS Destination



Creating MDBs

MDBs interact with queues and topics furnished by the Oracle JMS resource provider. A full description of how to use this resource provider is discussed in the JMS chapter in the *Oracle9iAS Containers for J2EE Services Guide.*

The JMS chapter details the following steps that enable each resource provider:

1. Install and configure the resource provider. For Oracle JMS, this includes the following:
   a. Create an RDBMS user through which the MDB connects to the database. Grant this user appropriate access privileges to perform Oracle JMS operations.
   b. Create the tables and queues to support the JMS `Destination` objects.
   c. Configure the Oracle JMS resource provider by configuring a data source with the capabilities that are appropriate for the functionality within your application.
2. Configure the location of the resource provider in the OC4J XML files.

To create an MDB that uses the resource provider, perform the following steps:

1. Implement the bean, which includes the following:
   a. The bean class must implement the `javax.ejb.MessageDrivenBean` and `javax.jms.MessageListener` interfaces, which includes the following:
      * the `onMessage` method in the `MessageListener` interface
      * the `setMessageDrivenContext` method in the `MessageDrivenBean` interface
   b. The bean class must implement the container callback methods that normally match methods in the EJB home interface. A remote, local, and home interface are not implemented with an MDB. However, some of the callback methods required for these interfaces are implemented in the bean implementation. These methods include the following:
      * an `ejbCreate` method
      * an `ejbRemove` method
2. Create the MDB deployment descriptors.

a. Define the JMS connection factory and `Destination` used in the EJB deployment descriptor. Define if any durable subscriptions are used.

b. Map the JMS connection factory and `Destination` type to the MDB in the OC4J-specific deployment descriptor—`orion-ejb-jar.xml`.

c. If the MDB is a container-managed transaction, specify the `onMessage` method in the `<container-transaction>` element.

3. Create an EJB JAR file containing the bean and the deployment descriptors. Configure the application-specific `application.xml` file, create an EAR file, and install the EJB in OC4J.

The following sections demonstrate a simple MDB, using Oracle JMS as the resource provider. For directions on configuring other resource providers, see the JMS chapter in the *Oracle9iAS Containers for J2EE Services Guide.*

- Install And Configure The Resource Provider
- Bean Class Implementation
- Configure Deployment Descriptors
- Deploy the Entity Bean

Install And Configure The Resource Provider

Before you can use the MDB within the application, you must choose and configure a resource provider for the JMS `Destination` objects used by the MDB. The following sections discuss how to configure Oracle JMS.

- Create User and Assign Privileges
- Create JMS Destination Objects
- Configure the DataSource
- Configure the Resource Provider

Create User and Assign Privileges

Create an RDBMS user through which the MDB connects to the database. Grant access privileges to this user to perform Oracle JMS operations. The privileges that you need depend on what functionality you are requesting. Refer to the *Oracle9i Application Developer's Guide - Advanced Queuing* for more information on privileges necessary for each type of function.

The following example creates MYUSER with privileges required for Oracle JMS operations:

```
create user MYUSER identified by MYPASSWORD;

grant connect, resource to MYUSER;

grant execute on sys.dbms_aqadm  to  MYUSER;
grant execute on sys.dbms_aq     to  MYUSER;
grant execute on sys.dbms_aqin   to  MYUSER;
grant execute on sys.dbms_aqjms  to  MYUSER;

connect MYUSER/MYPASSWORD;
```

You may need to grant other privileges, such as two-phase commit (requires FORCE ANY TRANSACTION) or system administration privileges, based on what the user needs.

Create JMS Destination Objects

Each resource provider requires its own method for creating the JMS Destination object. Refer to the *Oracle9i Application Developer's Guide - Advanced Queuing* for more information on the DBMS_AQADM packages and Oracle JMS messages types. For our example, Oracle JMS requires the following methods:

> **Note:** The SQL for creating the tables for the Oracle JMS example is included in the MDB example available from OTN.

1. Create the tables that handle the JMS Destination (queue or topic).

   In Oracle JMS, both topics and queues use a queue table. The Oracle JMS example within this chapter creates two tables: QTque for a queue and QTtpc for a topic.

   To create the queue table, execute the following SQL:

   ```
   DBMS_AQADM.CREATE_QUEUE_TABLE(
           Queue_table            => 'QTque',
           Queue_payload_type     => 'SYS.AQ$_JMS_BYTES_MESSAGE',
           multiple_consumers     =>  false);
   ```

   The type of message is defined as one of the following:

   - Bytes: SYS.AQ$_JMS_BYTES_MESSAGE

- Map: `SYS.AQ$_JMS_MAP_MESSAGE`
- String: `SYS.AQ$_JMS_STRING_MESSAGE`
- Text: `SYS.AQ$_JMS_TEXT_MESSAGE`
- Object: `SYS.AQ$_JMS_OBJECT_MESSAGE`

The third parameter denotes whether there are multiple consumers or not; thus, is always false for a queue and true for a topic.

To create the topic table, execute the following SQL:

```
DBMS_AQADM.CREATE_QUEUE_TABLE(
        Queue_table            => 'QTtpc',
        Queue_payload_type     => 'SYS.AQ$_JMS_BYTES_MESSAGE',
        multiple_consumers     => TRUE);
```

2. Create the JMS `Destination`. If you are creating a topic, you must add each subscriber for the topic. This example chooses to use durable subscribers, which results in additional configuration within the deployment descriptors.

   The Oracle JMS example within this chapter requires a single topic with two subscribers—`topic1` with `MDBSUB` and `MDBSUB2`—and a single queue—`queue1`.

   The following creates a topic called `topic1` within the topic table `QTtpc` with max retries set to 2. After creation, two durable subscribers are added to the topic. Finally, the topic is started.

```
DBMS_AQADM.CREATE_QUEUE( 'topic1', 'QTtpc');
DBMS_AQADM.ADD_SUBSCRIBER('topic1', sys.aq$_agent('MDSUB', null, null));
DBMS_AQADM.ADD_SUBSCRIBER('topic1', sys.aq$_agent('MDSUB2', null, null));
DBMS_AQADM.START_QUEUE('topic1');
```

   The following creates a queue called `queue1` within the queue table `QTque` with max retries set to 2. After creation, the queue is started.

```
DBMS_AQADM.CREATE_QUEUE( 'queue1' , 'QTque');
DBMS_AQADM.START_QUEUE('queue1');
```

Configure the DataSource

Configure the Oracle JMS resource provider by configuring a data source. The topics and queues connect to the database and 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 7–1 Non-Emulated With Thin JDBC Driver

The following example contains a non-emulated data source that uses the thin JDBC driver. It can support a two-phase commit transaction; it cannot support session pooling.

The example is displayed in the format of an XML definition; see the *Oracle9iAS Containers for J2EE User's Guide* for directions on adding a new data source to the configuration.

```
<data-source
  class="com.evermind.sql.OrionCMTDataSource"
  name="myDS"
  location="jdbc/MyDS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="myuser"
  password="mypasswd"
  url="jdbc:oracle:thin:@myhost.foo.com:1521:mydb"
  inactivity-timeout="30"
/>
```

Configure the Resource Provider

Identify the JNDI name of the data source that is to be used as the resource provider within the `<resource-provider>` element.

- If this is to be the resource provider for all applications (global), configure the global `application.xml` file.
- If this is to be the resource provider for a single application (local), configure the `orion-application.xml` file of the application.

The following code sample shows how to configure the resource provider using XML syntax for Oracle JMS.

- `class` attribute—The Oracle JMS resource 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 resource provider in the `property` element. The topic or queue connects to this data source to access the tables and queues that facilitate the messaging.

The following example demonstrates that the data source identified by "jdbc/CartEmulatedDS" is to be used as the Oracle JMS resource provider. This JNDI name is identified in the ejb-location element in Example 7–1.

```
<resource-provider class="oracle.jms.OjmsContext" name="cartojms1">
  <description> OJMS/AQ </description>
  <property name="datasource" value="jdbc/myDS"></property>
</resource-provider>
```

Bean Class Implementation

Most MDBs receive messages from a queue or a topic, then invoke an entity bean to process the request contained within the message.

The following example is a MessageBean MDB. Its functionality is to print out a message sent to it through a durable topic. The topic is identified in the deployment descriptors.

As an MDB, it is responsible for the following:

- implements the javax.ejb.MessageDrivenBean and javax.jms.MessageListener interfaces
- defined as public (not final or abstract)
- implements a constructor and the following methods: setMessageDrivenContext, ejbCreate, onMessage, and ejbRemove

```
package cart.ejb;

import com.evermind.server.ThreadState;

import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.MessageDrivenBean;
import javax.ejb.MessageDrivenContext;
import javax.ejb.CreateException;
import javax.naming.*;
import javax.transaction.*;

import javax.jms.*;
import  oracle.AQ.*;
import  oracle.jms.*;

 public class MessageBean implements javax.ejb.MessageDrivenBean,
    javax.jms.MessageListener
 {
```

```
private transient MessageDrivenContext  mdbCtx = null;

/* Constructor, which is public and takes no arguments.*/
public MessageBean() { }

/* setMessageDrivenContext method */
public void setMessageDrivenContext(MessageDrivenContext mdc)
{
 /* As with all EJBs, you must set the context in order to be
    able to use it at another time within the MDB methods. */
 this.mdbCtx = mdc;
}

/* ejbCreate method, declared as public (but not final or
 * static), with a return type of void, and with no arguments.
 */
public void ejbCreate() throws  Exception
{
 /* no implementation is necessary for this MDB */
 /* An MDB does not carry state for an individual client. However, you can
    retrieve state for use across many calls for multiple clients - state
    such as an entity bean reference or a database connection. If so,
    retrieve these within the ejbCreate and remove them in the
    ejbRemove method. */
}

/* ejbRemove method */
public void ejbRemove()
{
 /* no implementation is necessary for this MDB*/
}

/**
 * onMessage method
 * Casts the incoming Message to a TextMessage and displays
 * the text.
 */
 public void onMessage(Message msg)
{
 /* The whole point for this message MDB is to receive and print
    messages. It is not complicated, but it shows how MDBs are set up to
    receive JMS messages from queues and topics. */

  BytesMessage     msgBytes = null;
```

```
        try
        {
          /* This message was created as a JMS BytesMessage. */
          if (msg instanceof BytesMessage)
          {
           /* Convert the BytesMessage into printable text... */
            msgBytes = (BytesMessage) msg;
            byte[]  msgdata = ((AQjmsBytesMessage) msgBytes).getBytesData();
            String  txt = new  String(msgdata);
           /* Print out message */
            System.out.println("Message received=" + txt);
          }
        }
        catch (Exception e)
        {
          throw  new  RuntimeException("onMessage throws exception");
        }
      }
}
```

Configure Deployment Descriptors

The deployment descriptors define MDB configuration in the `<message-driven>` element.

- The EJB deployment descriptor (`ejb-jar.xml`) specifies whether a queue or a topic is used. This example uses a durable topic.
- The OC4J-specific deployment descriptor (`orion-ejb-jar.xml`) associates the queue or topic with the actual JMS `Destination` created in the resource provider.

EJB Deployment Descriptor for the MDB

Within the EJB deployment descriptor (`ejb-jar.xml`), 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 must also define whether it is durable.

The following example demonstrates the deployment information for the `MessageBean` MDB in the `<message-driven>` element, as follows:

- MDB name specified in the `<ejb-name>` element.
- MDB class defined in the `<ejb-class>` element.

- JMS `Destination` type is a `Topic` that is specified in the `<message-driven-destination><jms-destination-type>` element.
- The topic is durable, which is specified in the `<message-driven-destination><subscription-durability>` element. Options are "`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.

ejb-jar.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>
     <description>A demo cart bean package.</description>
     <display-name>A simple cart jar</display-name>

     <enterprise-beans>
     ...
       <message-driven>
         <description></description>
         <display-name>MessageBeanTpc</display-name>
         <ejb-name>MessageBeanTpc</ejb-name>
         <ejb-class>cart.ejb.MessageBean</ejb-class>
         <transaction-type>Container</transaction-type>
         <message-driven-destination>
            <destination-type>javax.jms.Topic</destination-type>
            <subscription-durability>Durable</subscription-durability>
         </message-driven-destination>
       </message-driven>
     ...
       <assembly-descriptor>
          <container-transaction>
              <method>
                 <ejb-name>MessageBeanTpc</ejb-name>
                 <method-name>onMessage</method-name>
              </method>
              <trans-attribute>Required</trans-attribute>
           </container-transaction>
       ...
```

```
  </assembly-descriptor>
</enterprise-beans>
```

OC4J-Specific Deployment Descriptor

Once you have configured the MDB and the JMS `Destination` type, inform the container which JMS `Destination` to associate with the MDB. To identify the `Destination` that is to be associated with the MDB, map the `Destination` location and connection factory to the MDB through the `<message-driven-deployment>` element in the `orion-ejb-jar.xml` file.

The following is the `orion-ejb-jar.xml` deployment descriptor for the `MessageBean` example. It maps an Oracle JMS `Topic` to the `MessageBean` MDB, providing the following:

- MDB name, as defined in the `<message-driven><ejb-name>` in the EJB deployment descriptor, is specified in the `name` attribute.
- JMS `Destination Connection Factory` is specified in the `connection-factory-location` 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. For this example, the resource provider name is defined in the `<resource-provider>` element in the `application.xml` file as `cartojms1` and the user defined name is `aqTcf`.
- 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.

  For this example, the resource provider name is defined in the `<resource-provider>` element in the `application.xml` file as `cartojms1`. In this example, the topic name is `topic1`.
- Because this is a topic, the subscription name is defined in the `Csubscription-name` attribute. Two subscriptions were created from the SQL in the database: `MDBSUB` and `MDBSUB2`. This topic uses the `MDBSUB` subscription.
- Listener threads, as defined in the `listener-threads` attribute. The listener threads are spawned off when MDBs are deployed and are used to listen for

incoming JMS messages on the topic or queue. These threads concurrently consume JMS messages. The default is one thread.

- transaction timeout, as defined in the `transaction-timeout` attribute. This attribute controls the transaction timeout interval for any container-managed transactional MDB. The default is one day. If the transaction has not completed in this timeframe, the transaction is rolled back.

After you specify all of these in the `<message-driven-deployment>` element, the container knows how to map the MDB to the correct JMS `Destination`.

```
<enterprise-beans>
  <message-driven-deployment
      connection-factory-location=
          "java:comp/resource/cartojms1/TopicConnectionFactories/aqTcf"
      name="MessageBeanTpc"
      destination-location="java:comp/resource/cartojms1/Topics/topic1"
      subscription-name="MDBSUB"
      listener-threads=50 transaction-timeout=172800>
  ...
</enterprise-beans>
```

Deploy the Entity Bean

Archive your EJB into a JAR file. You deploy the MDB in the same way as the session bean, which "Prepare the EJB Application for Assembly" on page 2-13 and "Deploy the Enterprise Application to OC4J" on page 2-15 describe.

Accessing MDBs

The client sends a message to the MDB through a JMS `Destination`. The MDB is associated with the JMS `Destination` by the container.

To send a JMS message to an MDB, perform the following:

1. Retrieve both the configured JMS `Destination` and its connection factory using a JNDI lookup.
2. Create a connection from the connection factory. For a queue, start the connection.
3. Create a session over the connection.
4. Providing the retrieved JMS `Destination`, create a sender for a queue, or a publisher for a topic.

5. Create the message.
6. Send out the message using either the queue sender or the topic publisher.
7. Close the queue session. Close the connection for either JMS `Destination` types.

The following code sends a message over a topic to the `MessageBean` MDB.

```
Context ic = new InitialContext();
/*1. Retrieve an Oracle JMS Topic and connection factory through JNDI */
topic = (Topic) ic.lookup("java:comp/resource/ojms/Topics/topic1");
/*Retrieve the Oracle JMS Topic connection factory */
topicConnectionFactory = (TopicConnectionFactory) ic.lookup
                ("java:comp/resource/ojms/TopicConnectionFactories/aqTcf");

/*2. Create a Topic connection */
topicConnection = topicConnectionFactory.createTopicConnection();

/*3. Create a Topic session over the connection */
topicSession = topicConnection.createTopicSession(true,
                                                  Session.AUTO_ACKNOWLEDGE);

/*4. Create a publisher to send a message to the MDB -- The createPublisher
   method is invoked off of the session object, but requires the retrieved
   topic as its input. */
topicPublisher = topicSession.createPublisher(topic);

/*5. Create the message to send to the MDB */
message = topicSession.createTextMessage();

for (int i = 0; i < NUM_MSGS; i++) {
    message.setText("This is message " + (i + 1));
    System.out.println("Sending message: " +
    message.getText());
/*6. Send the message using the topic publisher */
    topicPublisher.publish(message);
}

/*7. After message is sent, close the connection */
topicConnection.close();
```

Using Logical Names in the JMS JNDI Lookup

If you have another EJB acting as the MDB client, you can retrieve the connection factory and the JMS `Destination` using logical references that have been configured in the client-side deployment descriptor.

The following defines logical names in the client-side deployment descriptor. If the client is a true Java client, this would be in its `application-client.xml` file. If the client is another EJB, these additions would be added in the `ejb-jar.xml` file.

- The connection factory is defined in a `<resource-ref>` element.
  - The logical name of the connection factory is defined in the `<res-ref-name>` element.
  - The class type is defined in the `<res-type>` element: `javax.jms.QueueConnectionFactory` or `javax.jms.TopicConnectionFactory`.
  - The authentication responsibility (`Container` or `Bean`) is defined in the `<res-auth>` element.
  - The sharing scope (`Shareable` or `Unshareable`) is defined in the `<res-sharing-scope>` element.
- The JMS `Destination` is defined in a `<resource-env-ref>` element.
  - The logical name of the topic or queue is defined in the `<resource-env-ref-name>` element.
  - The class type is defined in the `<resource-env-ref-type>` element: `javax.jms.Queue` or `javax.jms.Topic`.

The following shows an example of how to define a queue and a topic.

```
<resource-ref>
  <res-ref-name>jms/Queue/senderQueueConnectionFactory</res-ref-name>
  <res-type>javax.jms.QueueConnectionFactory</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
<resource-env-ref>
  <resource-env-ref-name>jms/Queue/senderQueue</resource-env-ref-name>
  <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
</resource-env-ref>
<resource-ref>
  <res-ref-name>jms/Topic/senderTopicConnectionFactory</res-ref-name>
  <res-type>javax.jms.TopicConnectionFactory</res-type>
  <res-auth>Container</res-auth>
```

```
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
<resource-env-ref>
  <resource-env-ref-name>jms/Topic/senderTopic</resource-env-ref-name>
  <resource-env-ref-type>javax.jms.Topic</resource-env-ref-type>
</resource-env-ref>
```

Then, you map the logical names to actual names in the OC4J deployment descriptor. If the client is a true Java client, these additions would be within the `orion-application-client.xml`. If the client is another EJB, these additions are added to the `orion-ejb-jar.xml`.

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

- The connection factory defined in the `<resource-ref>` element is mapped to its JNDI name in the `<resource-ref-mapping>` element.
- The JMS `Destination` defined in the `<resource-env-ref>` element is mapped to its JNDI name in the `<resource-env-ref-mapping>` element.

The following example maps the logical names for the connection factories, topic and queue defined above in the client deployment descriptor to their actual JNDI names. Specifically, the topic defined logically as "`jms/Topic/senderTopic`" in the `ejb-jar.xml` file is mapped to its JNDI name of "`java:comp/resource/cartojms1/Topics/topic1`."

```
<session-deployment name="MyCart" max-instances="10" location="MyCart">

<resource-ref-mapping
        name="jms/Topic/senderTopicConnectionFactory"
       location="java:comp/resource/cartojms1/TopicConnectionFactories/aqTcf">
</resource-ref-mapping>

<resource-env-ref-mapping
       name="jms/Topic/senderTopic"
       location="java:comp/resource/cartojms1/Topics/topic1">
</resource-env-ref-mapping>

<resource-ref-mapping
       name="jms/Queue/senderQueueConnectionFactory"
       location="java:comp/resource/cartojms1/QueueConnectionFactories/aqQcf">
</resource-ref-mapping>

<resource-env-ref-mapping
       name="jms/Queue/senderQueue"
       location="java:comp/resource/cartojms1/Queues/queue1">
```

```
</resource-env-ref-mapping>

</session-deployment>
```

Once the mapping is complete, you can modify your JNDI lookup to use the logical name, as follows:

```
Context ic = new InitialContext();
/*Retrieve an Oracle JMS Topic and connection factory through JNDI */
topic = (Topic) ic.lookup("jms/Topic/senderTopic");
/*Retrieve the Oracle JMS Topic connection factory */
topicConnectionFactory = (TopicConnectionFactory) ic.lookup
                ("jms/Topic/senderTopicConnectionFactory");
```

8 Advanced EJB Subjects

This chapter discusses how to extend beyond the basics mentioned in each of the previous chapters. This chapter covers the following subjects:

- Accessing EJBs
- Packaging and Sharing Classes
- Entity Bean Concurrency and Database Isolation Modes
- Configuring Pool Sizes For Entity Beans
- Techniques for Updating Persistence
- Configuring Environment References
- Configuring Security
- Setting Performance Options
- Common Errors

Accessing EJBs

To access an EJB from a client, you must do the following:

1. Download the `oc4j.jar` file. This JAR contains only the classes necessary for client interaction.
2. Set up JNDI properties for the connection. If the client is a standalone EJB client, you must determine the RMI or JMS port to which the client sends the request. This is denoted in the JNDI properties as well as the `opmn.xml` file.
3. Determine which `InitialContext` you will use for the connection.
4. Retrieve an EJB using an EJB reference, which is configured in the deployment descriptor.

These subjects are discussed in the following sections:

- Client Installation of OC4J.JAR
- EJB Reference Information
- Setting JNDI Properties
- Using the Initial Context Factory Classes
- Accessing an EJB in a Remote Server

Within your client code, you retrieve an EJB reference to the target bean in order to execute methods on that bean. In OC4J, you use JNDI to retrieve this reference. Most of the time, you must specify the target bean in an `<ejb-ref>` element in the originator's XML configuration file that is used in the `java:comp/env` logical name to designate the target bean to JNDI.

The method for accessing EJBs depends on where your client is located relative to the bean it wants to invoke. Consider the following when implementing the JNDI retrieval of the EJB reference of the bean:

1. Do you want to set up a logical name for the target bean?
   - Yes: Modify the XML configuration file to set up the `<ejb-ref>` element with the target bean information. The logical name specified in the `<ejb-ref-name>` element is used in the JNDI lookup.
   - No: The actual name of the bean is used in the JNDI lookup. This name has been specified in the target bean's XML deployment descriptors in the `<ejb-name>` element.
2. Where does the client exist relative to the target bean?

– Collocated with the target bean? Deployed in the same application? Or is the target bean part of an application that is this client's parent? You do not need to set up any JNDI properties.

– Otherwise, you must set up JNDI properties. There are two methods for setting up JNDI properties. See "Setting JNDI Properties" on page 8-4 for more information

Client Installation of OC4J.JAR

In order to access EJBs, the client-side must download the `oc4j.jar` file. This JAR contains only the classes necessary for client interaction. If you download this JAR into a browser, you must grant certain permissions. See <<<>>> for a list of these permissions.

EJB Reference Information

Specify the EJB reference information for the remote EJB in the `<ejb-ref>` element in the `application-client.xml`, `ejb-jar.xml`, or `web.xml` files. A full description or how to set up the `<ejb-ref>` element is given in "Configuring Environment References" on page 8-15.

For example, the following specifies the reference information for the employee example:

```
<application-client>
        <display-name>EmployeeBean</display-name>
        <ejb-ref>
                <ejb-ref-name>EmployeeBean</ejb-ref-name>
                <ejb-ref-type>Session</ejb-ref-type>
                <home>employee.EmployeeHome</home>
                <remote>employee.Employee</remote>
                <local-home>employee.EmployeeLocalHome</local-home>
                <local>employee.EmployeeLocal</local>
        </ejb-ref>
</application-client>
```

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.

For more information and examples of the `<ejb-ref>` element, see "Configuring Environment References" on page 8-15.

Setting JNDI Properties

If the client is collocated with the target, exists within the same application as the target, or the target exists within its parent, then you do not need a JNDI properties file. If not, you must initialize your JNDI properties either within a `jndi.properties` file, in the system properties, or within your implementation, before the JNDI call. The following sections discuss these three options:

- No JNDI Properties
- JNDI Properties File
- JNDI Properties Within Implementation

To specify credentials within the JNDI properties, see "Specifying Credentials in EJB Clients" on page 8-36.

No JNDI Properties

A servlet that is collocated with the target bean automatically accesses the JNDI properties for the node. Thus, accessing the EJB is simple: no JNDI properties are required.

```
//Get the Initial Context for the JNDI lookup for a local EJB
InitialContext ic = new InitialContext();
//Retrieve the Home interface using JNDI lookup
Object empObject = ic.lookup("java:comp/env/employeeBean");
```

This is also true if the target bean is in the same application or an application that has been deployed as this application's parent. See "The default parent is the global 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 Oracle9iAS Containers for J2EE User's Guide for directions on how to specify a parent application." on page 8-9 for more information on setting the parent application.

JNDI Properties File

If setting the JNDI properties within the `jndi.properties` file, set the properties as follows. Make sure that this file is accessible from the `CLASSPATH`.

Factory

```
java.naming.factory.initial=
        com.evermind.server.ApplicationClientInitialContextFactory
```

Location

The ORMI default port number is 23791, which can be modified in `rmi.xml`. Thus, set the URL in the `jndi.properties`, in one of the two ways:

```
java.naming.provider.url=ormi://<hostname>/<application-name>
```

or

```
java.naming.provider.url=ormi://<hostname>:23791/<application-name>
```

Security

When you access EJBs in a *remote* container, you must pass valid credentials to this container. Stand-alone clients define their credentials in the `jndi.properties` file deployed with the client's code.

```
java.naming.security.principal=<username>
java.naming.security.credentials=<password>
```

JNDI Properties Within Implementation

Set the properties with the same values, only with different syntax. For example, JavaBeans running within the container pass their credentials within the `InitialContext`, which is created to look up the remote EJBs.

To pass JNDI properties within the `Hashtable` environment, set these as shown below:

```
Hashtable env = new Hashtable();
env.put("java.naming.provider.url", "ormi://myhost/ejbsamples");
env.put("java.naming.factory.initial",
      "com.evermind.server.ApplicationClientInitialContextFactory");
env.put(Context.SECURITY_PRINCIPAL, "guest");
env.put(Context.SECURITY_CREDENTIALS, "welcome");
Context ic = new InitialContext (env);
Object homeObject = ic.lookup("java:comp/env/employeeBean");

// Narrow the reference to a TemplateHome.
EmployeeHome empHome =
        (EmployeeHome) PortableRemoteObject.narrow(homeObject,
                                                    EmployeeHome.class);
```

Configuring RMI or JMS Port for Standalone EJB Clients

OC4J is configured to assign an RMI or JMS port dynamically within set ranges. However, if you have a standalone EJB client, you must know an exact port number to direct your request.

1. Pick a port number that is not being used by the OC4J process.
2. Modify the `opmn.xml` file within the Enterprise Manager Advanced Properties within the OPMN configuration. Change the RMI or JMS range to the specified port number. The following demonstrates setting the RMI port to 3202 in the `opmn.xml` file:

   ```
   <port ajp="..." jms="..." rmi="3202"/>
   ```

3. Restart the OC4J process to initialize the new port numbers.
4. Configure the same port number within the JNDI properties within the standalone client. The following demonstrates setting the same RMI port number in the JNDI properties for the EJB client:

   ```
   java.naming.provider.url=ormi://myhost:3202/myapp
   ```

Using the Initial Context Factory Classes

For most clients, set the initial context factory class to `ApplicationClientInitialContextFactory`. If you are not using a logical name defined in the `<ejb-ref>` in your XML configuration file, then you must provide the actual JNDI name of the target bean. In this instance, you must use a different initial context factory class, the `com.evermind.server.RMIInitialContextFactory` class.

Example 8–1 Servlet Accessing EJB in Remote OC4J Instance

The following servlet uses the JNDI name for the target bean: `/cmpapp/employeeBean`. Thus, this servlet may provide the JNDI properties in an `RMIInitialContext` object, instead of the `ApplicationClientInitialContext` object. The environment is initialized as follows:

- The `INITIAL_CONTEXT_FACTORY` is initialized to a `RMIInitialContextFactory`.
- Instead of creating a new `InitialContext`, it is retrieved. ’
- The actual JNDI name is used in the lookup.

```
Hashtable env = new Hashtable();
env.put(Context.PROVIDER_URL, "ormi://localhost/cmpapp");
env.put(Context.SECURITY_PRINCIPAL, "admin");
env.put(Context.SECURITY_CREDENTIALS, "welcome");
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.evermind.server.rmi.RMIInitialContextFactory");

Context ic =
    new com.evermind.server.rmi.RMIInitialContextFactory().
        getInitialContext(env);

Object homeObject = ic.lookup("/cmpapp/employeeBean");

// Narrow the reference to a TemplateHome.
EmployeeHome empHome =
     (EmployeeHome) PortableRemoteObject.narrow(homeObject,
                                                EmployeeHome.class);
```

An Initial Context Factory Specific to DNS Load Balancing

To use round-robin DNS for your incoming load balancing, you must do the following:

1. Within DNS, map a single host name to several IP addresses. Each of the port numbers must be the same for each IP address. Then, the incoming calls are randomly routed to one of the back-end machines.
2. Within each client, use the `RMILBInitialContextFactory` as your initial context. Each client must use this initial context for DNS round-robin load balancing to work properly.

When you perform a successful host name lookup from the name server, the value is cached. DNS load balancing does not occur if every lookup returns the same value from the cache. When you use the `RMILBInitialContextFactory` in the client, then a new context class is returned on each lookup.

Example 8–2 RMILBInitialContextFactory Example

```
java.naming.factory.initial=
           com.evermind.server.rmi.RMILBInitialContextFactory
java.naming.provider.url=ormi://DNSserver:23792/applname
java.naming.security.principal=admin
java.naming.security.credentials=welcome
```

```
dedicated.rmicontext=true
```

Accessing an EJB in a Remote Server

If an application is installed in the OC4J server with a JSP or servlet that wants to invoke an EJB in a remote server, do the following:

1. Deploy the intended EJB with the JSP/servlet in the same application.
2. Set "remote=true" attribute in the <ejb-module> element in orion-application.xml for the EJB module deployed in the local application. The local EJB will be ignored.
3. Configure the remote server where the remote EJB has been deployed in the <server> element in rmi.xml. You provide the hostname, port number, username, and password, as follows:

```
<server host=<remote_host> port=<remote_port> user=<username>
password=<password>
```

If multiple servers are configured, the OC4J container will search all remote servers for the intended EJB application. Thus, the JSP or servlet in one OC4J container will invoke an EJB deployed in another OC4J container.

Packaging and Sharing Classes

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.

Web application: The Web application copies referenced EJB classes in its WAR file in the same application. As described in the *Oracle9iAS Containers for J2EE Servlet Developer's Guide,* you can specify where to load the classes from in the <web-app-class-loader> element. However, when executing, the Web application retrieves the EJB bean reference using JNDI, the following may occur:

- If the bean reference is loaded from within the EJB JAR file, no exceptions are thrown and the execution continues correctly.
- If the bean reference is loaded from the copied classes within the WAR file, a ClassCastException is thrown.

Separate application: A separate application copies referenced EJB classes into its EAR file. If both applications are executing within the same process and the bean reference is retrieved using JNDI lookup, one of the following occurs:

- If the bean reference is loaded from the source classes, no exceptions are thrown and the execution continues correctly.
- If the bean reference is loaded from the copied classes, a `ClassCastException` is thrown.

To avoid this problem, do one of the following:

- If two EJBs use the same classes, include these classes in one of the EJBs. Place both 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 it 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. You can set shared library directories in the General Properties page of the default application. The `home/lib` is a default shared library.
- 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 is the global 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 *Oracle9iAS Containers for J2EE User's Guide* for directions on how to specify a parent application.

Entity Bean Concurrency and Database Isolation Modes

In order to avoid resource contention and overwriting each others changes to database tables while allowing concurrent execution, entity bean concurrency and database isolation modes are provided.

- Database Isolation Modes
- Entity Bean Concurrency Modes

Database Isolation Modes

The `java.sql.Connection` object represents a connection to a specific database. Database isolation modes are provided to define protection against resource contention. When two or more users try to update the same resource, a lost update can occur. That is, one user can overwrite the other user's data without realizing it. The `java.sql.Connection` standard provides four isolation modes, of which Oracle only supports two of these modes. These are as follows:

- `TRANSACTION_READ_COMMITTED`: Dirty reads are prevented; non-repeatable reads and phantom reads can occur. This level only prohibits a transaction from reading a row with uncommitted changes in it.
- `TRANSACTION_SERIALIZABLE`: Dirty reads, non-repeatable reads and phantom reads are prevented. This level includes the prohibitions in `TRANSACTION_REPEATABLE_READ` and further prohibits the situation where one transaction reads all rows that satisfy a WHERE condition, a second transaction inserts a row that satisfies that WHERE condition, and the first transaction rereads for the same condition, retrieving the additional "phantom" row in the second read.

You can configure one of these database isolation modes for a specific bean. That is, you can specify that when the bean starts a transaction, the database isolation mode for this bean be what is specified in the OC4J-specific deployment descriptor. Specify the isolation mode on what is important for the bean: parallel execution or data consistency. The isolation mode for this bean is set for the entire transaction.

The isolation mode can be set for each entity bean in the `<entity-deployment>` element in the `isolation` attribute. The values can be `committed` or `serializable`. The default is `committed`. To change it to `serializable`, configure the following in the `orion-ejb-jar.xml` for the intended bean:

```
<entity-deployment ...  isolation="serializable"
 ...
</entity-deployment>
```

There is always a trade-off between performance and data consistency. The `serializable` isolation mode provides data consistency; the `committed` isolation mode provides for parallel execution.

Note: There is a danger of lost updates with the `serializable` mode if the `max-tx-retries` element in the OC4J-specific deployment descriptor is greater than zero. The default for this value is three. If this element is set to greater than zero, then the container retries the update if a second blocked client receives a `ORA-8177` exception. The retry would find the row unlocked and the update would occur. Thus, the second client's update succeeds and overwrites the first client's update. If you use `serializable`, you should consider setting the `max-tx-retries` element to zero for data consistency.

If you do not set an isolation mode, you receive the mode that is configured in the database. Setting the isolation mode within the OC4J-specific deployment descriptor temporarily overrides the database configured isolation mode for the life of the global transaction for this bean. That is, if you define the bean to use the `serializable` mode, then the OC4J container will force the database to be `serializable` for this bean only until the end of the transaction.

Entity Bean Concurrency Modes

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. The concurrency modes configure when to block to manage resource contention or when to execute in parallel.

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.

- `READ-ONLY`: Multiple users can execute the entity bean in parallel. The container does not allow any updates to the bean's state.

To enable the CMP entity bean concurrency mode, add the appropriate concurrency value of "`pessimistic`", "`optimistic`", or "`read-only`" to the `locking-mode` attribute of the `<entity-deployment>` element in the OC4J-specific deployment descriptor (`orion-ejb-jar.xml`). The default is "`optimistic`". To modify the concurrency mode to `pessimistic`, do the following:

```
<entity-deployment ...  locking-mode="pessimistic"
 ...
</entity-deployment>
```

These concurrency modes are defined per bean and the effects of locking apply on the transaction boundaries.

Parallel execution requires that the pool size for wrapper and bean instances are set correctly. For more information on how to configure the pool sizes, see "Configuring Pool Sizes For Entity Beans" on page 8-13.

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.

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

Effects of the Combination of Isolation and Concurrency Modes

For the `pessimistic` and `read-only` concurrency modes, the setting of the database isolation mode does not matter. These isolation modes only matter if an external source is modifying the database.

If you choose `optimistic` with `committed`, you have the potential to lose an update. If you choose `optimistic` with `serializable`, you will never lose an update. Thus, your data will always be consistent. However, you can receive an `ORA-8177` exception as a resource contention error.

Differences Between Pessimistic and Optimistic/Serializable

An entity bean with the `pessimistic` concurrency mode does not allow multiple clients to execute the bean instance. Only one client is allowed to execute the instance at any one moment. An entity bean with the `optimistic` concurrency mode allows multiple instances of the bean implementation to execute in parallel. Setting the database isolation mode to `serializable` does not allow these multiple bean implementation instances to update the same row at the same time. Thus, the only difference between a `pessimistic` concurrency bean and an `optimistic/serializable` bean is where the blocking occurs. A `pessimistic` bean blocks at the bean instance; the other blocks at the database row.

Affects of Concurrency Modes on Clustering

All concurrency modes behave in a similar manner whether they are used within a standalone or a clustered environment. This is because the concurrency modes are locked at the database level. Thus, even if a pessimistic bean instance is clustered across nodes, the instant one instance tries to execute, the database locks out all other instances.

Configuring Pool Sizes For Entity Beans

You can set the minimum and maximum number of both the following instance pools:

- The bean instance pool contains EJB implementation instances that currently do not have assigned state. While the bean instance is in pool state, it is generic and can be assigned to a wrapper instance.
- The wrapper instance is OC4J-generated wrapper code that provides for the services requested in the deployment descriptor. Before the bean instance is

invoked, the client retrieves a handle to the wrapper instance. When the client invokes the bean, the wrapper is associated with a bean instance.

You can set the pool number of each instance type with the following attributes of the `<entity-deployment>` element.

- The `max-instances` attribute sets the maximum entity bean instances to be allowed in the pool. An entity bean is set to a pooled state if not associated with a wrapper instance. Thus, it is generic.

  The default is 10. Set the maximum bean implementation instances as follows:

  ```
  <entity-deployment ...  max-instances="20"
   ...
  </entity-deployment>
  ```

  Or the minimum number allowed in the pool as follows:

  ```
  <entity-deployment ... min-instances="2"
   ...
  </entity-deployment>
  ```

- The `disable-wrapper-cache` attribute disables the wrapper instance pool if true. The default is true. If it is better to create the wrapper instances on demand, then set this attribute to true. To do so, configure the following:

  ```
  <entity-deployment ...  disable-wrapper-cache="true"
   ...
  </entity-deployment>
  ```

Techniques for Updating Persistence

By default, the container persists only the modified fields in the bean. At the end of each call, a SQL command is created to update these fields. However, if you want to have all of your persistence fields updated, set the following attribute to false:

```
<entity-deployment ...  update-changed-fields-only="false"
 ...
</entity-deployment>
```

If you choose to have all fields updated, the SQL parsing cache is used. The same SQL command is used for each update.

Configuring Environment References

You can create three types of environment elements that are accessible to your bean during runtime: environment variables, EJB references, and resource managers. These environment elements are static and can not be changed by the bean.

ISVs typically develop EJBs that are independent from the EJB container. In order to distance the bean implementation from the container specifics, you can create environment elements that map to one of the following: defined variables, entity beans, or resource managers. This indirection enables the bean developer to refer to existing variables, EJBs, and a JDBC `DataSource` without specifying the actual name. These names are defined in the deployment descriptor and are linked to the actual names within the OC4J-specific deployment descriptor.

Environment variables

You can create environment variables that your bean accesses through a lookup on the `InitialContext`. These variables are defined within an `<env-entry>` element and can be of the following types: `String`, `Integer`, `Boolean`, `Double`, `Byte`, `Short`, `Long`, and `Float`. The name of the environment variable is defined within `<env-entry-name>`, the type is defined in `<env-entry-type>`, and its initialized value is defined in `<env-entry-value>`. The `<env-entry-name>` is relative to the `"java:comp/env"` context.

For example, the following two environment variables are declared within the XML deployment descriptor for `java:comp/env/minBalance` and `java:comp/env/maxCreditBalance`.

```
<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>
```

Within the bean's code, you would access these environment variables through the `InitialContext`, as follows:

```
InitialContext ic = new InitialContext();
```

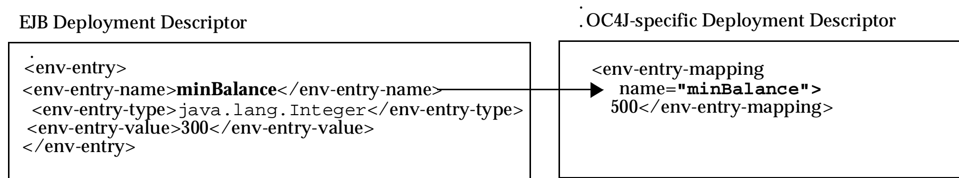
```
Integer min = (Integer) ic.lookup("java:comp/env/minBalance");
Integer max = (Integer) ic.lookup("java:comp/env/maxCreditBalance"));
```

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.

If you wanted the value of the environment variable to be defined in the OC4J-specific deployment descriptor, you can map the `<env-entry-name>` to the `<env-entry-mapping>` element in the OC4J-specific deployment descriptor. This means that the value specified in the `orion-ejb-jar.xml` file overrides any value that may be specified in the `ejb-jar.xml` file. The type specified in the EJB deployment descriptor stays the same.

Figure 8–1 shows how the `minBalance` environment variable is defined as 500 within the OC4J-specific deployment descriptor.

Figure 8–1 Environment Variable Mapping



Environment References To Other Enterprise JavaBeans

You can define an environment reference to an EJB within the deployment descriptor. If your bean calls out to another bean, you can enable your bean to invoke the second bean using a reference defined within the deployment descriptors. You create a logical name within the EJB deployment descriptor, which is mapped to the concrete name of the bean within the OC4J-specific deployment descriptor.

Declaring the target bean as an environment reference provides a level of indirection: the originating bean can refer to the target bean with a logical name.

To define a reference to another EJB within the JAR or in a bean declared as a parent, you provide the following:

1. Name—provide a name for the target bean. This name is what the bean uses within the JNDI URL for accessing the target bean. The name should begin with

"ejb/", such as "ejb/myEmployee", and will be available within the "java:comp/env/ejb" context.

- This name can be the actual name of the bean; that is, the name defined within the <ejb-name> element in the <session> or <entity> elements.
- This name can be a logical name that you want to use in your implementation. But it is not the actual name of the bean. If you use a logical name, the actual name must either be specified in the <ejb-link> element in this <ejb-ref> element or in the <ejb-ref-mapping> element in the OC4J-specific deployment descriptor.

These options are discussed below.

2. Type—define whether the bean is a session or an entity bean. Value should be either "Session" or "Entity".
3. Home—provide the fully qualified home interface name.
4. Remote—provide the fully qualified remote interface name.
5. Link—provide a name that links this EJB reference with the actual JNDI URL. This is optional.

If you have two beans in the JAR: BeanA and BeanB. If BeanB creates a reference to BeanA, you can define this reference in one of three methods:

- Provide the actual name of the bean. BeanB would define the following <ejb-ref> within its definition:

```
<ejb-ref>
 <ejb-ref-name>myBeans/BeanA</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <home>myBeans.BeanAHome</home>
 <remote>myBeans.BeanA</remote>
</ejb-ref>
```

  No <ejb-link> is necessary for this method. However, the BeanB implementation must refer to BeanA in the JNDI retrieval, which would use java:comp/env/myBeans/BeanA for retrieval within an EJB or Java client and use "myBeans/BeanA" within a Servlet.

Note: Servlets do not require the prefix of "java:comp/env" in the JNDI lookup. Thus, they will always either reference just the actual JNDI name or the logical name of the EJB.

- Provide the actual name of the bean in the <ejb-link> element. This method allows you to use any logical name in your bean implementation for the JNDI retrieval:

  ```
  <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>
  ```

  BeanB would use java:comp/env/ejb/nextVal in the JNDI retrieval of BeanA.

- Provide the logical name of the bean in the <ejb-ref-name> and the actual name of the bean in the <ejb-ref-mapping> element in the OC4J-specific deployment descriptor.

  The reference in the EJB deployment descriptor would be as follows:

  ```
  <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>
  ```

  The "ejb/nextVal" logical name is mapped to an actual name in the OC4J-deployment descriptor as follows:

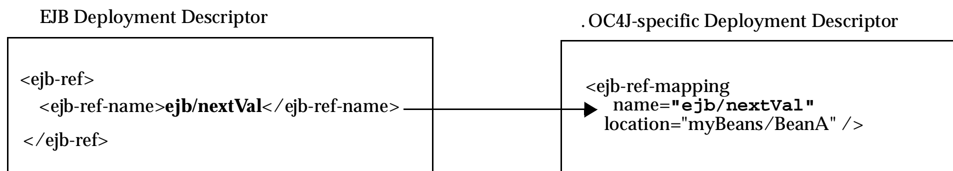
  ```
  <ejb-ref-mapping name="ejb/nextVal" location="myBeans/BeanA"/>
  ```

  BeanB would use java:comp/env/ejb/nextVal in the JNDI retrieval of BeanA.

As shown in Figure 8–2, the logical name for the bean is mapped to the JNDI name by providing the same name, "ejb/nextVal", in both the <ejb-ref-name> in the

EJB deployment descriptor and the `name` attribute within the `<ejb-ref-mapping>` element in the OC4J-specific deployment descriptor.

Figure 8–2 EJB Reference Mapping



Example 8–3 Defining an EJB Reference Within the Environment

The following example defines a reference to the `Hello` bean, as follows:

1. The logical name used for the target bean within the originating bean is "`java:comp/env/ejb/HelloWorld`".
2. The target bean is a session bean.
3. Its home interface is `hello.HelloHome`; its remote interface is `hello.Hello`.
4. The link to the JNDI URL for this bean is defined in the OC4J-specific deployment descriptor under the "`HelloWorldBean`" name.

EJB Deployment Descriptor

```
<ejb-ref>
  <description>Hello World Bean</description>
  <ejb-ref-name>ejb/HelloWorld</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>hello.HelloHome</home>
  <remote>hello.Hello</remote>
</ejb-ref>
```

As shown in Figure 8–2, the `<ejb-link>` is mapped to the `name` attribute within the `<ejb-ref-mapping>` element in the OC4J-specific deployment descriptor by providing the same logical name in both elements. The Oracle-specific deployment descriptor would have the following definition to map the logical bean name of "`java:comp/env/ejb/HelloWorld`" to the JNDI URL "`/test/myHello`".

OC 4J-specific Deployment Descriptor

```
<ejb-ref-mapping>
  name="ejb/HelloWorld"
  location="/test/myHello"/>
```

To invoke this bean from within your implementation, you use the <ejb-ref-name> defined in the EJB deployment descriptor. In EJB or pure Java clients, you prefix this name with "java:comp/env/ejb/", which is where the container places the EJB references defined in the deployment descriptor. Servlets only require the logical name defined in the <ejb-ref-name>.

The following is a lookup from an EJB client:

```
InitialContext ic = new InitialContext();
HelloHome hh = (HelloHome)ic.lookup("java:comp/env/ejb/HelloWorld");
```

The following is a lookup from a Servlet, if the Servlet defines the logical name of "ejb/HelloWorld" in <ejb-ref> in its web.xml file and maps it to the actual name of "/test/myHello" within the orion-web.xml file.

```
InitialContext ic = new InitialContext();
HelloHome hh = (HelloHome)ic.lookup("ejb/HelloWorld");
```

Environment References To Resource Manager Connection Factory References

The resource manager connection factory references can include resource managers such as JMS, Java mail, URL, and JDBC DataSource objects. Similar to the EJB references, you can access these objects from JNDI by creating an environment element for each object reference. However, these references can only be used for retrieving the object within the bean that defines these references. Each is fully described in the following sections:

- JDBC DataSource
- Mail Session
- URL

JDBC DataSource

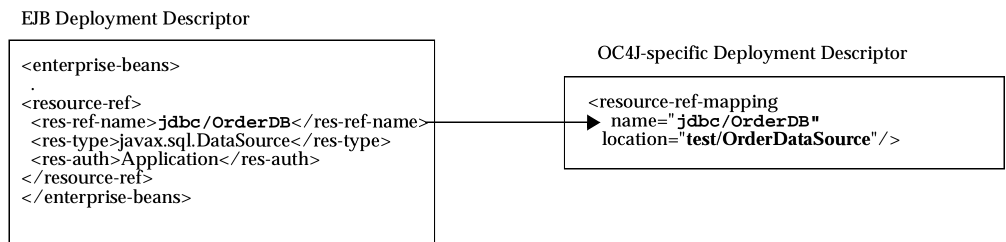
You can access a database through JDBC either using the traditional method or by creating an environment element for a JDBC DataSource. In order to create an environment element for your JDBC DataSource, you must do the following:

1. Define the DataSource in the data-sources.xml file.
2. Create a logical name within the <res-ref-name> element in the EJB deployment descriptor. This name should always start with "jdbc". In the bean code, the lookup of this reference is always prefaced by "java:comp/env/jdbc".

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/jdbc`" preface and the logical name defined in the EJB deployment descriptor.

As shown in Figure 8–3, the JDBC `DataSource` uses the JNDI name "`test/OrderDataSource`". The logical name that the bean knows this resource as is "`jdbc/OrderDB`". These names are mapped together within the OC4J-specific deployment descriptor. Thus, within the bean's implementation, the bean can retrieve the connection to `OrderDataSource` by using the "`java:comp/env/jdbc/OrderDB`" environment element.

Figure 8–3 JDBC Resource Manager Mapping



Example 8–4 Defining an environment element for JDBC Connection

The environment element is defined within the EJB deployment descriptor by providing the logical name, "`jdbc/OrderDB`", its type of `javax.sql.DataSource`, and the authenticator of "`Application`".

EJB Deployment Descriptor

```
<resource-ref>
  <res-ref-name>jdbc/OrderDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Application</res-auth>
</resource-ref>
```

The environment element of "`jdbc/OrderDB`" is mapped to the JNDI bound name for the connection, "`test/OrderDataSource`" within the Oracle-specific deployment descriptor.

OC4J-specific Deployment Descriptor

```
<resource-ref-mapping
  name="jdbc/OrderDB"
  location="/test/OrderDataSource"/>
```

Once deployed, the bean can retrieve the JDBC `DataSource` as follows:

```
javax.sql.DataSource db;
java.sql.Connection conn;
.
.
.
db = (javax.sql.DataSource)
initCtx.lookup("java:comp/env/jdbc/OrderDB");
conn = db.getConnection();
```

Note: This example assumes that a `DataSource` is specified in the `data-sources.xml` file with the JNDI name of "`/test/OrderDataSource`".

Mail Session

You can create an environment element for a Java mail `Session` object through the following:

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.

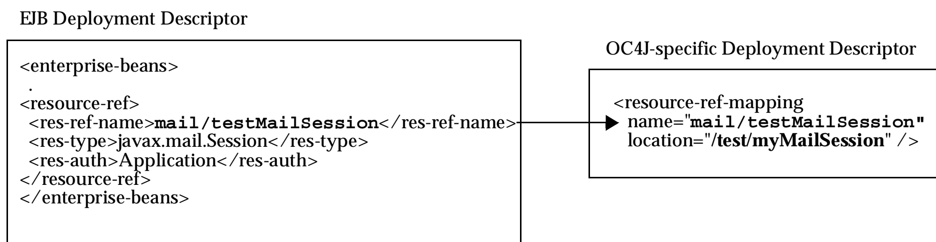
2. Create a logical name within the `<res-ref-name>` element in the EJB deployment descriptor. This name should always start with "`mail`". In the bean

code, the lookup of this reference is always prefaced by "`java:comp/env/mail`".

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 8–4, 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 8–4 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 ordering database 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 8–5 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 as follows:

```
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);
```

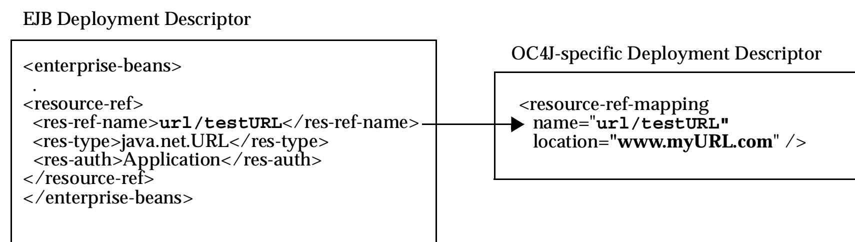
URL

You can create an environment element for a Java `URL` object through the following:

1. Create a logical name within the `<res-ref-name>` element in the EJB deployment descriptor. This name should always start with "`url`". In the bean code, the lookup of this reference is always prefaced by "`java:comp/env/url`".
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 8–5, the `URL` object was bound to the URL "`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 the connection to the bound `Session` object by using the "`java:comp/env/url/testURL`" environment element.

Figure 8–5 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 ordering database is "`url/testURL`".

Element	Description
<res-type>	The Java type of the resource. For the Java URL object, this is java.net.URL.
<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 8–6 Defining an environment element for JDBC Connection

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 "www.myURL.com" within the OC4J-specific deployment descriptor.

OC4J-specific Deployment Descriptor

```
<resource-ref-mapping
  name="url/testURL"
  location="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
URLConection conn = url.openConnection();
```

Configuring Security

EJB security involves two realms: granting permissions if you download into a browser and configuring your application for authentication and authorization. This section covers the following:

- Granting Permissions in Browser
- Authenticating and Authorizing EJB Applications
- Specifying Credentials in EJB Clients

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 JAZN or XML user manager. For a full description of security user managers, see the *Oracle9iAS Containers for J2EE User's Guide* and *Oracle9iAS 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:

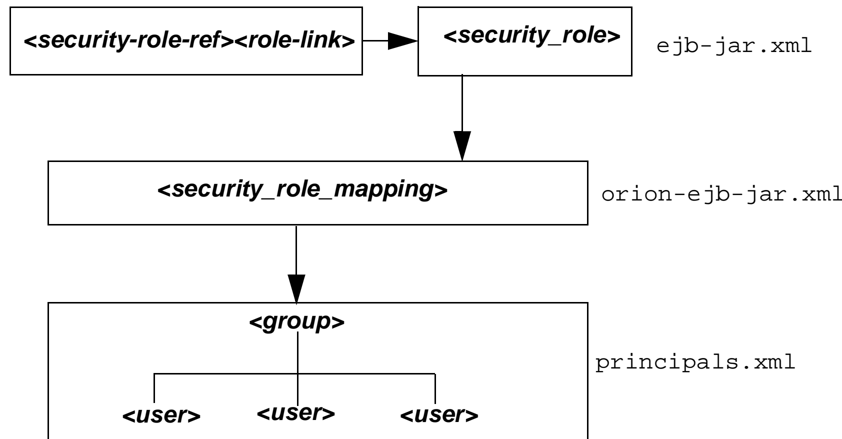
- The EJB deployment descriptor describes access rules using logical roles.
- The OC4J-specific deployment descriptor maps the logical roles to concrete users and groups, which are defined either the JAZN or XML user managers.

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 8–6.

Figure 8–6 Role Mapping



Defining users, groups, and roles are discussed in the following sections:

- Specifying Users and Groups
- Specifying Logical Roles in the EJB Deployment Descriptor
- Specifying 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

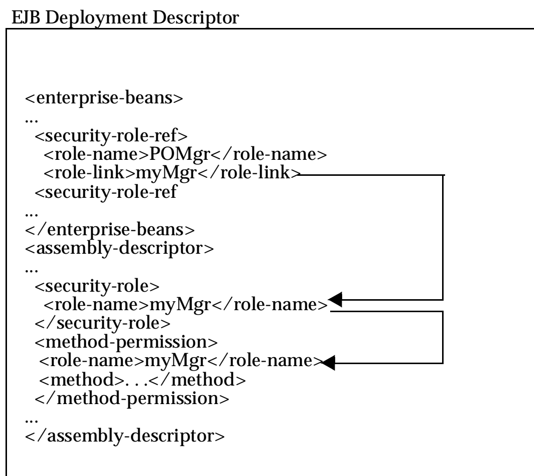
Specifying Users and Groups

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 JAZN or XML user managers. See the *Oracle9iAS Containers for J2EE User's Guide* and *Oracle9iAS Containers for J2EE Services Guide.* for directions.

Specifying Logical Roles in the EJB Deployment Descriptor

As shown in Figure 8–7, 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 8-34 for more information.

Figure 8–7 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.

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:

1. 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>
```

2. 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.

3. Defining a method with a specific signature among many overloaded versions, as follows:

```
<method-permission>
  <role-name>myMgr</role-name>
  <method>
    <ejb-name>myBean</ejb-name>
    <method-name>myMethod</method-name>
    <method-params>
       <method-param>javax.lang.String</method-param>
       <method-param>javax.lang.String</method-param>
    </method-params>
  </method>
</method-permission>
```

The parameters are the fully-qualified Java types of the method's input parameters. If the method has no input arguments, the `<method-params>` element contains no elements. Arrays are specified by the array element's type, followed by one or more pair of square brackets, such as `int[ ][ ]`.

Specifying Unchecked Security for EJB Methods

If you want certain methods to not be checked for security roles, you define these methods as unchecked, 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.

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 JAZN or XML User Managers.

Map the logical roles defined in the application deployment descriptors to JAZN 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 JAZN or XML User Manager configuration. See *Oracle9iAS Containers for J2EE User's Guide* and *Oracle9iAS Containers for J2EE Services Guide* for a description of the JAZN and XML User Managers.

Example 8–7 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:

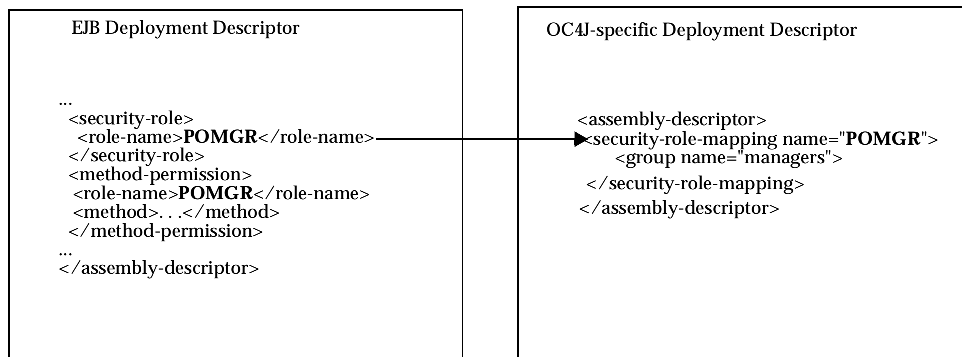
```
<security-role-mapping name="POMGR">
 <user name="guest" />
</security-role-mapping>
```

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 8–8, 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 8–8 Security Mapping



Notice that the `<role-name>` in the EJB deployment descriptor is the same as the name attribute in the `<security-role-mapping>` element in the OC4J-specific deployment descriptor. This is what identifies the mapping.

Specifying a Default Role Mapping for Undefined Methods

If any methods have not been associated with a role mapping, they are mapped to the default security role through the `<default-method-access>` element in the `orion-ejb-jar.xml` file. The following is the automatic mapping for any insecure methods:

```
<default-method-access>
   <security-role-mapping name="&lt;default-ejb-caller-role&gt;"
                          impliesAll="true" />
   </security-role-mapping>
</default-method-access>
```

The default role is `<default-ejb-caller-role>` and is defined in the `name` attribute. You can replace this string with any name for the default role. The `impliesAll` attribute indicates whether any security role checking occurs for these methods. 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 JAZN 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 JAZN 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 8-36 for more information.

Specifying Credentials in EJB Clients

When you access EJBs in a *remote* container, you must pass valid credentials to this container.

- Stand-alone clients define their credentials in the `jndi.properties` file deployed with the EAR file.
- Servlets or JavaBeans running within the container pass their credentials within the `InitialContext`, which is created to look up the remote EJBs.

Credentials in JNDI Properties

Indicate the username (principal) and password (credentials) to use when looking up remote EJBs in the `jndi.properties` file.

For example, if you want to access remote EJBs as `POMGR/welcome`, define the following properties. The `factory.initial` property indicates that you will use the Oracle JNDI implementation:

```
java.naming.security.principal=POMGR
java.naming.security.credentials=welcome
java.naming.factory.initial=
com.evermind.server.ApplicationClientInitialContextFactory
java.naming.provider.url=ormi://localhost/ejbsamples
```

In your application program, authenticate and access the remote EJBs, as shown below:

```
InitialContext ic = new InitialContext();
CustomerHome =
(CustomerHome)ic.lookup("java:comp/env/purchaseOrderBean");
```

Credentials in the InitialContext

To access remote EJBs from a servlet or JavaBean, pass the credentials in the `InitialContext` object, as follows:

```
Hashtable env = new Hashtable();
env.put("java.naming.provider.url", "ormi://localhost/ejbsamples");
env.put("java.naming.factory.initial",
        "com.evermind.server.ApplicationClientInitialContextFactory");
env.put(Context.SECURITY_PRINCIPAL, "POMGR");
env.put(Context.SECURITY_CREDENTIALS, "welcome");
Context ic = new InitialContext (env);
CustomerHome =
    (CustomerHome)ic.lookup("java:comp/env/purchaseOrderBean")
```

Setting Performance Options

Most performance settings are discussed in the *Oracle9i Application Server Performance Guide.* This section discusses other performance options.

You can manage these performance settings yourself from either the OC4J command-line option or by editing the appropriate XML file element.

- Performance Command-Line Options
- Thread Pool Settings
- Statement Caching
- Task Manager Granularity
- Using DNS for Load Balancing

Performance Command-Line Options

Each `-D` command-line option, except for the `dedicated.rmicontext` option, defaults to the recommended setting. However, you can modify these options by providing each `-D` command-line option as an OC4J option. See the *Oracle9iAS Containers for J2EE User's Guide* for an example.

- `dedicated.rmicontext=true/false`. The default value is false. This replaces the deprecated `dedicated.connection` setting. When two or more clients in the same process retrieve an `InitialContext`, OC4J returns a cached context. Thus, each client receives the same `InitialContext`, which is assigned to the process. Server lookup, which results in server load balancing, happens only if the client retrieves its own `InitialContext`. If you set `dedicated.rmicontext=true`, then each client receives its own `InitialContext` instead of a shared context. When each client has its own `InitialContext`, then the clients can be load balanced.

  This parameter is for the client. You can also set this in the JNDI properties. See Example 8–10 for an example.

- `oracle.dms.gate=true/false`. You can turn on and off collecting Oracle9*i*AS built-in performance metrics. The default value is true, which turns on the collection. To turn off the collection, set this option to false. This parameter should be set on the OC4J server.
- `DefineColumnType=true/false`. The default is true. If true, you avoid a round-trip when executing a select over the JDBC driver. You should be concerned with this flag only when you are using a non-Oracle JDBC driver.

With a non-Oracle JDBC driver, you want to turn this flag to false. This parameter should be set on the OC4J server.

When you change the value of this option and restart OC4J, it is only valid for applications deployed after the change. Any applications deployed before the change are not affected.

The `DefineColumnType` extension saves a round trip to the database that would otherwise be necessary to describe the table. When the Oracle JDBC driver performs a query, it first uses a round trip to an Oracle database to determine the types that it should use for the columns of the result set. Then, when JDBC receives data from the query, it converts the data, as necessary, as it populates the result set. When you specify column types for a query with the `DefineColumnType` extension, you avoid the first round trip to the database. The server, which is optimized to do so, performs any necessary type conversions. If you want OC4J to perform this optimization for you, then set this option to true, which is the default. If you do not, then set this option to false.

Thread Pool Settings

You can specify one or two thread pools for an OC4J process through the `global-thread-pool` element in the `server.xml` file. If you do not specify this element, then an infinite number of threads can be created.

There are two types of threads in OC4J:

- short lived threads: A worker thread that is process intensive and uses database resources. These threads are mapped `ApplicationServerThreadPool`.
- long lived threads: A connection thread that is not process intensive. It listens for events or processes socket IOs. These threads are mapped to `ConnectionThreadPool`.

OC4J always maintains a certain amount of worker threads, so that any client connection traffic bursts can be handled.

If you specify a single thread pool, then both short and long lived threads exist in this pool. The risk is that all the available threads in the pool are one type of thread. Then, performance can be poor because of a lack of resources for the other type of thread. However, OC4J always guarantees a certain amount of worker threads, which are normally mapped to short lived threads. If a need for a worker thread arises and no short lived thread is available, the work is handled by a long lived thread.

If you specify two thread pools, then each pool contains one type of thread.

To create a single pool, configure the `min`, `max`, `queue`, and `keepAlive` attributes. To create two pools, configure the `min`, `max`, `queue`, and `keepAlive` attributes for the first pool and the `cx-min`, `cx-max`, `cx-queue`, and `cx-keepAlive` attributes for the second pool.

The `global-thread-pool` element provides the following attributes:

Table 8–1 The Thread Pool Attributes

Thread Pool Attributes	Description
`min`	The minimum number of threads that OC4J can simultaneously execute. By default, a minimum number of threads are preallocated and placed in the thread pool when the container starts. Value is an integer. The default is 20. The minimum value you can set this to is 10.
`max`	The maximum number of threads that OC4J can simultaneously execute. New threads are spawned if the maximum size is not reached and if there are no idle threads. Idle threads are used first before a new thread is spawned. Value is an integer. The default is 40.
`queue`	The maximum number of requests that can be kept in the queue. Value is an integer. The default is 80.
`keepAlive`	The number of milliseconds to keep a thread alive (idle) while waiting for a new request. This timeout designates how long an idle thread remains alive. If the timeout is reached, the thread is destroyed. The minimum time is a minute. Time is set in milliseconds. To never destroy threads, set this timeout to a negative one. Value is a long. The default is 600000 milliseconds.
`cx-min`	The minimum number of threads that OC4J can simultaneously execute. Value is an integer. The default is 20. The minimum value you can set this to is 10.
`cx-max`	The maximum number of threads that OC4J can simultaneously execute. Value is an integer. The default is 40.
`cx-queue`	The maximum number of requests that can be kept in the queue. Value is an integer. The default is 80.

Table 8–1 The Thread Pool Attributes (Cont.)

Thread Pool Attributes	Description
cx-keepAlive	The number of milliseconds to keep a thread alive (idle) while waiting for a new request. This timeout designates how long an idle thread remains alive. If the timeout is reached, the thread is destroyed. The minimum time is a minute. Time is set in milliseconds. To never destroy threads, set this timeout to a negative one. Value is a long. The default is 600000 milliseconds.
debug	If true, print the application server thread pool information at startup. The default is false.

Recommendations:

- The `queue` attributes should be at least twice the size of the maximum number of threads.
- The minimum and maximum number of worker threads should be a multiple of the number of CPUs installed on your machine and fairly small. The more threads you have, the more burden you put on the operating system and the garbage collector. The minimum that you should set it to is 10.
- The `cx-min` and `cx-max` sets the thread pool size for the connection threads; thus, they are relative to the number of the physical connections you have at any point in time. The `cx-queue` handles burst in connection traffic.
- When running benchmarks or in a production environment, once you figure out the right number of threads, set the minimum to the maximum number and the `keepAlive` attribute to negative one.

Example 8–8 Setting Thread Pool

The following example initializes two thread pools for the OC4J process. Each contains at minimum 10 threads and maximum of 100 threads. The number of requests outstanding in each queue can be 200 requests. Also, idle threads are kept alive for 700 seconds. The thread pool information is printed at startup.

```
<application-server ...>
...
   <global-thread-pool min="10" max="100" queue="200"
        keepAlive=700000" cx-min="10" cx-max="100" cx-queue="200"
        cx-keepAlive=700000" debug="true"/>
...
```

```
</application-server>
```

Statement Caching

You can cache database statements, which prevents the overhead of repeated cursor creation and repeated statement parsing and creation. In the `DataSource` configuration, you enable JDBC statement caching, which caches executable statements that are used repeatedly. A JDBC statement cache is associated with a particular physical connection. See *Oracle9i JDBC Developer's Guide and Reference* for more information on statement caching.

You can dynamically enable and disable statement caching programmatically through the `setStmtCacheSize()` method of your connection object or through the `stmt-cache-size` XML attribute in the `DataSource` configuration. An integer value is expected with the size of the cache, which must be a value greater than 60. The cache size you specify is the maximum number of statements in the cache. The user determines how many distinct statements the application issues to the database. Then, the user sets the size of the cache to this number.

If you do not specify this element, this cache is disabled.

Example 8–9 Statement Caching

The following XML sets the statement cache size to 200 statements.

```
<data-source>
 ...
 stmt-cache-size="200"
</data-source>
```

Task Manager Granularity

The task manager is a background process that performs cleanup. However, the task manager can be expensive. You can manage when the task manager performs its duties through the `taskmanager-granularity` attribute in `server.xml`. This element sets how often the task manager is kicked off for cleanup. Value is in milliseconds. Default is 1000 milliseconds.

```
<application-server ...  taskmanager-granularity="60000" ...>
```

Using DNS for Load Balancing

To use DNS for your incoming load balancing, you can do one of the following:

- Use `RMILBInitialContextFactory` object: If you use this initial context factory, the OC4J client selects a randomly chosen host from the configured machines.
- Use `RMIInitialContextFactory` object for DNS round-robin lookup, and turn off DNS caching: If you use this option, the OC4J client uses the DNS round-robin algorithm to choos between the configured IP hosts.

You must start each OC4J process that is involved in load balancing on separate IP addresses, but with the same port number. Each IP address used must be configured in the DNS server.

Using RMILBInitialContextFactory Object

To retrieve a randomly selected machine from DNS, do the following:

1. Within DNS, map a single host name to several IP addresses. Each of the OC4J RMI port numbers must be the same for each IP address.
2. Within each client, use the `RMILBInitialContextFactory` as your initial context.

Then, the incoming calls are randomly routed to one of the back-end machines.

Example 8–10 RMILBInitialContextFactory Example

```
java.naming.factory.initial=
          com.evermind.server.rmi.RMILBInitialContextFactory
java.naming.provider.url=ormi://DNSserver:23792/applname
java.naming.security.principal=admin
java.naming.security.credentials=welcome
dedicated.rmicontext=true
```

Using RMIInitialContextFactory Object

You can choose to use the `RMIInitialContextFactory` object. In order for DNS round-robin to work properly, you must do the following:

1. Within DNS, map a single host name to several IP addresses and configure DNS for round-robin lookups. Each of the OC4J RMI port numbers must be the same for each IP address.

2. Turn off DNS caching on the client. For Solaris 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.

The incoming calls are routed in a round-robin fashion to one of the back-end machines.

Common Errors

The following are common errors that may occur when executing EJBs:

- NamingException Thrown
- Deadlock Conditions
- ClassCastException

NamingException Thrown

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 "Accessing EJBs" on page 8-2 for a discussion on setting up JNDI properties when accessing an EJB from a remote object or remote servlet.

Deadlock Conditions

If the call sequence of several beans cause a deadlock scenario, the OC4J container notices the deadlock condition and throws a Remote exception that details the deadlock condition in one of the offending beans.

ClassCastException

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.

See "Packaging and Sharing Classes" on page 8-8 for more information.

9
EJB Clustering

EJB clustering offers improved scalability and high-availability through the following circumstances:

- At a certain point, too many incoming client requests can overpower the abilities of your server. You can set up your environment to balance the load of incoming client requests among several servers.
- Servers failing and connections dropping occasionally happens. You can configure several servers in a cluster, so that communication is rerouted to another server in a failover situation.

The methods for providing load balancing and clustering for failover are different for HTTP requests than for EJB communications because Web components use different protocols than EJB components. This chapter discusses EJB clustering; the instructions for setting up the HTTP failover and load balancing environment is detailed in *Oracle9i Application Server Performance Guide*.

The following is discussed in this chapter:

- EJB Clustering Overview
- Enabling Clustering For EJBs

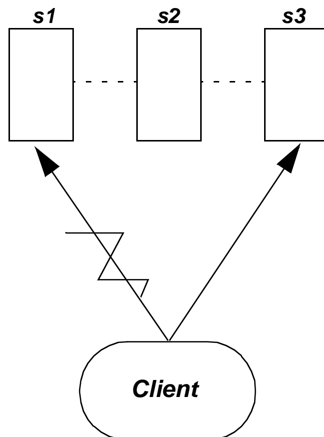
EJB Clustering Overview

To create an EJB cluster, you specify OC4J nodes that are to be involved in the cluster, configure each of them with the same multicast address, username, and password, and deploy the EJB to be clustered to each of the nodes in the cluster.

Unlike HTTP clustering, OC4J nodes included in an EJB cluster are not currently grouped in an island and do not have a load balancer as a front-end. Instead, the EJB client container stubs discover—either statically or dynamically—all the OC4J nodes in the EJB cluster, shuffle the destination addresses, and choose one from this group for the connection. Thus, the only method for load balancing and failover is a random methodology.

As Figure 9–1 demonstrates, the client container stubs chose server "`s1`" for its EJB connection. However, sometime during the conversation, the connection went down. At this point, the client container stubs shuffle the remaining OC4J node addresses and choose another server to connect to for the failover. In this example, server "`s3`" from the OC4J cluster resumes the conversation.

Figure 9–1 EJB Clustering Diagram



The client container stubs discover the OC4J server addresses by one of the following methods:

- static cluster discovery method

  The JNDI addresses of all OC4J nodes that should be contacted for load balancing and failover are provided in the lookup URL, and each address is

separated by a comma. For example, the following URL definition provides the client container with three OC4J nodes to use for load balancing and failover.

```
java.naming.provider.url=ormi://s1:23791/ejbsamples,
     ormi://s2:23793/ejbsamples, ormi://s3:23791/ejbsamples;
```

- dynamic cluster discovery method

  The JNDI addresses of all OC4J nodes that can be contacted for load balancing and failover are dynamically discovered during the first JNDI lookup. The client must perform a lookup with a "`lookup:`" prefix, as follows:

  ```
  ic.lookup("lookup:ormi://s1:23971/ejbsamples");
  ```

  During the JNDI lookup, server "`s1`" contacts the other OC4J nodes in the cluster, which are identified as a cluster if they all have the same multicast address (host/port), and retrieves their `ormi` addresses. These addresses are sent back to the client container. From this point forward, the client container shuffles these addresses for any load balancing or failover needs.

  However, the client container never tries to rediscover these addresses. Therefore, if you remove a node from the cluster and add another one during the connection, the client container will be unaware of it until the next time the client re-discovers the cluster nodes through the "`lookup:`" method.

The state of all beans are replicated at the end of every method call to all nodes in the cluster. This option is the most reliable in that the state of the bean is replicated to all nodes in the cluster, using a JMS multicast topic to all nodes in the cluster—which uses the same multicast address. This state stays in the topic until it is needed. Then when a method call comes in on the alternate node, the latest state for the bean is found in the JMS topic, reinstated, and the bean invocation continues.

These methods have different repercussions for each of the EJB types, which are discussed in the following sections:

- Stateless Session Bean Clustering
- Stateful Session Bean Clustering
- Entity Bean Clustering
- Combination of HTTP and EJB Clustering

Stateless Session Bean Clustering

Stateless session beans do not require any state to be replicated among nodes in a cluster. Thus, the only use of the clustering methods that stateless session beans have is load balancing between nodes. Both the dynamic and state cluster discovery methods can be used for stateless session beans. Failover defaults to the remote invocation handler by redirecting a request.

Stateful Session Bean Clustering

Stateful session beans require state to be replicated among nodes. In fact, stateful session beans must send all their state between the nodes, which can have a noticeable effect on performance. Thus, the following replication modes are available to you to decide on how to manage the replication performance cost:

- JVM termination replication mode—The state of the stateful session bean is replicated to only one other node in the cluster when the JVM is terminating, which uses JDK 1.3 shutdown hooks. Thus, you must use JVM version 1.3 or later. Within the JVM shutdown process, the state of all stateful session beans within this JVM is replicated to another server on the same multicast address. 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 will not be replicated if the power is shut off unexpectedly.
  - The state of the bean exists only on a single node at any time; the depth of failure is equal to one node.
- Stateful session context replication model—This is a finer-grain replication mode. In HTTP clustering, you can manage when and the type of information that is replicated through the `setAttribute` method of the `HTTPSession` object. Oracle offers a similar method through a new OC4J-specific class: `com.evermind.server.ejb.statefulSessionContext`. Although this option is a performant and reliable mechanism, it does not comply with the J2EE specification. Thus, if you provide this within your server code, you cannot port this application to any non-Oracle J2EE server.

Entity Bean Clustering

The state of the entity bean is saved in a persistent storage, such as a database. Thus, when the client loses the connection to one node in the cluster, it can switch to another node in the cluster without worrying about replication of the entity bean state. However, to ensure that the state is updated from the persistent storage when the load balancing occurs, the entity bean that changes state notifies other nodes

that their state is no longer in synch. That is, that their state is "dirty". At this point, nothing is done. If failover occurs and the client accesses another node for this entity bean, then the bean notices that its cache is dirty and resynchronizes its cache to the "`READ_COMMITTED`" state within the database.

Combination of HTTP and EJB Clustering

If you have a servlet that invokes an EJB, you must include both the HTTP and EJB clustering. For HTTP clustering options, see the HTTP clustering white paper. The type of EJB clustering you choose is based on the EJB type. If you do not configure for both types, you will not have the proper state replication for the type for which you did not configure.

If the HTTP invokes an EJB that is colocated, the `EJBReference` cannot be replicated to another node unless EJB clustering has been enabled. Instead, a null pointer will be copied to the other node. So, you must provide for both types of clustering in order for all of the correct information to be replicated.

Enabling Clustering For EJBs

To enable the OC4J nodes for EJB clustering, you must perform the following steps:

1. Configure each node in the cluster with the multicast address, an island identifier, and a unique node identifier.
2. Configure state replication for any stateful session beans.
3. Deploy the EJB to be clustered on all nodes.
4. Modify the client to use either the dynamic or static method for retrieving the cluster node addresses. The dynamic method is recommended.

Configure Nodes With Multicast Address and Identifier

When you are configuring each OC4J node included in the EJB cluster, you must configure the following:

- Configure each node with an identical multicast address (host and port number), username, and password.
- Configure the island in which the OC4J JVM will be involved.
- Configure each node in the cluster with its own unique identifier within the cluster.

You can test a network for multicast ability by pinging the following hosts:

- To ping all multicast hosts, execute: `ping 224.0.0.1`.
- To ping all multicast routers, execute: `ping 224.0.0.2`.

Modify the **rmi.xml** file and add the `<cluster>` tag to configure the multicast address, username, password, island identifier, and unique node identifier for the OC4J node, as follows:

```
<cluster host=<multi_host> port=<multi_port>
           username=<multi_user> password=<multi_pwd> id=<island_id> />
```

where each variable should be the following:

- `multi_host`: The multicast host used for the EJB cluster that communicates among the nodes in the cluster. The IP addresses that you can use for multicast are between 224.0.0.0 and 239.255.255.255. You must specify this variable.
- `multi_port`: The multicast port used for the EJB cluster for communication among the nodes in the cluster.
- `multi_user/multi_pwd`: The username and password used to authenticate itself to other nodes in the cluster. If the username and password are different for other nodes in the cluster, they will fail to communicate. You can have multiple username and password combinations within a multicast address. Those with the same username/password combinations will be considered a unique cluster.
- `island_id`: The identifier that designates what cluster the EJBs on this OC4J JVM are included.

For example, the following cluster definition identifies a cluster on mulitcast address of host=230.0.0.1, port=9127, username=`mult1`, password=`hwdr`, and id="1":

```
<cluster host="230.0.0.1" port="9127"
        username="mult1" password="hwdr" id="1" />
```

Specify the unique node identifier number as follows:

- Specify the node identifier in the **server.xml** file with the `<cluster>` tag, as follows:

  ```
  <cluster id="123"/>
  ```

- If no identifier is specified, a default identifier consists of the host IP address and port of the node itself.

Note: The dynamic peer discovery mechanism uses RMI as the mechanism for communication. You must have an RMI listener configured in the `rmi.xml` file with the following syntax:

```
<rmi_server host="<hostname>" port="<port>" />
```

The host name must be the actual name of your node. Do not use the "`localhost`" variable.

EJB Replication Configuration

Modify the **`orion-ejb-jar.xml`** file to add the state replication configuration for stateful session beans. Since you configure the replication type for the stateful session bean within the bean deployment descriptor, each bean can use a different type of replication.

VM Termination Replication Set the `replication` attribute of the `<session-deployment>` tag in the `orion-ejb-jar.xml` file to "`VMTermination`". This is shown below:

```
<session-deployment replication="VMTermination" .../>
```

End of Call Replication Set the `replication` attribute of the `<session-deployment>` tag in the `orion-ejb-jar.xml` file to "`endOfCall`". This is shown below:

```
<session-deployment replication="EndOfCall" .../>
```

Stateful Session Context No static configuration is necessary when using the stateful session context to replicate information across the clustered nodes. To replicate the desired state, set the information that you want replicated and execute the `setAttribute` method within the `StatefulSessionContext` class in the server code. This enables you to designate what information is replicated and when it is replicated. The state indicated in the parameters of this method is replicated to all nodes in the cluster that share the same multicast address, username, and password.

Deploy EJB Application To All Nodes

Deploy the EJB application to all nodes in the cluster. If you do not do so, the client container shuffles through the nodes in the cluster until it finds a node with the EJB deployed on it. This will affect your performance.

You can either deploy the application to each node individually using the `-cluster` option of the `admin.jar` tool or you can use Oracle Enterprise Manager (OEM), which can deploy your application for you to multiple nodes.

Use the following syntax with the `admin.jar` tool:

```
java -jar admin.jar ormi://myhost admin welcome
          -deploy -file bmpapp.ear -deploymentName bmpapp -cluster
```

Application Client Retrieval Of Clustered Nodes

The client container designates randomly within the nodes in the cluster where to direct the client request. As discussed above, the container discovers the nodes within the cluster through one of the following methods:

- Static Retrieval
- Dynamic Retrieval

Static Retrieval

The JNDI addresses of all OC4J nodes that should be contacted for load balancing and failover are supplied in the lookup URL, and each address is separated by a comma. For example, the following URL definition provides the client container with three OC4J nodes to use for load balancing and failover.

```
java.naming.provider.url=ormi://s1:23791/ejbsamples,
        ormi://s2:23793/ejbsamples, ormi://s3:23791/ejbsamples;
```

Dynamic Retrieval

The JNDI addresses of all OC4J nodes that can be contacted for load balancing and failover are dynamically discovered during the first JNDI lookup. The client must perform a lookup with a "`lookup:`" prefix, as follows:

```
ic.lookup("lookup:ormi://s1:23971/ejbsamples");
```

During the JNDI lookup, server "`s1`" contacts the other OC4J nodes in the cluster, which are identified as a cluster if they all have the same multicast address (host/port), and retrieves their `ormi` addresses. These addresses are sent back to the

client container. From this point forward, the client container shuffles these addresses for any load balancing or failover needs.

The client container never tries to rediscover these addresses, though. Therefore, if you remove a node from the cluster and add another one during the connection, the client container will be unaware of it until the next JNDI lookup.

Load Balancing Options

If you configure for load balancing, it balances the load at the connection level. However, if you want load balancing to occur on each JNDI lookup, configure the `LoadBalanceOnLookup` property to true in the JNDI properties before retrieving the InitialContext, as follows:

```
env.put("LoadBalanceOnLookup", "true");
```

10

Active Components for Java

Active Components for Java (AC4J) enables applications to interact as peers in a loosely coupled manner. Two or more applications participating in a business interaction exchange information for the purpose of requesting service and responding with results.

This document describes software Oracle provides to manage loosely coupled interactions between autonomous applications. It also discusses the architecture necessary to run the software.

- Future Needs of Business Applications
- Architectures
- AC4J Architecture
- Configuring Oracle Databases to Support AC4J
- AC4J Example
- Administering AC4J

Future Needs of Business Applications

The future of business applications requires the ability to perform loosely coupled interactions. That is, applications should be able to exchange information with other applications over a long period of time, without limiting resources, and by surviving system crashes. Loosely coupled interactions have the following requirements:

1. Autonomous peer—Each application, when interacting with another application, exists as an autonomous peer. That is, the responding application can choose to ignore the request, or to execute one or more functions on behalf of the requester (possibly different from the one that the requester asked for), before responding to the initiating application. As peers, both applications can make requests to each other, but neither can require submission from the other. Neither application can assume control over the resources that its peer application owns.
2. No time constraints—Because the tasks that are performed sometimes take days, even months, to complete, there must be no time limits imposed.
3. Asynchronous exchange of information—Loosely coupled interactions require that all exchange of information exists within an asynchronous environment. The inter-peer interaction performs one of the following:
   - Distribute information interaction—Sometimes an application must be able to distribute information asynchronously to its peer where no response is required. However, reliability of the delivery for this message must be ensured.
   - Request or respond to interaction between components in an asynchronous manner—Applications use a request and response mode of communication, in which each entity knows where to respond with the results.
4. Reliability—For any application to exist over a period of time, the application must be reliable; that is, recoverable and restartable, in case of system failures during that time.
5. Scalability—The application must be scalable; that is, a long-running application cannot block execution or lock resources for long periods of time. For the application to execute in a reasonable time frame, the framework must provide performance enhancements through concurrently executing computations.
6. Monitoring ability—The application must be able to define and track its business processes and their interaction patterns, including the following:

- What business processes have started or completed under what business conditions
- What business processes are pending, and waiting for what business documents
- What the pattern of interaction of the business processes is, where processes can exchange information, and what information they are authorized to push to or pull from other processes
- What the sequencing of execution of the defined processes is

Architectures

Until AC4J, the following two architectures were available:

- Remote Procedure Call Model—Provides a tightly coupled environment that uses request-response mechanisms in communication.
- Database Transactional Queueing Model—Provides a loosely coupled environment that uses a one-way mechanism for communication.

The following sections briefly describe these models and show why they do not offer the basis necessary for the six goals presented in “Future Needs of Business Applications” on page 10-2. After the two architectures are described, AC4J is shown to be not just a new technology, but one that builds on the two architectures, eliminating their negatives and drawing on their positives. AC4J has the look and feel of Remote Procedure Call and database queueing, which it uses as building blocks.

Remote Procedure Call Model

One of the building blocks of AC4J, the Remote Procedure Call (RPC) programming model, facilitates a tightly coupled environment that provides for request-response communication. Transactional RPC implementations provide for ACID (atomicity, consistency, isolation, and durability) qualities.

Most RPC implementations currently provide two modes of method invocations: synchronous and deferred synchronous.

Transactional RPC Synchronous Invocation

The client program blocks when a remote invocation is made, and waits until the results arrive or an exception is thrown. Examples of application types that use

transactional RPC implementations are EJB and most CORBA applications. Web services are also based on the RPC model, but are not transactional.

Advantage This model of communication—also known as online or connected—is based on the request-response paradigm, in which the requester and responder of the service are tightly coupled. Tightly coupled applications understand how to reply transparently to the requester.

Disadvantage The programs must be available and running for the application to work. In the event of a network or system failure, or when the application providing the service is busy, the application is not able to continue forward with its processing work. In this case, the state is inconsistent and the application must roll back to a consistent state through JTA (Java Transaction API). In addition, the application is not autonomous. One application can control resources of other applications for a long time.

JTA is based on the two-phase commit specification. The two-phase commit protocol can cause loss of application autonomy in the case of network disconnection, where the coordinator is incapable of making a coherent global decision over the outcome of the global transaction for a long period of time.

Example If a purchase order is created and the customer wants to purchase 20 widgets, then the transactional RPC application must do two things:

- Check inventory for 20 widgets and ask for them to be shipped to the customer
- Check the customer's credit to see if the customer has the ability to purchase these widgets

In this example, an RPC synchronous application would (within a global transaction) do the following:

- Send a request to the inventory database and block until the answer returns
- Send a request to the credit bureau and block until the answer returns

If both requests come back with a satisfactory report, then the transaction is committed, and the purchase order is forwarded on to shipping. If one of the two requests fails, the transaction is rolled back; of course, to prevent rollback, the application could perform the following alternatives:

- If the inventory is not available, ask if the customer will wait for a back order.
- If the credit check fails, ask the customer for another method of payment.

If the transaction is rolled back, the purchase order is voided (unless one of the alternatives is performed).

RPC Deferred Synchronous Invocation

An RPC-deferred synchronous invocation is queue-oriented. The client places a request in a queue and is then able to continue processing without blocking for the response. An example of this is a CORBA DII (Dynamic Invocation Interface) application.

Advantages The client does not need to wait for a reply to the request. Instead, the client continues processing. Then when the client wants to receive a response, it blocks or polls for the availability of the response. A response can be delivered only to the same process that made the original deferred request. Thus, if multiple deferred requests are pending, only one response is processed at a time.

Disadvantage If the client is nonexistent, then the response is lost. Thus, for deferred execution to work correctly in the presence of network, system, and application failures, the requests must be stored persistently and processed exactly once.

Example In the purchase order example, the requests to the inventory and credit bureau can be made in parallel. After executing both requests, the client can poll for both responses. The disadvantages are the same as listed within the RPC synchronous invocation example.

Database Transactional Queueing Model

Another of the building blocks, the database transactional queueing model, supports a loosely coupled environment where applications use one-way communication. Oracle AQ is an implementation of a database transactional queueing model.

Applications need to process and deliver each message exactly once, even in the presence of multiple failures of the sender or the receiver. Mixing the transactional ACID construct with queue processing creates a model that enables applications to reliably process messages with the ACID guarantees.

Applications can be disconnected for long periods of time, and occasionally they can reconnect to communicate, using messages. By decoupling the applications that send messages from the applications that receive messages and process them, queueing facilitates complex scheduling of autonomous applications. Each message can be durably saved until processed exactly once. Processing of the data is

performed in a time-independent fashion, even in a situation in which a message receiver is temporarily unavailable.

Advantage Delivers and processes messages exactly once, no matter whether the network or receiver application is available.

Disadvantage This model is based on sending and receiving messages, not on requesting and responding to service requests. Sending and receiving messages is the foundation of all business protocols for loosely coupled applications. To satisfy this requirement, the application is responsible for creating and parsing each message. Both sides must know the format, security, and headers required for each message. There is no automatic mechanism for routing messages and executing business methods. The implementation of application logic for these mechanisms is the responsibility of the applications. If a response is called for, the application cannot easily reply, because there is no context that captures the relationship between a requester and a responder application; this is not true for RPC. This model is not intended for a request-response environment, so if the client needs a response back from the destination object, it must receive and parse a separate message off of its own queue.

Exception handling describes communication failures—not application exceptions.

There is no guarantee for the consistency of the business transactions. Instead, the program itself must guarantee that the application semantic rollbacks (semantics of undoing a process but not necessarily restoring the original state) occur appropriately in a failure situation.

Example In the purchase order example, the client would enqueue a message to the inventory queue and another to the credit bureau queue. Both must be reliably processed once for the transaction to commit. If either the inventory is not available or the client's credit is not good, the business transaction cannot be successfully completed, and another message must be created to semantically roll back the one message that was processed positively.

AC4J Solution

The RPC and transactional database queueing models both have advantages and disadvantages. The disadvantages keep them from being the best solution. The disadvantages within J2EE application types, and the reasons that previous methods did not work, are as follows:

- The tightly coupled, synchronous communication of EJBs does not allow loosely coupled interactions or autonomous peer communication.

- The loosely coupled, asynchronous communication of JMS offers no correlation of messages nor does it support application consistency. JMS provides only a transport with no syntax for one-way messages.
- The loosely coupled, asynchronous communication of JMS does not enable request-response interaction between entities.
- The need for the JTA Coordinator to control all resources in the two-phase commit cannot include autonomous resources in the global transaction.

The disadvantages prevent each model from solving the business goals laid out in “Future Needs of Business Applications” on page 10-2. Thus, a new model is necessary to incorporate the advantages of both models and exclude the disadvantages.

Arising out of and building upon the previous two models, AC4J is a manager of loosely coupled interactions between autonomous EJB applications. You can partition the application into concurrently executing active units of work—known as *reactions*—whose execution is driven by data availability, and its purpose is to execute business logic and produce new data. AC4J coordinates the flow of data between reactions. When data become available on AC4J, the conditions specified by all registered reactions are checked and, if satisfied, then the execution of the methods of all matched reactions is triggered.

AC4J Architecture

AC4J allows EJBs to interact in a loosely coupled fashion. It provides the following features:

- It furnishes support for reliable asynchronous, disconnected, one-way, or request-response types of interaction with complexities of JMS programming removed, using the following:
  - It hides queues and topics and related JMS constructs from applications.
  - It supplies automatic definition of communication message formats.
  - It automatically packs and unpacks messages.
  - It automatically routes service requests to the appropriate service provider.
  - It automatically propagates the security context.
  - It supplies authorization and identity impersonation.
  - It provides automatic exception routing and handling.

    All the preceding features are integrated in the EJB framework.
- It offers transactional data-driven execution of EJB applications. It does this with composite matching on available data based on specified rules, which describe under which conditions these data can fire which EJB method. AC4J also offers transparent scheduling and activation of EJBs and execution of their methods.
- It furnishes support for forking and joining operations. This involves parallel invocation of EJB methods and synchronization on their results.
- It provides automatic tracking of the work in progress.

Introduction to AC4J Components

AC4J provides a framework for loosely coupled interactions, which are included in the following components, each described more fully after this section:

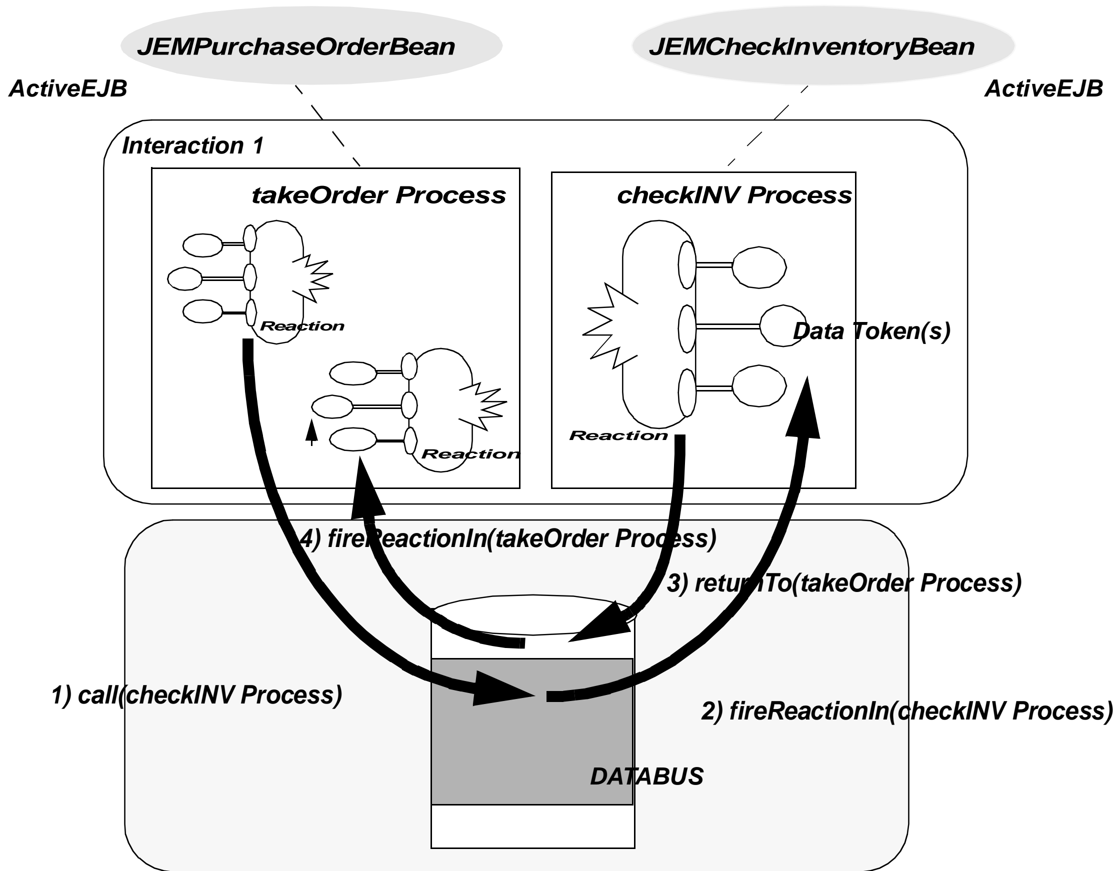
- Active EJBs: An Active EJB contains the business logic. An Active EJB business object (stateless session or entity bean) is instantiated and its method is invoked when a reaction fires.
- Interaction: An interaction is a long-lived unit of work that reflects the behavior of a business transaction. It groups a series of data exchanges (with

asynchronous, concurrent, and request-response characteristics) between processes.

- Processes: A process represents a business task. It encapsulates the units of work—reactions—that perform the detailed work of a business task.
- Reactions: A reaction performs the detailed work of a business task and is used to do the following:
  - push data to and pull data from the data bus
  - process service requests
  - request service from other Active EJBs
  - return results to the caller Active EJB business task or to the application client
  - enforce business constraints that preserve the consistency of a business transaction
  - provide application restartability in case of failures
- Data Tokens: A data token describes a request for service, or a response from a service request, or an exception condition, such as an expiration of a timer.
- Data Bus: The data bus is the fundamental component in AC4J. Applications attach to the data bus to exchange data and request services. The data bus is responsible for routing and matching of data tokens with registered reactions and enables transparent load-balancing of the attached application.

Figure 10–1 demonstrates the relationship of these components to each other. The sections following (up to, but not including, “Configuring Oracle Databases to Support AC4J” on page 10-20) describe each component.

Figure 10–1 AC4J Relationship Diagram



Active EJBs

An EJB provides a natural way for describing a business object—such as a customer, purchase order, or invoice. The externally visible business tasks of a business object, which is accessible by other applications, are separated from their internal implementation details and are described in the EJB interface.

Traditional EJBs are passive: they must be ready to immediately service a request from a client and return results quickly. Failure to deliver on these promises causes

an EJB to be unusable. AC4J allows standard stateless session and entity EJBs to become active. Active EJBs permit requests for service to be decoupled from the actual service execution. The policies that control when and which EJB methods are actually invoked are controlled by the service provider EJB. This de-coupling permits service request and service providers to interact as autonomous peers.

An application can create or look up a `JEMHandle` and then request service from a business task, which is exposed in the EJB interface.

An Active EJB is uniquely identified by a `JEMHandle` object. A `JEMHandle` object encapsulates the following:

- Active EJB name
- J2EE application name
- EJB JAR name
- EJB name
- Class name
- EJB home interface name
- EJB remote interface name
- Instance name (SID) of the database in which the data bus resides
- Primary key (available only for entity beans) of the EJB

Interaction

An interaction is a long-lived unit of work that reflects the behavior of a business transaction. A business transaction can span multiple applications that reside in different organizations. Contrary to the life of a local or a global transaction, the duration of these business transactions in this disconnected environment can be long.

The interaction represents a business goal that you want to complete. For example, if a customer wants to buy something from a business, all the actions necessary to allow the customer to pay for and receive the item he wants is characterized as an interaction. The interaction groups a series of business data exchanges by providing the global execution context of the business transaction.

These applications can run in isolation and commit or roll back their own data without knowledge of other applications. However, these applications should not be considered as different pieces, because the relationships formed among them must be coordinated and their consistency maintained. When a business transaction

becomes inconsistent, its participating applications may need to recover. The application recovery can be obtained by registering compensating reactions. For example, when the supplier has confirmed the purchase order request back to the buyer, the buyer must register a compensating reaction that monitors additional responses from the supplier that may inform him that, for example, the purchase order cannot be fulfilled because the manufacturing department is running late. If the supplier's confirmation of the request is cancelled, then the buyer's compensating reaction is matched and then fired to allow the buyer application to recover its application consistency. This reaction can pick a new supplier and request the item from the supplier or abandon the purchase order process completely.

An interaction is uniquely identified by an interaction identifier (IID). An interaction can contain multiple processes.

Processes

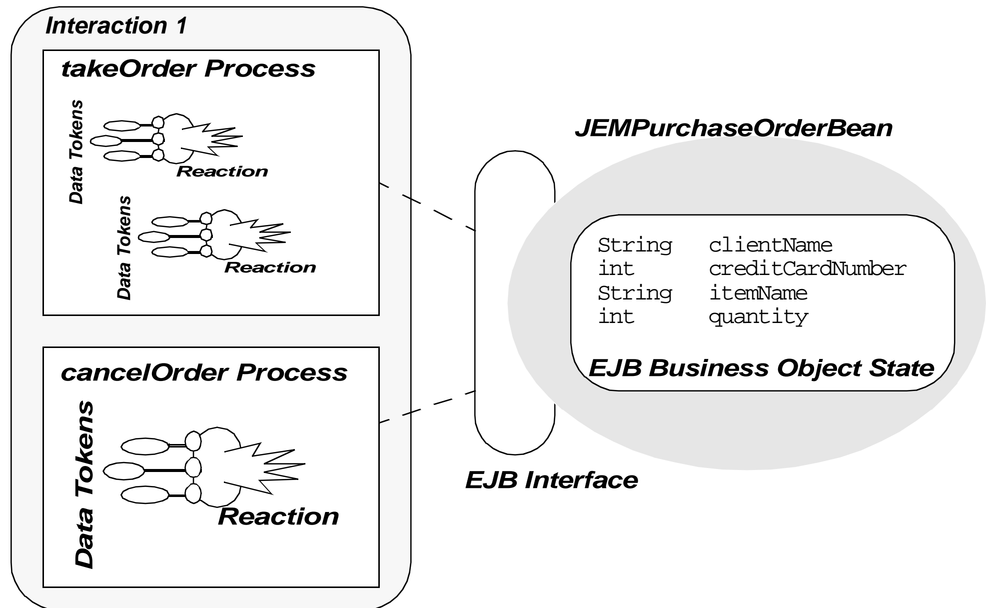
A process identifies a business task. In our purchase order example, a process exists for each of the following business tasks: creating a purchase order, checking inventory, checking customer credit, and shipping the order.

Each process does the following:

- Encapsulates the reactions that perform its detailed work
- Encapsulates data tokens, which contain the business task input parameters and its responses
- Maintains the data flow context that determines how to return the response to the invoking business task

Figure 10–2 demonstrates an Active EJB, an interaction, and two of its processes.

Figure 10–2 Relationship of Active EJB, Interaction, and Processes



A process is uniquely identified by a `JEMPortHandle` object, which encapsulates the process context and the `JEMHandle` of the Active EJB that the process belongs to. The process context is a union of an interaction identifier and the process activation identifier. AC4J automatically creates the interaction and process activation identifiers within a `call` operation. Alternatively, the application can supply them in the AC4J `JEMSession::call` operation.

Reactions

A reaction performs the detailed work of a process. Using this construct, an application can specify its persistence interest in the availability of a collection of correlated data tokens that trigger the execution of an Active EJB method. A reaction is a combination of the following:

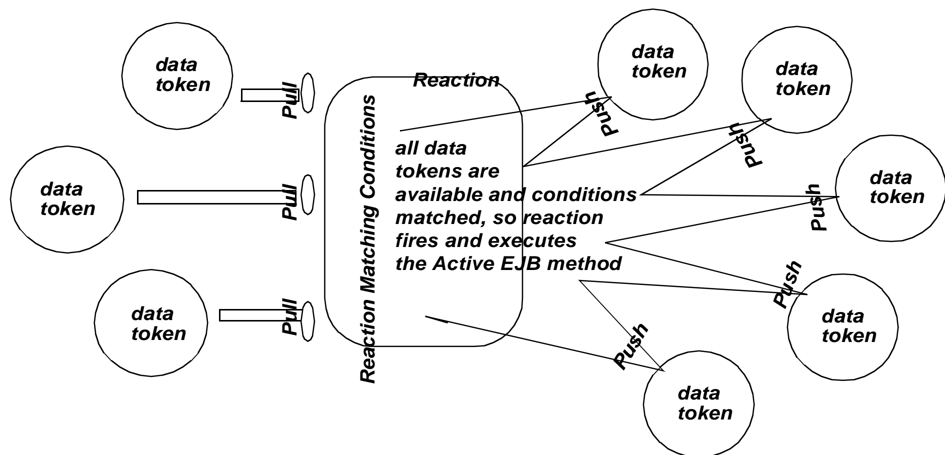
- A reaction template is a set of rules designating when the reaction will match, what data tokens are required to be pulled before firing, and under what conditions the reaction is allowed to fire.
- An Active EJB method is executed when the reaction fires.

When a process is created as the result of a AC4J `call` operation, AC4J implicitly creates a base reaction. Additionally, an application can explicitly create a reaction at run time, using the `JEMReaction::registerReaction` operation to synchronize on data tokens. The implicit or explicit `registerReaction` operation specifies the reaction template and the Active EJB method to be executed when matching succeeds.

Reactions (EJB methods) can access and modify shared database objects. These objects can be traditional database objects—thus, facilitating *coarse grain information sharing* in a transactional manner. Similarly, the reactions exchange *fine grain information*—such as Active EJB method input parameters and return values—using the AC4J data bus.

The reaction processes incoming requests, returns results based on the request, and enforces business constraints to preserve application consistency. When a reaction is fired, it can consume one or more input data parameters, process them, and then possibly produce one or more output data tokens for other reactions. Figure 10–3 illustrates how, when all data tokens are available and the conditions are matched, the reaction fires, which causes the method to execute. This method can return results that are converted to data tokens by the AC4J infrastructure and routed to the caller. This method can request additional services from other Active EJBs to complete the business task. These requests result in the creation of new data tokens, which are pushed and routed by the AC4J data bus.

Figure 10–3 Firing a Reaction



Reactions inside a process context instance can push data tokens to the AC4J data bus in the following ways:

- By issuing one or more `JEMReaction::call` operations that request service from other processes in the same or different interaction context instance
- By returning or throwing exception operations to the caller processes
- By registering a timer, using the `JEMReaction::registerReactionTimer` operation

  When the timer expires, AC4J pushes a time-out exception data token in the current reaction context instance.

Reactions inside a process context instance can pull data tokens from the AC4J data bus by registering one or more reactions in the current process context instance, using the `JEMReaction::registerReaction` method.

One or more reactions can exist for each business task. A reaction is used for the request and another for the response to support the asynchronous nature in a request-response environment. The number of reactions depends on the number of requests and responses necessary.

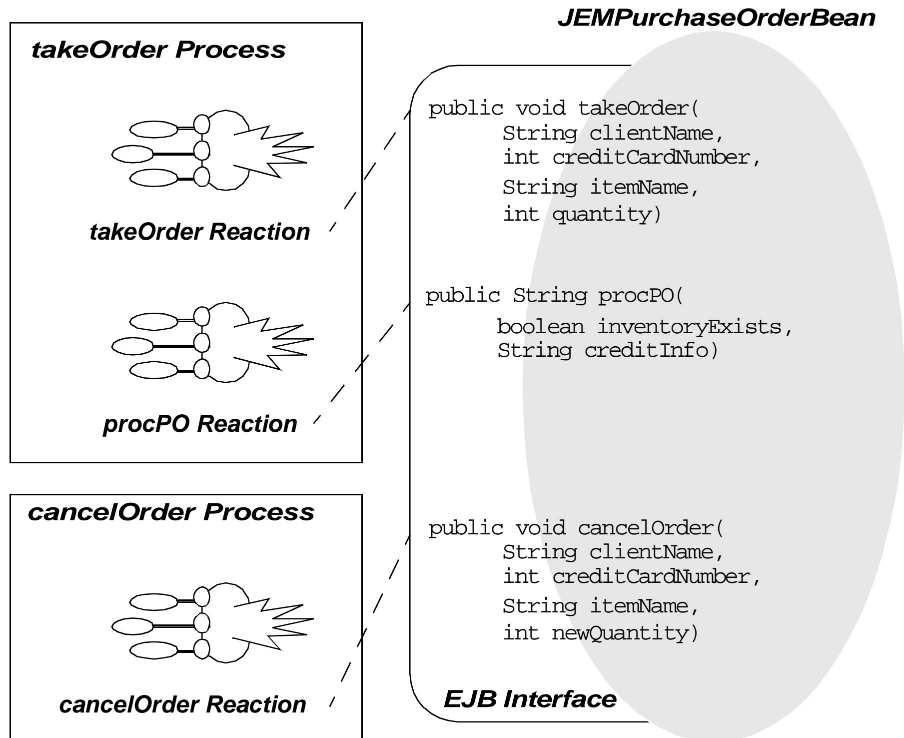
The following example demonstrates how one can receive an asynchronous communication between processes, but still have a request-response environment. The `takeOrder` process is the business task for creating the purchase order. To create the purchase order, you must check the inventory and the customer's credit. Thus, the `takeOrder` reaction invokes the following processes:

- `checkINV`—Under the conditions that the customer asks for a new purchase and provides the data of the items wanted, the `checkINV` process is activated, its `JEMInventoryBean` Active EJB is instantiated, and its base reaction—`checkINV`—reacts. Later, it returns its results to the `takeOrder` process and its `JEMPurchaseOrderBean` Active EJB.
- `checkCRED`—This process is activated, its `JEMCreditBean` Active EJB is instantiated, and its base reaction—`checkCRED`—reacts to check the customer's credit. Later, it returns its results to the `takeOrder` process and its `JEMPurchaseOrderBean` Active EJB.

After sending the asynchronous requests to the `checkINV` and `checkCRED` processes, the `takeOrder` reaction registers another reaction in the same process—`procPO`—that waits for the responses back from both the `checkCRED` and `checkINV` processes. When all data tokens expected from these processes are available, the `procPO` reaction fires and processes the responses. As Figure 10–4

shows, both the `takeOrder` and `procPO` reactions exist in the same process, because they are components of the same request-response communication.

Figure 10–4 Relationship of JEMPurchaseOrderBean Interface Methods



Note: To satisfy the AC4J requirement of not locking resources, the call should be an asynchronous AC4J call. However, you can still perform synchronous EJB calls to another bean.

Data Tokens

The activation of a reaction is triggered by the availability of data tokens. Availability is defined by the arrival of one or more data tokens, with the right conditions, and the right access mode.

When an application is requesting a service by using an AC4J call operation, the system automatically pushes a request data token, which comprises the following:

- A process descriptor, which specifies the service that is requested (such as `takeOrder`)
- A request `JEMPortHandle` object of the service provider to whom the request is destined
- A response `JEMPortHandle` object, which contains the process context (interaction and process activation identifiers) instance and the `JEMHandle` of the requester process that will later receive the results from the service provider
- Business task input arguments, which the service provider uses to honor the service

Later, when a reaction returns a response data token that is automatically generated by AC4J when an active EJB returns or throws an exception, AC4J fills in the routing information needed for sending the returned information to the caller process and fills the port handle object of the response data token. In the case in which the caller of the returning process is a client and not another process, then the data bus stores the response data token to a special data bus area from where the client can retrieve it, using the `JEMSession::receiveReactionResponseObjectInstance` operation.

The data types of the objects carried inside an input or output data token can be basic data types (such as integer, string, float, boolean) or constructed class types (such as Java serializable objects).

Data Bus

Improving the autonomy, scalability, and availability of applications requires components that are requesting services to be unaware of the identity, location, and number of components that provide these services. In AC4J, applications are attached to a data bus before starting their operation. The AC4J data bus is responsible for routing and matching data tokens that are pushed and must be pulled by registered reactions. Additionally, the data bus enables scheduling, activation, and execution of the matched reactions.

Matching Reactions

The data bus routing subsystem is responsible for making the different types of data tokens available at the specified destination, the process context instance that comprises the interaction identifier and the process activation identifier, specified by a `JEMPortHandle` object.

When data tokens are routed and become available in the data bus inside a process context instance, AC4J tries to match these data tokens with all registered reactions that are available in that context instance. The system tries to match the data token tags that are specified in a reaction template, evaluating all constraint conditions against the matched data tokens to filter and discard the inappropriate ones.

Availability of some data tokens does not mean that a registered reaction will match immediately. Only when all data tokens required by a reaction become available does matching succeed. For example, inside `takeOrder` process the `takeOrder` base reaction has registered the `procPO` reaction that is waiting for the `checkCRED` and `checkINV` processes to respond. When the `checkINV` process responds to the `takeOrder` process, the `procPO` reaction is not matched because it is also waiting for the `checkCRED` process to respond. When the `checkCRED` process responds to the `takeOrder` process, the `procPO` reaction is matched.

Additionally, data tokens that are available in the data bus can be matched with a reaction that will be registered in the future. This can be used for sequencing processes, where the completion of one process can enable another process. Inside the same interaction, the `takeOrder` process must be completed before the `cancelOrder` process can start executing. If the `takeOrder` process has not completed but the `cancelOrder` process is requested from a client, then its base reaction, which is implicitly created by the system, will not be matched, because it is waiting for the completion data token of the `takeOrder` process to be available. If the `takeOrder` process has completed (thus having already pushed its completion data token), then the `cancelOrder` process is requested from a client, and it will be immediately matched, because the completion data token of the `takeOrder` process is already available.

Matching data tokens with reactions triggers the activation of zero, one, or more reactions, which are executed in parallel if they do not conflict for shared resources.

Firing Reactions

Each method of the remote interface of an Active EJB implements the application business logic. When the data tokens become available, and are matched with a reaction, AC4J verifies that the types (primitive or class types) of the data tokens matched on the tags also match the types of the reaction Active EJB method types. Then AC4J verifies that the matched reaction is authorized to pull the available matched data tokens. If everything passes successfully, AC4J schedules the activation of the reaction.

When the matched reaction is fired, the AC4J container begins a JTA transaction and instantiates the requested Active EJB (stateless session bean or entity EJB) using the

primary key inside the `JEMHandle` request object. Then the EJB method of the fired reaction, is executed using the matched data tokens of the reaction.

AC4J automatically commits the current reaction at the end of every Active EJB method. A reaction commit marks the end of a JTA transaction so that all its changes to shared data tokens, and all its service requests and responses that have been sent, become visible. The activation of a reaction has “exactly once” semantics (that is, the code specifies that it executes exactly once) if the reaction commits. If a failure occurs after a commit, then the reaction cannot be rolled back and the changes will persist. If a failure occurs before or during a commit, then the container rolls back the current reaction. A reaction rollback reverses all changes to shared data tokens, and the service requests and responses are never sent to any recipient component. In case of failures, the data bus will retry to fire the reaction for a preconfigured number of times. The reaction is marked as completed, with exception completion status if the maximum retry attempts are reached.

In traditional databases, where the duration of a transaction is short, abnormal situations cause the whole transaction to be undone, so all performed work is lost and must be submitted again for execution. Since interactions have usually long duration and contain a large number of reactions, AC4J provides additional mechanisms to handle exceptions (such as an Oracle9*i*AS node crash or an Oracle database node crash).

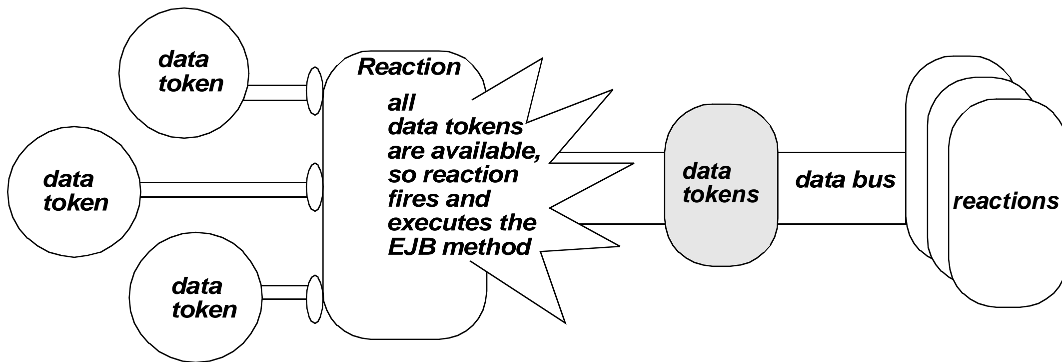
AC4j automatically makes a reaction persist in the data bus if it completes successfully. The state that is saved (process input variable data, process local variable data, and data flow context information) can be used to continue the application with minimum restart time from the last reaction. When a node crashes, all reactions that were running and did not end successfully are rolled back. AC4J then reexecutes the interrupted reactions in another OC4J instance.

AC4J uses a mechanism to capture, propagate, and match the application state and control flow information needed for resuming an application after the crash. Additionally, because reaction execution is data-driven, there is no need for the system to keep a volatile or persistent copy of the entire program state (such as program execution stack) to facilitate the storage of the control flow descriptors or the storage of data variables.

Relationships of Data Bus, Data Tokens, and Reactions

Figure 10–5 demonstrates how data tokens cause reactions to fire, and how reactions send new data tokens to other reactions over the data bus. The data bus coordinates and matches the data tokens with its reactions.

Figure 10–5 Data Bus, Data Tokens, and Reactions



After the method completes, the reaction sends information in the form of a data token to another reaction. All data tokens are sent asynchronously from one reaction to another over a data channel known as the AC4J data bus. The AC4J data bus routes the data tokens from a producer reaction to one or more consumer reactions.

Configuring Oracle Databases to Support AC4J

Before you can execute any interactions, you must initialize an Oracle9*i* database as a repository for the AC4J data bus. You must configure it to include the following:

- AC4J connection and session capabilities: This defines the number of threads AC4J can observe in the data bus
- AC4J system tablespace:
- AC4J superuser: This must be created and needs special privileges for transactions, security, and administration
- AC4J data bus: This must be configured with the number of tables and the AQ topics and queues
- One or more client users

You can add the elements of the preceding list to your Oracle9*i* database with scripts that are contained in the `ac4j-sql.jar` file that was downloaded with your Oracle9*i*AS installation. Unzip this JAR file, which contains a `README.TXT` file that discusses the different SQL command options that are available to you. These commands are also described in the following:

To create AC4J capabilities, you must execute one of the following SQL scripts as a `SYS` user on the same system as the database.

- `createall`: To create all the defaults, including the default data bus, AC4J superuser, and default client user (`JEMCLIUSER`), execute the `createall` SQL script.
- `createjemtablespace`: To create the table space for your AC4J system, execute the `createjemtablespace` SQL script. You must provide the `SYS` username and password and the `TNSENTRY` of this database where the data bus is created.
- `createjem`: To install and create the data bus, execute the `createjem` SQL script. This requires the `SYS` username and password, `TNS_ENTRY`, and an AC4J client username.
- `createclient`: To create another client on an existing data bus, execute the `createclient` SQL script. Provide the `SYS` username and password, client username and password, and client tablespace.
- `recreatedatabus`: To re-create an existing data bus, which deletes the existing data bus and all its contents and then re-creates it, execute the `recreatedatabus` script. Provide the `SYS` username and password and `TNSENTRY` of the database where the data bus resides.
- `recreateclient`: To re-create an existing client, execute the `recreateclient` SQL script. Provide the SYS username and password and the client username and password.

AC4J Data Bus XML Configuration

The interaction supports JTA global transactions within the database that the data bus exists in. Thus, you need a nonemulated data source for the superuser to handle the two-phase commit, and a nonemulated data source for the client to send its asynchronous requests to the data bus. See the DataSource and JTA Chapters in the *Oracle9iAS Containers for J2EE Services Guide* for a full description of this configuration.

For our purchase order example, the following data sources are configured in the `data-sources.xml` file for the two-phase commit.

```
<!--NON-Emulated DataSource for two-phase commit used by super user-->
<data-source
                class="com.evermind.sql.OrionCMTDataSource"
```

```
                location="jdbc/jemSuperuserDS"
                username="jemuser"
                password="jempasswd"
                url="jdbc:oracle:thin:@host:port:ORCL-SID"
                inactivity-timeout="60" >
                <property name="dblink"
                    value="JEMLOOPBACKLINK.REGRESS.RDBMS.DEV.US.ORACLE.COM" />
</data-source>

<!--NON-Emulated DataSource for the client user -->
<data-source
                class="com.evermind.sql.OrionCMTDataSource"
                location="jdbc/jemClientDS"
                username="jemcliuser"
                password="jemclipasswd"
                url="jdbc:oracle:thin:@host:port:ORCL-SID"
                inactivity-timeout="60" >
                <property name="dblink"
                   value="JEMLOOPBACKLINK.REGRESS.RDBMS.DEV.US.ORACLE.COM" />
</data-source>
```

Both of these users were created as defaults with the SQL scripts listed earlier. The `jemuser` is the superuser username, and the `jemcliuser` is the default client username. The `DBLINK` is the link to the database that contains the data bus. For the superuser data source, the `DBLINK` is a loopback link.

AC4J Example

AC4J is designed for complex applications that interact with each other over long periods of time. This section illustrates the usage of AC4J with a portion of the purchase order example shown in Figure 10–6. To simplify the example, the code sample does not show error handling or import statements. Download the full example off the OTN site at `http://otn.oracle.com/sample_code/tech/java/oc4j/htdocs/oc4jsamplecode/oc4j-demo-ejb.html`.

Example 10–1 Purchase Order Example

For the purchase order, the `POInteraction` is created. Within the interaction, several business tasks exist as follows:

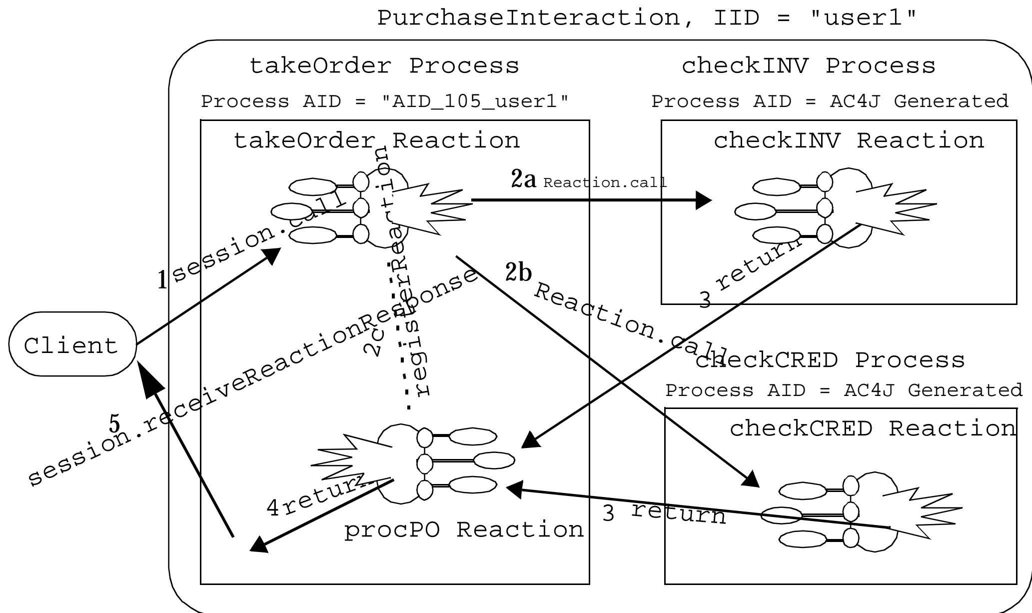
- Create purchase order (the `takeOrder` process)
- Check inventory (the `checkINV` process)
- Check customer credit (the `checkCRED` process)
- Process responses from previous checks requests (the `procPO` reaction)

The example includes the following:

- The `takeOrder` reaction, which pushes two data tokens:
  - A data token that asks the `checkINV` process if the inventory contains the desired items.
  - A data token that asks the `checkCRED` process if the credit card given by the customer is able to make the purchase.
- The `procPO` reaction, which acts on the responses from the inventory and credit check processes. If the inventory is available and the credit check goes well, then the `procPO` returns the purchase order confirmation to the client.

Figure 10–6 illustrates the information flow inside an interaction. Figure 10–6 also demonstrates how all of the reactions act on data tokens and provide data tokens to other processes. This assumes that the customer data has already been made available to the `takeOrder` process. The numbers designate the order in which the reactions fire. That is, the `procPO` is dependent on data tokens from both the `checkINV` and `checkCRED` processes; thus, it cannot fire until both return their responses back to the `takeOrder` process.

Figure 10–6 Information Flow Inside An Interaction



Here is a summary of the steps in processing the Purchase Order example of Figure 10–6:

1. *Client sends an asynchronous request to an Active EJB:* The client requests a service from an Active EJB, `JEMPurchaseOrderBean`. The client starts a new purchase order by sending an asynchronous request through the data bus (not shown) to a `takeOrder` process.
2. *Active EJB processes the client's request:* The `takeOrder` process starts a `takeOrder` base reaction. This base reaction starts a new purchase order. To complete the purchase order, it must perform three things:
   a. Send an asynchronous request to the `checkINV` process of `JEMInventoryBean` to verify that the items are in inventory
   b. Send an asynchronous request to the `checkCRED` process of `JEMCreditBean` to verify that the customer's credit is satisfactory
   c. Register a `procPO` reaction in the current process to receive the results from the preceding two processes

3. *Asynchronous response to the requesting Active-EJB:* Both the `checkINV` and `checkCRED` processes return responses to the `takeOrder` process.
4. *Asynchronous response to the client:* The `procPO` reaction, within the `takeOrder` process, reacts to the information provided by the `checkINV` and `checkCRED` processes. If satisfactory, the `procPO` reaction sends the confirmation to the client through the AC4J data bus.
5. *Client receives the response:* The client retrieves the response from the data bus.

Asynchronous Request to An Active EJB

The following code sample shows the steps in performing loosely coupled interactions in AC4J.

Example 10–2 Client Asynchronously Invoking Active EJB

```
public static void main(String[] args) throws  ClassNotFoundException, Exception
{
    // 0. create a JNDI context
    Context  context = new  InitialContext();

    // 1. look up a datasource where Databus exists
    DataSource clientDS = (DataSource)
                          context.lookup("java:comp/env/jdbc/jemClientDS");
    // 2. Get a JDBC-connection to the database where Databus resides
    Connection conn = clientDS.getConnection("jemcliuser", "jemclipasswd");

    // 3. Create an AC4J connection using the JDBC connection
    JEMConnection AC4JConn = new  JEMConnection(conn);

    // 4. Create an AC4J session over an AC4J connection to the Databus
    JEMSession AC4JSess = new  JEMSession(AC4JConn);
    // 5. Look up the Active EJB handle using the jem-name defined
    //    in the orion-ejb-jar.xml
    JEMHandle activeEJBHandle =
    (JEMHandle)context.lookup("JEMPurchaseOrderBean");

    // 6. Gather the base Reaction input parameters. These input parameters are
    //    required by the receiving method, takeOrder.
    Object[] inputParams = new  Object[] { (Object) new  String("user1"),
                                           (Object) new  Integer("1234-119"),
                                           (Object) new  String("pens"),
                                           (Object) new  Integer("3")   };
```

```
    // 7. Create the Process Context, Interaction-ID and Activation ID.
    //    NOTE: IID = "user1" = requester's name
    //          AID = AID = AID_105_user1 = AID_<PO_number>_<cust_name>

    // 8. Make the call over the AC4J session providing the parameters.
    JEMEmitToken req = AC4JSess.call("user1", "AID_105_user1",
                                     activeEJBHandle, "takeOrder",
                                     null, inputParams, null, 0, 0);

    // 9. Commit the changes to the Databus by committing the transaction
    conn.commit();

    // 10. The client must close the AC4J session and connection because it
    //     does not exist within an AC4J container, which would normally
    //     close these.
    conn.close();
    AC4JConn.close();
    jemsess.close();
}
```

The client exists outside of an AC4J server and is requesting a service from an Active EJB through the AC4J data bus. The AC4J data bus is the conduit and controls the asynchronous communication between the client and all reactions. Thus, every client residing outside of an AC4J server must first connect to the AC4J data bus and create a new session for interaction to occur.

After you have retrieved a connection to the AC4J data bus and created an AC4J session within it, you can send asynchronous messages to Active EJBs in the same or other AC4J instances. The AC4J data bus coordinates the asynchronous messages and acts as a transactional manager for all AC4J beans in the transaction. "Connect to the AC4J Data Bus" on page 10-26 describes the steps in creating an AC4J-session and completing the client's request.

Connect to the AC4J Data Bus

The following steps detail how to create an AC4J session on the AC4J data bus. These steps are a subset of the steps (those numbered 0 to 4) contained in Example 10–2.

1. Retrieve an AC4J connection.

   An AC4J connection exists above a JDBC connection. Perform the following:

   a. Retrieve the `DataSource` defined for the database acting as the AC4J conduit. The `DataSource` you use should be defined in the `data-`

sources.xml file as a nonemulated data source with a <dblink> defined to the database where the AC4J data bus resides. See "AC4J Data Bus XML Configuration" on page 10-21 for more information.

```
Context context = new InitialContext();
DataSource clientDS = (DataSource)
               context.lookup("java:comp/env/jdbc/jemClientDS");
```

b. Retrieve the JDBC connection off of the DataSource object.

```
OracleConnection conn = (OracleConnection)clientDS.getConnection();
```

c. Create an AC4J connection off of the JDBC connection object.

```
JEMConnection AC4JConn = new JEMConnection(conn);
```

2. Create an AC4J session in a specified data bus. Using the AC4J connection to the database and providing the name of the data bus you are interested in, create a session within the data bus in the indicated Oracle database.

```
JEMSession AC4JSess = new  JEMSession(AC4Jconn);
```

Executing an Asynchronous Request

After you have created an AC4J session on the AC4J data bus, the client can send asynchronous messages to Active EJBs. The client must provide the Active EJB handle, the process handle, and all the required input parameters to the base reaction. The following steps explain the details of the call that the client must make to complete the AC4J request.

1. *Process Context:* To identify the context where the process exists, you must provide both the interaction identifier and the process activation identifier. The combination of both of these identifiers is the processing context. There are two ways of providing a processing context:

- CLIENT PROVIDES—The AC4J data bus uses the identifiers provided by the client to uniquely identify the processing context. The client uses the same identifiers to either retrieve the response to the current request or send additional parameters to the process. In the current example, the client supplies the interaction identifier (IID) as a customer's name, and process activation identifier (P-AID) as a union of the purchase order number and the customer's name, as shown:

```
String iid = "user1";         // = customer_name
String p_aid = "AID_105_user1"; // = AID_<PO_number>_<customer_name>
JEMEmitToken req = AC4JSess.call(iid, p_aid, ..all other
```

```
parameters..);
```

- AUTOMATIC CONTEXT—The interaction and process activation identifiers are optional and can be omitted or can be null, in which case the system automatically creates them. If a client fails to provide either of these identifiers, then the AC4J data bus creates them to uniquely identify a processing context. However, the client has to retrieve these identifiers and use them later to pull the response from the AC4J data bus.

```
JEMEmitToken req =
         AC4JSess.call (null, null, ...all other parameters...);
JEMPortHandle portHandle = req.getPortHandle();
String iid = portHandle.getIid();
String p_aid = portHandle.getAid();
```

2. *Active EJB handle:* In a synchronous EJB environment, you would use a remote EJB handle for invocation. In an AC4J asynchronous environment, you must provide a similar handle of class type `JEMHandle` that identifies an active EJB. OU can get the active EJB handle by looking up the `jem-name` defined in the `orion-ejb-jar.xml` file (see “AC4J Active EJB Deployment” on page 10-38).

```
Context context = new InitialContext();
JMHandle activeEJBHandle =
          (JEMHandle) context.lookup("JEMPurchaseOrderBean");
JEMEmitToken req = AC4JSess.call(...., activeEJBHandle, ....);
```

3. *Reaction name and input parameters:* Client provides the base reaction (method) name and all or part of its input parameters that it wishes to call. In the current example, the client provides all the input parameters to complete the AC4J session call as follows:

```
// collect input values for the takeOrder method
Object[] inputParams = new Object[] { (Object) new String("user1"),
                                     (Object) new Integer("1234-123"),
                                     (Object) new String("pens"),
                                     (Object) new Integer("3")
                                   };
JEMEmitToken req = AC4JSess.call(..., "takeOrder", null, inputParams,
                                 ...);
```

- If the client provides only some of the parameters to this reaction, then it must supply a set of input parameter types, and the indexes of the input parameters as well. The following example shows how a client can

complete a call by furnishing the first two parameters for the `takeOrder` process:

```
// input parameter types (java-class) for takeOrder method
Class[] takeOrderInputClassTypes =
            new Class[] { String.class, Integer.TYPE,
                          String.class, Integer.TYPE };

// indexes of input parameters you wish to provide for takeOrder method
int[] indexOfInputParams = new int[] {0, 1};

// input values corresponding to the indexes for takeOrder method
Object[] inputParams = new Object[] { (Object) new String("user1"),
                                        (Object) new Integer("1234-123")
                                      };
// remember the interaction and process-activation ids for this call
JEMEmitToken req =
       AC4JSess.call(iid, p_aid, ..., "takeOrder",
                            takeOrderInputClassTypes,
                            indexOfInputParams, inputParams, ...);
```

- When the client provides the remaining two parameters, it must use the same process context (interaction and process activation identifiers) that it used in the first call it made to the process. During the second invocation the steps are:

```
// input parameter types (java-class) for takeOrder method
Class[] takeOrderInputClassTypes =
            new Class[] { String.class, Integer.TYPE,
                          String.class, Integer.TYPE };

// indexes of input parameters you wish to provide for takeOrder method
// NOTE: now, client provides the last 2-input parameters
int[] indexOfInputParams = new int[] {2, 3};

// input values corresponding to the indexes for takeOrder method
Object[] inputParams = new Object[] { (Object) new String("pens"),
                                        (Object) new Integer("3")
                                      };
// use the same interaction and process activation ids as those in the
// previous call
JEMEmitToken req =
       AC4JSess.call(iid, p_aid, ..., "takeOrder",
                            takeOrderInputClassTypes,
```

```
                                    indexOfInputParams, inputParams, ...);
```

4. *AC4J Session call:* Send all asynchronous requests for any Active EJB to the AC4J Session, using the `JEMSession::call` method.

   When a reaction wants to provide data to an active EJB method (to the base reaction of the process), it executes a `JEMSession::call` with this information. The `JEMSession::call` contains the interaction identifier of the EJB, the process activation identifier to identify the process where the method is instantiated, and the `JEMHandle` of the active EJB. The interaction and process activation identifier are optional and can be omitted or can be null, in which case the system automatically creates them. The data bus identifies the context of the process and routes the data tokens to the intended process. Thus, all EJB calls are invoked asynchronously, through the mediation of the data bus.

5. *Commit Transaction:* The client must commit the changes to the AC4J data bus. If the client forgets to commit the transaction, then the request is lost and is not visible to the AC4J data bus. To make the request visible to the AC4J data bus, perform the JDBC commit as follows:

   ```
   conn.commit();
   ```

6. Finally, the client must close the JDBC connection, the AC4J session, and the connection, because the client does not exist within an AC4J container. The AC4J container normally closes the AC4J session and connection objects.

   ```
   conn.close(); // client as well an application-code must close
   AC4JConn.close(); // client must close
   jemsess.close(); // client must close
   ```

Active EJB Processes the Client's Request

Once the client commits the request, the AC4J data bus matches the data tokens provided by the client with those of the requested reaction, and internally schedules the instantiation of the `JEMPurchaseOrderBean` Active EJB and activation of the `takeOrder` process. The `takeOrder` process starts a `takeOrder` base reaction, which starts a new purchase order. As shown in Figure 7-6, this reaction, `takeOrder`, processes the client's request by invoking additional services from the other Active EJBs, `JEMInventoryBean` and `JEMCreditBean`. This is shown in the following code sample:

Example 10–3 Active EJB Asynchronously Invoking Another Active EJB

```
public void takeOrder(String clientName, int creditCardNumber,
```

```
                    String itemName, int quantity)
      throws RemoteException, TestException
{

    // 0. create a JNDI context
    Context context = new InitialContext();

    // 1. Retrieve the current AC4J Reaction.
    JEMReaction currentAC4JReaction = (JEMReaction) JEMReaction.getReaction();

    // 2. Look up the Active EJB handle using the jem-name defined
    //    in the orion-ejb-jar.xml
    JEMHandle activeInvHandle = (JEMHandle) context.lookup("JEMInventoryBean");

    // 3. Gather all input and return parameters for the checkINV Reaction.
    //    Define input and return parameter types and the parameter values
    Object[] checkINVInputParamValues =
                new Object[] { (Object)itemName,
                               (Object) new Integer (quantity) };
    Class[] checkINVReturnClassType = new Class[] { Boolean.TYPE };

    // 4. Request a service from JEMBeancheckINV through Databus
    JEMEmitToken inventoryRequest=
                currentAC4JReaction.call(activeInvHandle, "checkINV", null,
                                         checkINVInputParamValues,
                                         checkINVReturnClassType,
                                         null, null, 0,  0);
    // 5. Repeat Steps 2-4 above to request a service from another
    //    Active EJB, JEMCreditBean. The returned JEMEmitToken is
    //    named creditRequest.

    // 6. Register a Reaction, procPO, that will be activated when the
    //    responses from the above two asynchronous calls to the
    //    active-EJBs return
    Class[] procPOInputClassTypes = new  Class[] { Boolean.TYPE, String.class };
    JEMEmitToken[] requests = new JEMEmitToken[] { inventoryRequest,
                                                   creditRequest };
    currentJEMReaction.registerReaction
             ("procPO", procPOInputClassTypes, requests, 1, null, null, 0);
}
```

The AC4J data bus instantiates the Active EJB, `JEMPurchaseOrderBean` (corresponding to the `JEMHandle` provided by the client), in an AC4J server. The

takeOrder process starts a takeOrder base reaction. Here are the steps in the completion of this initiation process:

1. *Process Context:* The current reaction, takeOrder, is running in an AC4J server. Therefore, it already has a process context and can be used by the application (or Active Bean) code. The application code can retrieve the process context through the demarcation, as follows:

```
// retrieve the current-Reaction context--a static method
JEMReaction currentAC4JReaction = (JEMReaction) JEMReaction.getReaction();
String iid = currentAC4JReaction.getIid();
String p_aid = currentAC4JReaction.getAid();
```

   – The application may use these identifiers to make additional asynchronous JEMSession::call calls by co-relating the business transaction.

   – Alternatively, the application code can use the currentAC4JReaction to make the additional calls with request-response characteristics. The AC4J data bus then creates a new process context for the next invocation by using the current interaction identifier and a new process activation identifier. The current example uses this approach with the currentAC4JReaction.

2. *AC4J handle:* The base reaction, takeOrder, starts the purchase order initiation process by requesting services from two other Active EJBs, JEMInventoryBean and JEMCreditBean. The application code must retrieve the AC4J handles to these Active EJBs by doing the following:

```
Context context = new InitialContext();
// call to JEMInventoryBean
JEMHandle activeInvHandle = (JEMHandle) context.lookup("JEMInventoryBean");
JEMEmitToken inventoryRequest=
                currentAC4JReaction.call(activeInvHandle, .....);

// call to JEMCreditBean
JEMHandle activeCreditHandle = (JEMHandle) context.lookup("JEMCreditBean");
JEMEmitToken creditRequest=
                currentAC4JReaction.call(activeCreditHandle, .....);
```

3. *Reaction name, return parameter type, and input parameters:* The client (now a takeOrder reaction) provides the base reaction (method) name, the java-class type of the return parameter and all or part of its input parameters that it wishes to call. In the current example (which started at “Connect to the AC4J Data Bus” on page 10-26), the client provides all the input parameters needed by the called reactions (checkINV, CheckCRED) as follows:

```
// collect input values for the checkINV method
```

```
Object[] checkINVInputParamValues =
            new Object[] { (Object)itemName,
                           (Object) new Integer (quantity)
                         };

// state the return Class type of checkINV method
Class[] checkINVReturnClassType = new Class[] { Boolean.TYPE };

// make the call to the checkINV method
JEMEmitToken inventoryRequest=
               currentAC4JReaction.call(..., "checkINV", null,
                                        checkINVInputParamValues,
                                        checkINVReturnClassType, ....);

// collect input values for the checkCRED method
Object[] checkCreditInputParamValues =
            new Object[] { (Object) clientName,
                           (Object) new Integer (creditCardNumber),
                           (Object) new Float (quantity * 1.4) };

// state the return Class type of checkINV method
Class[] checkCreditReturnClassType = new Class[] { String.class };

// make the call ro checkCRED method
JEMEmitToken creditRequest=
               currentAC4JReaction.call(..., "checkCRED", null,
                                        checkCreditInputParamValues,
                                        checkCreditReturnClassType, ....);
```

4. *Register a return reaction:* The application code then registers a new reaction, `procPO`, in the same process context of the `currentAC4JReaction`. This registration of the reaction requires the reaction name, `procPO` in this example, the input parameter types of the new `procPO` reaction, and the `JEMEmitTokens` retrieved from the call to the `currentAC4JReaction`. If the new reaction has multiple input parameters and is receiving them from different processes, then the Array of `JEMEmitToken` must be constructed in proper order. For example, in the following code the first parameter (`"procPO"`) is waiting for the reply from the `JEMInventoryBean`, and the second one (`procPOInputClassTypes`) is waiting for the reply from `JEMCreditBean`.

```
Class[] procPOInputClassTypes = new  Class[] { Boolean.TYPE, String.class };
JEMEmitToken[] requests = new JEMEmitToken[] { inventoryRequest,
                                               creditRequest };
```

```
    currentJEMReaction.registerReaction
                ("procPO", procPOInputClassTypes, requests, 1, null, null, 0);
```

Asynchronous Response to the Requesting Active EJB

The `takeOrder` base reaction is completed only after the AC4J infrastructure commits the transaction that includes the calls to the other two Active EJBs and a registered reaction. The `checkINV` and `checkCRED` processes receive the requests from the AC4J data bus as if they were invoked from any other EJB. The `JEMInventoryBean` and `JEMCreditBean` Active EJBs are instantiated. The `checkINV` and `checkCRED` base reactions are fired when they receive the data tokens from the AC4J data bus, which were initiated from the `takeOrder` reaction. Both of them receive the request, perform their tasks, and return. The returned values are forwarded by the AC4J data bus to the registered reaction—`procPO`.

The code sample in Example 10–4 shows the `checkINV` method. The `checkCRED` method is similar in its AC4J responsibilities.

Example 10–4 checkINV Processes Request

```
public boolean checkINV(String itemName, int quantity)
            throws RemoteException, TestException
{
    boolean inventoryExists = false;
    // The logic in the next step is ommitted
    inventoryExists = query its own database for the item and quantity;
    return inventoryExists;
}
```

Asynchronous Response to the Client

Both the `checkINV` and `checkCRED` processes return their responses to the `procPO` reaction through the AC4J data bus. The AC4J data bus makes sure that the return data-tokens have valid `takeOrder` process context and matches the input parameter types of the `procPO` reaction. When both parameters arrive, the `procPO` reaction fires and executes the `procPO` method of the `JEMPurchaseOrderBean` Active EJB, which reacts to the information provided by the `checkINV` and `checkCRED` processes. It completes the client's request by posting the result to the AC4J data bus. The code sample in Example 10–5 shows the `procPO` method.

Example 10–5 procPO Reaction Fires

```
public String procPO(boolean inventoryExists, String creditInfo)
            throws RemoteException, TestException
```

```
{
    String poStatus = "Not Shipped";
    if(creditInfo == null)
        return poStatus;
    if (inventoryExists)
    {
        if(creditInfo.equalsIgnoreCase("Credit approved"))
            poStatus = "Shipped";
        else if (creditInfo.equalsIgnoreCase("Credit failed"))
            poStatus = "Credit failed";
    }
    else
        poStatus = "Items unavailable";

    return poStatus;
}
```

Response from the Client

The client needs to know the response to its purchase order request. As stated earlier, each request (or call) is identified by a process context (interaction ID and activation ID). Using the process context, the client can pull the response from the AC4J data bus.

The received `JEMEmitToken` from the response can then be parsed by the client. If the client existed inside the OC4J container, the container would deconstruct the `JEMEmitToken` to the required type. Instead, the client must parse out the response correctly, as shown below:

Example 10–6 Client Processes Return

```
public static void main(String[] args) throws  ClassNotFoundException, Exception
{
    // 0. create a JNDI context
    Context  context = new  InitialContext();

    // 1. Look up a data source where the databus exists
    DataSource clientDS = (DataSource)
                          context.lookup("java:comp/env/jdbc/jemClientDS");

    // 2. Get a JDBC-connection to the database where Databus resides
    Connection conn = clientDS.getConnection("jemcliuser", "jemclipasswd");

    // 3. Create an AC4J connection using the JDBC connection
```

```
    JEMConnection AC4JConn = new  JEMConnection(conn);

    // 4. Create an AC4J session over an AC4J connection to the Databus
    JEMSession AC4JSess = new  JEMSession(AC4JConn);

    // 5. Look up the Active EJB handle using the jem-name defined
    //    in the orion-ejb-jar.xml
    JEMHandle activeEJBHandle =
              (JEMHandle) context.lookup("JEMPurchaseOrderBean");

    // 6. Retrieve the Response using the Process context with which
    //    the initial request was made.
    JEMEmitToken  rcvresp = AC4JSess.receiveReactionResponse
            ("user1", "AID_105_user1", activeEJBHandle, "takeOrder", 0);

    // 7. The getReactionResponseObjectInstance method parses the returned
    //    parameter into an java.lang.Object.
    Object obj = rcvresp.getReactionResponseObjectInstance();
    // 8. Print out results
    if (obj  instanceof  java.lang.String)
        String  ret = (String)  obj;

    // 9. The client must commit the transaction
    conn.commit();

    // 10. The client must close the AC4J session and connection because it
    //     does not exist within an AC4J container, which would normally
    //     close these.
    conn.close();
    jemsess.close();
    AC4JConn.close();
}
```

As seen earlier, the `procPO` reaction reacts to the information provided by the `checkINV` and `checkCRED` processes. It completes the client's request by posting the result to the AC4J data bus. The client must connect to the AC4J data bus to retrieve its response by providing a proper process context. The steps in connecting to the AC4J data bus were described in Example 10–3. After receiving the response, the client can retrieve a `java.lang.Object` instance that must be processed further.

Retrieving an Asynchronous Response

After creating an AC4J session on the AC4J data bus, the client can retrieve the response by performing the following steps:

1. *Process Context:* The client must provide a proper process context that identifies where the request was made, and both the interaction identifier and the process activation identifier. In the example under discussion, the client provides the interaction Identifier (IID) as a customer's name and process activation identifier (P-AID) as a union of purchase order number and the customer's name, as shown:

```
String iid = "user1";         // = customer_name
String p_aid = "AID_105_user1"; // = AID_<PO_number>_<customer_name>
JEMEmitToken  rcvresp = AC4JSess.receiveReactionResponse
          (iid, p_aid, ...);
```

2. *Active EJB handle:* The client must supply the Active EJB handle to which the initial request was made. The Active EJB handle can be obtained by looking up the `jem-name` defined in the `orion-ejb-jar.xml` file (see "AC4J Active EJB Deployment" on page 10-38).

```
Context context = new InitialContext();
JMHandle activeEJBHandle =
          (JEMHandle) context.lookup("JEMPurchaseOrderBean");
JEMEmitToken  rcvresp = AC4JSess.receiveReactionResponse
          (..., activeEJBHandle, ...);
```

3. *Reaction Name:* The client may need to provide the process name to which it initiated the call, which, in this case, is the `takeOrder` process.

```
JEMEmitToken  rcvresp = AC4JSess.receiveReactionResponse
          (..., "takeOrder", ...);
```

4. *Retrieve Object:* The `JEMEmitToken` received from the `receiveReactionResponse` can be used to retrieve the Object instance, as follows:

```
Object obj = rcvresp.getReactionResponseObjectInstance();
```

5. *Commit Transaction:* The client must commit the changes to the AC4J data bus. If the client forgets to commit the transaction, then the client can pull the response multiple times. However, we do not a recommend this mode of operation. To let the AC4J data bus know that the response was properly retrieved, perform the following:

```
conn.commit();
```

AC4J Active EJB Deployment

The active EJB is developed as any other EJB. The changes that enable the EJB to be used in an AC4J interaction are in the OC4J-specific deployment descriptor. These are discussed below:

Deploy the EJB with AC4J element specifications in the OC4J-specific deployment descriptor. The following example defines the `takeOrder` EJB as an active EJB.

- The `<jem-server-extension>` element defines the database with the data bus that the active EJBs in this JAR file use for their AC4J communication.

  ```
  <jem-server-extension data-source-location="jdbc/jemSuperuserDS">
    <description>AC4J datasource location</description>
  </jem-server-extension>
  ```

- The `<jem-deployment>` element in the `orion-ejb-jar.xml` file identifies the EJB defined in the `ejb-jar.xml` file as an active EJB. This element provides an AC4J name (`jem-name`) that is used to identify the bean within the AC4J calls. For example, this bean is defined as `JEMPurchaseOrderBean`, which was used in the `JEMHandle` creation. The identity of the caller, which is allowed to request services and retrieve responses from the Active EJB, can be declared in the `called-by` tag. This `caller` tag identifies the user in the data bus. For example, `JEMCLIUSER` is the user name that was used to create a `jem-session`,

  ```
  <jem-deployment jem-name="JEMPurchaseOrderBean"
                  ejb-name="PurchaseOrderBean">
    <description>AC4J EJB</description>
    <called-by>
      <caller caller-identity="JEMCLIUSER"/>
    </called-by>
  </jem-deployment>
  ```

The following is the entire `orion-ejb-jar.xml` file for the three Active EJBs.

```
<?xml version="1.0"?>
<!DOCTYPE orion-ejb-jar PUBLIC "-//Evermind//DTD Enterprise JavaBeans 1.1
runtime//EN" "http://www.orionserver.com/dtds/orion-ejb-jar.dtd">

<orion-ejb-jar deployment-version="1.4.5" deployment-time="e60dffcea9">
<enterprise-beans>
    <jem-server-extension
        data-source-location="jdbc/nonEmulatedDS"
        scheduling-threads="1">
        <description>AC4J deployment</description>
```

```
    </jem-server-extension>

    <jem-deployment jem-name="JEMPurchaseOrderBean"
                    ejb-name="PurchaseOrderBean">
        <description>Active Purchase Order bean</description>

        <called-by>
            <caller  caller-identity="JEMCLIUSER"/>
        </called-by>

        <security-identity>
            <description>using the caller identity </description>
            <use-caller-identity>true</use-caller-identity>
        </security-identity>
    </jem-deployment>

    <jem-deployment jem-name="JEMInventoryBean"
                    ejb-name="InventoryBean">
        <description>Active Inventory bean</description>

        <called-by>
            <caller  caller-identity="JEMCLIUSER"/>
        </called-by>

        <security-identity>
            <description>using the caller identity </description>
            <use-caller-identity>true</use-caller-identity>
        </security-identity>
    </jem-deployment>

    <jem-deployment jem-name="JEMCreditBean"
                    ejb-name="CreditBean">
        <description>Active Credit bean</description>

        <called-by>
            <caller  caller-identity="JEMCLIUSER"/>
        </called-by>

        <security-identity>
            <description>using the caller identity </description>
            <use-caller-identity>true</use-caller-identity>
        </security-identity>
    </jem-deployment>

</enterprise-beans>
```

Administering AC4J

The remainder of this chapter shows the structure of the AC4J data bus within an Oracle database and refers to administering OC4J to support AC4J.

Administering Oracle Databases to Support AC4J

The `createjem` script creates a complete JEM repository in the database; the repository is administered by a PL/SQL package called `JEMDatabus`.

Note: `JEM` is an internal name that is equivalent to `AC4J`.

The complete AC4J packages are shown in the Javadoc, which can be found on the documentation CD accompanying this product or on OTN.

Description of the JEM PL/SQL package

Under the JEMUSER schema the following package is created:

```
create or replace package  JEMDatabus
procedure  setDatabusProperties(dfbusname  IN VARCHAR2,
                tokttldiff      NUMBER,
                tokttlcalldiff  NUMBER,
                tokttldatdiff  NUMBER,
                rxnttldiff      NUMBER,
                gccycle         NUMBER);

procedure  setTokttldiff(
                dfbusname  IN VARCHAR2,
                tokttldiff      NUMBER);

procedure  setTokttlcalldiff(
                dfbusname  IN VARCHAR2,
                tokttlcalldiff       NUMBER);

procedure  setTokttldatdiff(
                dfbusname  IN VARCHAR2,
                tokttldatdiff      NUMBER);

procedure  setTokclnttldiff(
                 dfbusname  IN VARCHAR2,
                tokclnttldiff      NUMBER);
```

```
procedure  setRxnttldiff(
                dfbusname  IN VARCHAR2,
                rxnttldiff       NUMBER);

procedure  setGCcycle(
                dfbusname  IN VARCHAR2,
                gccycle       NUMBER);


procedure  createDatabusTpc(
                 dfbusname                    IN     VARCHAR2  DEFAULT NULL,
                 description                  IN     VARCHAR2  DEFAULT NULL,
                 max_retries                  IN     NUMBER    DEFAULT 1,
                 retry_delay                  IN     NUMBER    DEFAULT 0,
                  retention_time              IN     NUMBER    DEFAULT 0,
                 gccycle                      IN     NUMBER    DEFAULT 10000);

procedure  dropDatabusTpc(dfbusname  IN VARCHAR2  DEFAULT NULL);

procedure  createAppGroupSubscriber(
                 dbusname   IN     VARCHAR2,
                 subname    IN     VARCHAR2 DEFAULT  'JEMSUB');

procedure  dropAppGroupSubscriber(
                 dbusname  IN     VARCHAR2,
                 subname   IN     VARCHAR2 DEFAULT  'JEMSUB');

end  JEMDatabus;
```

Description of the *createDatabusTpc* Package Public Method

The following PL/SQL procedure creates a databus:

```
procedure  createDatabusTpc(
             dfbusname              IN     VARCHAR2  DEFAULT NULL,
             description            IN     VARCHAR2  DEFAULT NULL,
             max_retries            IN     NUMBER    DEFAULT 1,
             retry_delay            IN     NUMBER    DEFAULT 0,
             retention_time         IN     NUMBER    DEFAULT 0,
             gccycle                IN     NUMBER    DEFAULT 10000);
```

Description of the *dropDatabusTpc* Package Public Method

The following PL/SQL procedure drops a databus:

```
procedure  dropDatabusTpc(dfbusname  IN VARCHAR2  DEFAULT NULL);
```

Description of the JEM Schema Objects

When you create a databus, the following schema objects are created under the JEMUSER schema:

- Database tables
- Advanced Queueing queues
- Database views

The process also creates unique sequence-numbers and indexes for these tables. It creates multi-consumer queues using DBMS_AQADM package. All the views are created with a `public-synonym` and are granted to view for public.

Table 10–1 Database Tables

Table Name	Description
TABDFB + `dfbusname`	data bus table
TABPRS + `dfbusname`	process table
TABRXN + `dfbusname`	reaction table
TABRTL + `dfbusname`	match-tuple (reaction template) table
TABTOK + `dfbusname`	token table (active-data)
TABTRK + `dfbusname`	tracking table

In Table 10–1, the table name consists of the characters in the Table Name column, with an appended name, represented by `dfbusname`. This appended name is provided by the user that creates the data bus; the default is the empty string. For example, if the user provides the string `_EDB`, then the data bus table would be named `TABDFB_EDB`; if the user makes no specification when creating the data bus, then the data bus table would simply be named `TABDFB`.

The remainder of this section describes the tables created.

Data Bus Table

```
Name         = "TABDFB" + dfbusname
          PRIMARY KEY = Instance-ID (or Oracle SID)

Column-Name       Column-Type
```

```
--------------------------------------------
 --SID of the database the Databus resides in
 instid          VARCHAR2(50)  PRIMARY KEY,

 --version of the Databus
 version         VARCHAR2(20),

 --Databus Description
 description     VARCHAR2(2000),

 --Latest Databus version number
 scn             NUMBER,

 sysiid          VARCHAR2(50),   --System generated unique ids
 sysaid          VARCHAR2(150),  --System generated unique ids
 glbiid          VARCHAR2(50),   --System generated unique ids
 glbaid          VARCHAR2(150),  --System generated unique ids
 ixnaid          VARCHAR2(150),  --System generated unique ids

 --Default Retry number of a Reaction
 maxrxnretry     NUMBER,

 --State of the Databus (not used now):
--OPEN:
--SUSPENDED:
 state           VARCHAR2(30),

 --creation date of this DATABUS
 creatdate       VARCHAR2(30),

 --The following Data Token types can exist in the Databus:
 --CALL   =created as a result of a Session/Reaction call operation
 --STORE  =created as a result of a Reaction storeData operation
 --RET    =created as a result of a Reaction return result operation
 --EXC    =created as a result of a Reaction throw exception operation
 --SND    =created as a result of a Reaction sendData operation
 --SCHED  =System token, used for triggering a scheduling of a Reaction firing
 --REMATCH=System token, used for triggering the garbage collector to
--begin scanning
 --        for Processes, Reaction and Data Tokens to make unavailable
 --        and then potentially clean.
 --        The garbage collector does the following every 'gccycle' period:
 --           CompactTabTokens:
 --             Mark as 'INVALID' versions of all Tokens that have passed the
 --             TimeToLive (TTL) mark; no more reader/writer Reactions will be
```

```
-- able to be matched
 --             using this Data Token.
 --           GarbageCollectTokens:
 --             Delete all 'INVALID' versions of all Data Tokens when all
--reader/writer Reactions
 --             are gone (the reader reference counts and the exclusive
-- modification state of
 --             the Data Token are nulled) and the tokclnttl has passed
 --           GarbageCollectRxns:
 --             Delete all Reactions that are their state is UNMATCHED (no
-- suitable Data Tokens) or
 --             COMPLETED and the rxnttl has passed. Matched Reactions are not
-- deleted, since
 --             the Reaction firing can be scheduled must later of the Reaction
-- matching either
 --             because of resource unavailability or because of user directives
 --           GarbageCollectProcesses:
 --             Delete all Processes that encapsulate no Reactions
 --Default Time a Call Data Token is available for matching in the Databus
 --after it is stored in the Databus
 tokttlcalldiff  NUMBER,

 --Default Time a Return, Exception or Stream Data Token
 --is available for matching will live in the Databus
 --after it is stored in the Databus
 tokttldatdiff   NUMBER,

 --Default Time a Serialized Data Token is available for matching in the Databus
 --after it is stored in the Databus
 tokttldiff      NUMBER,

 --Default Time a Data Token is retained since having being unavailable for
--matching in the Databus
 tokclnttldiff   NUMBER,

 --Default Time a Reaction is available to be matched with available Data Tokens
 --in the Databus
 --after it is created in the Databus
 rxnttldiff      NUMBER,

 --How often the Databus garbage collector runs:
 gccycle         NUMBER,

 epoch           NUMBER,
 epochnext       NUMBER
```

```
The TABDBF databus table is initialized with the following values:
         version    =  "1.0"
         sysiid     =  "JEM$SYSIID"
         sysaid     =  "JEM$SYSAID"
         glbiid     =  "JEM$GLBIID"
         glbaid     =  "JEM$GLBAID"
         ixnaid            =  "JEM$IXNAID"
         state             =  "DFBST_INIT"
         dateformat =  "DD-MON-YYYY:HH:MI:SS"
```

Process Table

```
Name        = "TabPRS"+dfbusname
          PRIMARY KEY = (iid, aid)

Column-Name       Column-Type
--------------------------------------------
 --
prsseq              NUMBER,

 --interaction-ID; identifies the callee interaction
 iid                 VARCHAR2(50)  NOT NULL,

 --activation-ID; identifies the callee process activation
 aid                 VARCHAR2(150),

 --return IID; identifies the caller interaction
 retiid              VARCHAR2(50),

 --return AID; identifies the caller process activation
 retaid              VARCHAR2(150),

 --JEMHandle; identifies the callee Active EJB
 handle              BLOB,
 handlelen           NUMBER,

 --JEMHandle; identifies the caller Active EJB
 rethandle           BLOB,
 rethandlelen        NUMBER,

 --process-name; identifies the callee Active EJB method name of the base
--Reaction
 prspeid             VARCHAR2(100),
```

```
 --
 funpolytup          BLOB,
 funpolytuplen       NUMBER,

 --
 retpolytup          BLOB,
 retpolytuplen       NUMBER,

 excretpolytup       BLOB,
 excretpolytuplen    NUMBER,
 cmpltag             VARCHAR2(250),
 callindx            NUMBER,

 --Not used
 state               VARCHAR2(50),

 flags               NUMBER,

 --start time for the process
 prsstart            VARCHAR2(30),

 --end time for the process
 prsend              VARCHAR2(30),

 --user-ID of starter of the Process activation
 ixnoriginatorusrid  VARCHAR2(50),

 epoch               NUMBER


           - creates a unique index ("Idx1Prs"+dfbusname)
 on ("TabPRS"+dfbusname+prsseq)
```

Reaction Table

```
Name        = "TabRXN"+dfbusname
           PRIMARY KEY = (iid, aid, rid, recursid)

Column-Name       Column-Type
--------------------------------
 rxnseq           NUMBER,

 --interaction-ID; identifies the callee interaction
 iid              VARCHAR2(50)  NOT NULL,
```

```
 --activation-ID; identifies the callee process activation
 aid              VARCHAR2(150),

 --reaction-ID; identifies the reaction
 rid              VARCHAR2(100),
 recursid         NUMBER,

 --JEMHandle; identifies the callee Active EJB
 handle           BLOB,
 handlelen        NUMBER,

 --process-name; identifies the callee Active EJB method name of the base
--Reaction
 prspeid          VARCHAR2(100),

 --reaction name; identifies the callee Active EJB method name of the fired
--Reaction
 --prspeid is the same as rxnpeid for the base Reaction of a Process
 rxnpeid          VARCHAR2(100),

 --
 grpid            VARCHAR2(100),

 vid              NUMBER,

 --
 funpolytup       BLOB,
 funpolytuplen    NUMBER,

 --number of matching tuples (Data Tokens) that must be matched in order for the
--Reaction to be
 --marked as state=MATCHED
 totmattups       NUMBER,

 mattupsprops     BLOB,
 mattupspropslen  NUMBER,

 --retry count of the Reaction in case of a Reaction rollback; when this count
--reaches the
 --maximum attempts the Reaction is marked state=COMPLETED with status=max
--retries reached
 retrycnt         NUMBER,

 --Reaction states:
```

```
 --  UNMATCHED: when the Data Tokens needed for matching
 --             are not available (not arrived in the Databus, not visible yet,
--have conflict in
 --               the interest mode)
 --  MATCHED  : when the Data Tokens  needed for matching are available
 --  COMPLETED: when the Reaction commits or rollbacks with exception status
--message
 state           VARCHAR2(50),

 --Reaction statuses:
 --   PRSUNMATCHED:
 --   RXNDISCARDED:
 --   RXNMAXRETRY :
-- COMMITED:
 status          VARCHAR2(50),

 --Type of the Reaction:
 --  CALL : if it is a base Reaction (implicitly created by the Databus)
 --  MATCH: if it is not a base Reaction (explicitly created by the application)
 type            VARCHAR2(50),

 flags           NUMBER,

 --Reaction priority: Reactions registered in a Process when matched are firing
--ordered by the
 --time of registration if they have the same priority.
 --The firing ordering is:
 --    order by ReactionPriority descending, ReactionRegistrationTime ascending
 rxnpri          NUMBER,

 --time-to-live for the reaction
 rxnttl          VARCHAR2(30),

 --description of this reaction
 descr           VARCHAR2(2000),

 --date of the reaction registration as Julian date
 rxndate         NUMBER,

 --date of the reaction registration
 rxnstart        VARCHAR2(30),

 --date of the reaction commit
 rxnend          VARCHAR2(30),
```

```
 --user-ID of registered this reaction
 schemausrid      VARCHAR2(50),

 schemamsgid      VARCHAR2(50),

 auxctx           BLOB,
 auxctxlen        NUMBER,
 epoch            NUMBER,

 --Time a Call Data Token is available for matching in the Databus
 --after it is stored in the Databus
 tokttlcalldiff  NUMBER,

 --Time a Return, Exception or Stream Data Token
 --is available for matching will live in the Databus
 --after it is stored in the Databus
 tokttldatdiff   NUMBER,

 --Time a Serialized Data Token is available for matching in the Databus
 --after it is stored in the Databus
 tokttldiff      NUMBER,

 --Time a Data Token is retained since having being unavailable for matching
 --in the Databus
 tokclnttldiff   NUMBER,

 --Time a Reaction is available to be matched with available Data Tokens
 --in the Databus
 --after it is created in the Databus
 rxnttldiff      NUMBER


            - creates a unique index ("Idx1RXN"+dfbusname)
 on ("TabRXN"+dfbusname+"  "+(iid, aid, rid, recursid, prspeid, rxnpeid, grpid,
state))

            - create a unique index ("Idx2RXN"+dfbusname)
 'on ("TabRXN"+dfbusname+"  "+rxnseq)
```

Reaction Template Table

```
Name       = "TabRTL"+dfbusname

Column-Name       Column-Type
--------------------------------------------
```

```
 mattupseq     NUMBER,

 --every Reaction template needed for matching a Data Token has an index,
--starting from 0
 --This is used for a composite Reaction matching
 matindx       VARCHAR2(50)   NOT NULL,

 --1. First level of matching:
 -- The following 3 conditions need to be valid for a template to match with a
--Data Token
 --

 --1.1: Interaction-ID that needs to match with a Data Token's Interaction-ID
 tokiid        VARCHAR2(50),

 --1.2: Activation-ID that needs to match with a Data Token's Activation-ID
 tokaid        VARCHAR2(150),

 --1.3: tag name that needs to match with a Data Token's tag name
 tag           VARCHAR2(250),

 --
 vid           NUMBER,
 vidtype       VARCHAR2(50),

 --2. Second level of matching:
 --The filter conditions specified here need to be true for the propereties of
--the Data Token matched in the first level
 polycnd       CLOB,
 polycndlen    NUMBER,

 --Interaction-ID of the Reaction this template belongs to
 iid           VARCHAR2(50)  NOT NULL,

 --Activation-ID of the Reaction this template belongs to
 aid           VARCHAR2(150),

 --Reaction-ID of the Reaction this template belongs to
 rid           VARCHAR2(100),
 recursid      NUMBER,

 --java-class type of the object value that is needed for matching
 objclassname  VARCHAR2(1000),

 --Reaction template states:
```

```
 --  UNMATCHED: when the Data Token needed for matching
 --            is not available (not arrived in the Databus, not visible yet,
--have conflict in the interest mode)
 --  MATCHED  : when the Data Token needed for matching is available
 state         VARCHAR2(50),

 type          VARCHAR2(50),

 flags         NUMBER,

 --Same as Reaction priority
 rxnpri        NUMBER,

 timeout       NUMBER,

 --Same as Reaction rxnttl
 rxnttl        VARCHAR2(30),

 --Same as Reaction rxndate
 rxndate       NUMBER,

 --points to the Data Token when this template is MATCHED; otherwise null
 tokseq        NUMBER,

 --Scope Interaction-ID; used to minimize the matching process phase
 scpiid        VARCHAR2(1000),

 --Scope Activation-ID; used to minimize the matching process phase
 scpaid        VARCHAR2(1000),

 epoch         NUMBER


           - creates an index ("Idx1RTL"+dfbusname)
 on ("TabRTL"+dfbusname+"  "+(iid, aid, rid, recursid, state))
           - creates an index ("Idx2RTL"+dfbusname)
 on ("TabRTL"+dfbusname+"  "+(tokiid, tokaid, tag))
           - create a unique index ("Idx3RTL"+dfbusname)
 'on ("TabRTL"+dfbusname+"  "+mattupseq)
```

Token (Active Data) Table

```
Name        = "TabTOK"+dfbusname
PRIMARY KEY = sequence-number
```

```
Column-Name       Column-Type
--------------------------------------------
 --identifies uniquely this Data Token. A matched Reaction template
 --points to this id
 tokseq        NUMBER   PRIMARY KEY,

 --Data Tokens are chained:
 --seq-number of the previous version of the Data Token
 prevtokseq    NUMBER,

 --seq-number of the next version of the Data Token
 nexttokseq    NUMBER,

 aliastokseq   NUMBER,

 --Interaction-ID of this data-token
 iid           VARCHAR2(50)  NOT NULL,

 --Activation-ID of this data-token
 aid           VARCHAR2(150),

 --
 tag           VARCHAR2(250),
 classtag      VARCHAR2(10),

 --
 vid           NUMBER,
 prevvid       NUMBER,
 nextvid       NUMBER,

 --
 polycnd       CLOB,
 polycndlen    NUMBER,

 --
 objinst       BLOB,
 objinstlen    NUMBER,
 textinst      CLOB,
 textinstlen   NUMBER,

                undoentry     BLOB,
 undoentrylen  NUMBER,

 --used to describe the number of Reactions having matched a query interest
 --on this Data Token
```

```
 readers       NUMBER,

 consumer      VARCHAR2(100),

 --Data Token states:
 --  UNMATCHED: when the Data Token is not matched by a Reaction template or
--matched in query interest mode
 --  MATCHEDX  : when the Data Token is matched by a Reaction template in
--exclusive modification interest mode
 state         VARCHAR2(50),

 --Data Token statuses:
 --  VALID  : Tok is available; so readers can see it and a writer can reserve
it
 --  INVALID: Tok is unavailable because its time-to-live has expired; nobody
can
--access it
 status        VARCHAR2(50),


 --     CALL   :
 --     MATCH  :
 --     STORE  :
 --     RET    :
 --     EXC    :
 --     SND    :
 --     SCHED  :
 --     REMATCH:
 --     QUIT   :
 --
 op            VARCHAR2(50),

 reason        VARCHAR2(50),

 type          VARCHAR2(50),

 flags         NUMBER,

 --Time Data Token is available for matching in the Databus
 --after it is stored in the Databus
 tokttl        VARCHAR2(30),

 --Time a Data Token is retained since having being unavailable for matching
 --in the Databus
 tokclnttl     VARCHAR2(30),
```

```
--date of the Data Token creation
toktime       VARCHAR2(50),

--user-ID of the who created this Data Token
schemausrid   VARCHAR2(50),

schemamsgid   VARCHAR2(50),

auxctx        BLOB,
auxctxlen     NUMBER,

--description of this Data Token
descr         VARCHAR2(2000),

epoch         NUMBER


           - creates an unique index ("Idx1TOK"+dfbusname)
on ("TabTOK"+dfbusname+"  "+(iid, aid, tag, vid))
```

Tracking Table

```
Name        = "TabTRK"+dfbusname
           PRIMARY KEY = sequence-number

Column-Name       Column-Type
--------------------------------------------
 trkseq        NUMBER  PRIMARY KEY,

 --token seq-number
 tokseq        NUMBER,

 --instance-ID = Oracle_SID
 instid        VARCHAR2(100),

 --Interaction-ID; identifies the caller interaction
 iid           VARCHAR2(50),

 --Activation-ID; identifies the caller process activation
 aid           VARCHAR2(150),

 --JEMHandle; identifies the caller Active EJB
 handle        VARCHAR2(2000),
```

```
 --reaction ID; identifies the caller Reaction
 rid           VARCHAR2(100),
 recursid      NUMBER,

 --Data Token push operation; see above
 op            VARCHAR2(50),

 --Data Token push reason; see above
 reason        VARCHAR2(50),

 --Interaction-ID; identifies the callee interaction
 toiid         VARCHAR2(50),

 --Activation-ID; identifies the callee process activation
 toaid         VARCHAR2(150),

 --JEMHandle; identifies the callee Active EJB
 tohandle      VARCHAR2(100),

 --process-name; identifies the callee Active EJB method name of the base
--Reaction
 toprspeid     VARCHAR2(100),

 --reaction name; identifies the callee Active EJB method name of the fired
--Reaction
 --prspeid is the same as rxnpeid for the base Reaction of a Process
 torxnpeid     VARCHAR2(100),

 --description of this tracking info
 description   VARCHAR2(2000),

 --date of the Data Token push
 trkdate       VARCHAR2(30),

 --user-ID of the who pushed the Data Token
 schemausrid   VARCHAR2(50),

 schemamsgid   VARCHAR2(50),

 --  Direction of the push:
 --    SND: at the sending side (caller with a call operation, callee with a
--return operation)
 --    RCV: at the receiving side (callee with a call operation, caller with a
--return operation)
 direction     VARCHAR2(50),
```

```
epoch          NUMBER
```

Table 10–2 Advanced Queueing Topics

AQ Topic Name	Description
JEMUSER.QT + dfbusname	queue table
JEMUSER.MQ + dfbusname	match queue
JEMUSER.SQ + dfbusname	scheduling queue
JEMUSER.FQ + dfbusname	foreign queue
JEMUSER.AQ$_QT + dfbusname + E	exception queue

In Table 10–2, the queue name consists of the characters in the AQ Topic Name column, plus an appended string, represented by dfbusname. This appended name is provided by the user that creates the data bus; the default is the empty string. For example, if the user provides the string _EDB, then the scheduling queue would be named JEMUSER.SQ_EDB; if the user makes no specification when creating the data bus, then the scheduling queue would simply be named JEMUSER.SQ. In the case of the exception queue, the character E is appended at the end.

The remainder of this section describes the AQ schema objects created.

Create AQ Queue table, topics, and default subscriber to all topics using AQ PL/SQL Packages. For every J2EE application deployed to OC4J, the JEM runtime system will add a subscriber to all topics with the J2EE application name deployed used as the subscriber name.

In the following descriptions, dfbusname is the string provided by the user that creates the data bus; the default is the empty string.

Queue Table

```
DBMS_AQADM.CREATE_QUEUE_TABLE(
              Queue_table             => quetablename,
              Multiple_consumers      => TRUE,
              Queue_payload_type      => 'SYS.AQ$_JMS_BYTES_MESSAGE',
              compatible              => '8.1.5');
```

Match Queue

```
DBMS_AQADM.CREATE_QUEUE(
              Queue_name          => matchquename,
```

```
              Queue_table          => quetablename,
              max_retries          => max_retries,
              retry_delay          => retry_delay,
              retention_time       => retention_time);

Create and add the subscriber to MATCH-Queue
subscriber :=
              sys.aq$_agent('JEMSUB' || dfbusname, null, null);

DBMS_AQADM.ADD_SUBSCRIBER(
              queue_name            => matchquename,
              subscriber            => subscriber);
```

Scheduling Queue

```
DBMS_AQADM.CREATE_QUEUE(
              Queue_name           => schedquename,
              Queue_table          => quetablename,
              max_retries          => max_retries,
              retry_delay          => retry_delay,
              retention_time       => retention_time);

Create and add the subscriber to SCHEDULING-Queue
              subscriber :=
              sys.aq$_agent('JEMSUB' || dfbusname, null, null);

DBMS_AQADM.ADD_SUBSCRIBER(
              queue_name            => schedquename,
              subscriber            => subscriber);
```

Foreign Queue

```
DBMS_AQADM.CREATE_QUEUE(
              Queue_name           => foreignquename,
              Queue_table          => quetablename,
              max_retries          => max_retries,
              retry_delay          => retry_delay,
              retention_time       => retention_time);

Create and add the subscriber to FOREIGN-Queue
              subscriber :=
              sys.aq$_agent('JEMSUB' || dfbusname, null, null);

DBMS_AQADM.ADD_SUBSCRIBER(
              queue_name            => foreignquename,
```

```
            subscriber          => subscriber);
```

Table 10–3 Database Views

View Name	Description
DFB + `dfbusname`	data bus view
PRS + `dfbusname`	process view
RXN + `dfbusname`	reaction view
RTL + `dfbusname`	reaction template view
TOK + `dfbusname`	token view (active-data)
TRK + `dfbusname`	tracking view

In Table 10–3, the view name consists of the characters in the View Name column, plus an appended string, represented by `dfbusname`. This appended name is provided by the user that creates the data bus; the default is the empty string. For example, if the user provides the string `_EDB`, then the data bus view would be named `DFB_EDB`; if the user makes no specification when creating the data bus, then the data bus table would simply be named `DFB`.

Data Bus View

```
Name        =  ("DFB"+dfbusname) [ from ("TabDFB"+dfbusname) ]

               Column-Name
               ------------
                instid,                 (Instance-ID or Oracle-SID)
                version,
                maxrxnretry,
                state,
                creatdate,
                tokttldiff,
                tokttlcalldiff,
                tokttldatdiff,
                tokclnttldiff,
                rxnttldiff,
                gccycle
```

Process View

```
Name            =  "PRS"+dfbusname
```

```
Column-Name
------------
 iid           ,   (Interaction-ID)
 aid           ,   (Activation-ID)
 retiid        ,   (return IID)
 retaid        ,   (return AID)
 rethandlelen  ,
 prspeid       ,   (process name=method name)
 cmpltag       ,
 state         ,
 flags         ,
 prsstart      ,
 prsend
```

Reaction View

```
Name="RXN"+dfbusname [ from ("TabRXN"+dfbusname) ]
```

```
Column-Name
------------
 iid           ,   (Interaction-ID)
 aid           ,   (Activation-ID)
 rid           ,   (Reaction-ID)
 recursid      ,
 prspeid       ,   (process name)
 rxnpeid       ,   (reaction name)
 grpid         ,
 totmattups    ,
 retrycnt      ,
 state         ,
 status        ,
 type          ,
 flags         ,
 rxnpri        ,
 rxnttl        ,
 descr         ,
 rxndate       ,
 rxnstart      ,
 rxnend
```

Reaction Template View

```
Name      =  "RTL"+dfbusname [ from ("TabRTL"+dfbusname) ]
```

```
Column-Name
```

```
------------
 mattupseq      ,
 matindx        ,
 tokiid         ,
 tokaid         ,
 tag            ,
 vid            ,
 vidtype        ,
 iid            ,   (Interaction-ID)
 aid            ,   (Activation-ID)
 rid            ,   (Reaction-ID)
 recursid       ,
 polycnd        ,
 objclassname   ,   (java-class name of the obj for reaction)
 state          ,
 type           ,
 flags          ,
 rxnpri         ,   (priority of the reaction)
 timeout        ,   (timeout for the reaction)
 tokseq         ,
 scpiid         ,
 scpaid
```

Token (Active Data) View

```
Name              =  "TOK"+dfbusname

               Column-Name
               -------------
                tokseq         ,
                prevtokseq     ,
                nexttokseq     ,
                aliastokseq    ,
                readers        ,
                consumer       ,
                status         ,
                iid            ,   (Interaction-ID)
                aid            ,   (Activation-ID)
                tag            ,
                vid            ,
                prevvid        ,
                nextvid        ,
                polycnd        ,
                textinst       ,
                textinstlen    ,
```

```
state        ,
op           ,
reason       ,
type         ,
flags        ,
tokttl       ,
tokclnttl    ,
toktime
```

Tracking View

```
Name =  "TRK"+dfbusname [ from "TabTRK"+dfbusname and from "TabTOK'"+dfbusname ]
                         Condition   =  trk.tokseq = tok.tokseq  AND
                                        trk.schemausrid = (select UPPER(user)
from dual)))

                Column-Name
                ------------
                 tok.tokseq        ,   (seq-number if the token)
                 tok.prevtokseq    ,   (prev seq-number for the token)
                 tok.nexttokseq    ,   (next seq-number for the token)
                 tok.aliastokseq   ,
                 tok.readers       ,
                 tok.consumer      ,
                 tok.status        ,
                 tok.iid           ,   (Interaction-ID of the token)
                 tok.aid           ,   (Activation-ID of the token)
                 tok.tag           ,
                 tok.vid           ,
                 tok.prevvid       ,
                 tok.nextvid       ,
                 tok.polycnd       ,
                 tok.state         ,
                 tok.type          ,
                 tok.flags         ,
                 tok.tokttl        ,   (time-to-live for the token)
                 tok.tokclnttl     ,
                 tok.toktime       ,
                 trk.instid        ,   (instance-ID or Oracle-SID)
                 trk.iid              ,   (Interaction-ID for tracking)
                 trk.aid              ,   (Activation-ID for tracking)
                 trk.rid           ,   (Reaction-ID for tracking)
                 trk.recursid      ,
                 trk.op            ,
                 trk.reason        ,
```

```
trk.toiid          ,
trk.toaid          ,
trk.tohandle       ,
trk.toprspeid      ,
trk.torxnpeid      ,
trk.description    ,
trk.direction      ,
trk.trkdate
```

A

EJB 1.1 CMP Entity Beans

If you have EJB 1.1 CMP entity beans from last release, this appendix informs you of how OC4J maps your EJB 1.1 CMP deployment descriptor elements to OC4J-specific mappings. Oracle encourages you to migrate to using the EJB 2.0 method for CMP entity beans; however, Oracle supports both specifications.

This chapter demonstrates simple CMP EJB 1.1 development with a basic configuration and deployment. Download the CMP entity bean example (`cmpapp.jar`) from the OC4J sample code page at `http://otn.oracle.com/sample_code/tech/java/oc4j/htdocs/oc4jsamplecode/oc4j-demo-ejb.html` on the OTN site.

This chapter demonstrates the following:

- Creating Entity Beans—Demonstrates how to create a simple container-managed persistent entity bean.
- Advanced CMP Entity Beans—Demonstrates advanced configuration for finder methods, object-relational mapping, and so on.

See Chapter 3, "CMP Entity Beans" for details on CMP EJB 2.0 entity beans.

Creating Entity Beans

To create an entity bean, perform the following steps:

1. Create a remote interface for the bean. The remote interface declares the methods that a client can invoke. It must extend `javax.ejb.EJBObject`.
2. Create a home interface for the bean. The home interface must extend `javax.ejb.EJBHome`. It defines the `create` and finder methods, including `findByPrimaryKey`, for your bean.
3. Define the primary key for the bean. The primary key identifies each entity bean instance. The primary key must be either a well-known class, such as `java.lang.String`, or defined within its own class.
4. Implement the bean. This includes the following:
   a. The implementation for the methods that are declared in your remote interface.
   b. The methods that are defined in the `javax.ejb.EntityBean` interface.
   c. The methods that match the methods that are declared in your home interface. This includes the following:
      * the `ejbCreate` and `ejbPostCreate` methods with parameters matching the associated `create` method defined in the home interface
      * an `ejbFindByPrimary` key method, which corresponds to the `findByPrimaryKey` method of the home interface
      * any other finder methods that were defined in the home interface
5. Create the bean deployment descriptor. The deployment descriptor specifies properties for the bean through XML elements. This step is where you identify the data within the bean that is to be managed by the container.
6. If the persistent data is saved to or restored from a database and you are not using the defaults provided by the container, then you must ensure that the correct tables exist for the bean. In the extreme default scenario, the container will actually create the table and columns for your data based on deployment descriptor and datasource information.
7. Create an EJB JAR file containing the bean, the remote and home interfaces, and the deployment descriptor. Once created, configure the `application.xml` file, create an EAR file, and install the EJB in OC4J.

The following sections demonstrate a simple CMP entity bean. This example continues the use of the employee example, as in other chapters—without adding complexity.

Home Interface

The home interface must contain a `create` method, which the client invokes to create the bean instance. Each `create` method can have a different signature. For an entity bean, you must develop a `findByPrimaryKey` method. Optionally, you can develop other finder methods for the bean, which are named `find<name>`.

Example A–1 Entity Bean Employee Home Interface

To demonstrate an entity bean, this example creates a bean that manages a purchase order. The entity bean contains a list of items that were ordered by the customer.

The home interface extends `javax.ejb.EJBHome` and defines the `create` and `findByPrimaryKey` methods.

```
package employee;

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

public interface EmployeeHome extends EJBHome
{

  public Employee create(Integer empNo)
    throws CreateException, RemoteException;

  // Find an existing employee
  public Employee findByPrimaryKey (Integer empNo)
    throws FinderException, RemoteException;

  //Find all employees
  public Collection findAll()
    throws FinderException, RemoteException;
}
```

Remote Interface

The entity bean remote interface is the interface that the customer sees and invokes methods upon. It extends `javax.ejb.EJBObject` and defines the business logic methods. For our employee entity bean, the remote interface contains methods for adding and removing employees, and retrieving and setting employee information.

```
package employee;

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

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 setEmpName(String newEmpName) throws RemoteException;
  public void setSalary(Float newSalary) throws RemoteException;
}
```

Entity Bean Class

The entity bean class must implement the following methods:

- the target methods for the methods that are declared in the home interface, which includes the `ejbCreate` method and any finder methods, including `ejbFindByPrimaryKey`
- the business logic methods that are declared in the remote interface
- the methods that are inherited from the `EntityBean` interface

However, with container-managed persistence, the container manages most of the target methods and the data objects. This leaves little for you to implement.

```
package employee;

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

public class EmployeeBean extends Object implements EntityBean
{
```

```
public Integer empNo;
public String empName;
public Float salary;
public EntityContext entityContext;

public EmployeeBean()
{
  // Constructor. Do not initialize anything in this method.
  // All initialization should be performed in the ejbCreate method.
}

public Integer getEmpNo()
{
  return empNo;
}

public String getEmpName()
{
  return empName;
}

public Float getSalary()
{
  return salary;
}

public void setEmpName(String empName)
{
  this.empName = empName;
}

public void setSalary(Float salary) {
  this.salary = salary;
}

public Integer ejbCreate(Integer empNo)
  throws CreateException, RemoteException
{
  this.empNo = empNo;
  return empNo;
}

public void ejbPostCreate(Integer empNo)
  throws CreateException, RemoteException
```

```
    {
      // Called just after bean created; container takes care of implementation
    }

    public void ejbStore()
    {
      // Called when bean persisted; container takes care of implementation
    }

    public void ejbLoad()
    {
      // Called when bean loaded; container takes care of implementation
    }

    public void ejbRemove()
    {
      // Called when bean removed; container takes care of implementation
    }

    public void ejbActivate()
    {
      // Called when bean activated; container takes care of implementation.
      // If you need resources, retrieve them here.
    }

    public void ejbPassivate()
    {
      // Called when bean deactivated; container takes care of implementation.
      // if you set resources in ejbActivate, remove them here.
    }

    public void setEntityContext(EntityContext entityContext)
    {
      this.entityContext = entityContext;
    }

    public void unsetEntityContext()
    {
      this.entityContext = null;
    }
  }
```

Persistent Data

In CMP entity beans, you define the persistent data both in the bean instance and in the deployment descriptor. The declaration of the data fields in the bean instance creates the resources for the fields. The deployment descriptor defines these fields as persistent.

In our employee example, the data fields are defined in the bean instance, as follows:

```
public Integer empNo;
public String empName;
public Float salary;
```

These fields are defined as persistent fields in the `ejb-jar.xml` deployment descriptor within the `<cmp-field><field-name>` element, as follows:

```
<enterprise-beans>
      <entity>
         <display-name>Employee</display-name>
         <ejb-name>EmployeeBean</ejb-name>
         <home>employee.EmployeeHome</home>
         <remote>employee.Employee</remote>
         <ejb-class>employee.EmployeeBean</ejb-class>
         <persistence-type>Container</persistence-type>
         <prim-key-class>java.lang.Integer</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>
         <primkey-field>empNo</primkey-field>
      </entity>
...
</enterprise-beans>
```

In most cases, you map the persistent data fields to columns in a table that exists in a designated database. However, you can accept the defaults for these fields—thus, avoiding more deployment descriptor configuration.

OC4J contains some defaults for mapping these fields to a database and its table.

- Database—The default database as set up in your OC4J instance configuration. For the JNDI name, use the `<location>` element for emulated data sources and `<ejb-location>` element for non-emulated data sources.

Upon installation, the default database is a locally installed Oracle database that must be listening on port 5521 with a SID of ORACLE.

Note: Unfortunately, you must change the "default" database configuration in the data-sources.xml file to coordinate with the default installation for an Oracle database. The default port and SID for an Oracle database are 1521 and ORCL, respectively.

To customize the default database, change the first configured database (including its <ejb-location>) to point to your database.

- Table with correct column names—The container creates a default table with the same name as the bean name (defined in <ejb-name>), with columns having the same name as the <cmp-field> elements in the designated database. The data types for the database, translating Java data types to database data types, are defined in the specific database XML file, such as oracle.xml.

If you want to designate another database or generate a table that has a different naming convention, see "EJB 1.1 Object-Relational Mapping of Persistent Fields" on page A-13 for a description of how to customize your database, table, and column names.

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 either primitive, serializable, or types that can be mapped to SQL types. You can define your primary key in one of two ways:

- Define the type of the primary key to be a well-known type. The type is defined in the <prim-key-class> in the deployment descriptor. The data field that is identified as the persistent primary key is identified in the <primkey-field> element in the deployment descriptor. 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. This class is declared in the <prim-key-class> element in the deployment descriptor. This is an advanced method for defining a primary key, so it is discussed in "Defining the Primary Key in a Class" on page A-9.

For a simple CMP, you can define your primary key to be a well-known type by defining the data type of the primary key within the deployment descriptor.

The employee example defines its primary key as a `java.lang.Integer` and uses the employee number (`empNo`) as its primary key.

```
<enterprise-beans>
      <entity>
         <display-name>Employee</display-name>
         <ejb-name>EmployeeBean</ejb-name>
         <home>employee.EmployeeHome</home>
         <remote>employee.Employee</remote>
         <ejb-class>employee.EmployeeBean</ejb-class>
         <persistence-type>Container</persistence-type>
         <prim-key-class>java.lang.Integer</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>
         <primkey-field>empNo</primkey-field>
      </entity>
...
</enterprise-beans>
```

Defining the Primary Key in a Class

If your primary key is more complex than a simple data type, your primary key must be a class that is serializable of the name `<`*`name`*`>PK`. You define the primary key class within the `<prim-key-class>` element in the deployment descriptor.

The primary key variables must adhere to the following:

- Be defined within a `<cmp-field><field-name>` element in the deployment descriptor. This enables the container to manage the primary key fields.
- Be declared within the bean class as `public` and restricted to be either primitive, serializable, or types that can be mapped to SQL types.

Within the primary key class, you implement a constructor for creating a primary key instance. Once defined in this manner, the container manages the primary key, as well as storing the persistent data.

The following example is a complex primary key made up of employee number and country code. Our company is so large that it reuses employee numbers in different countries. Thus, the combination of both the employee number and the country code uniquely identifies each employee.

```
package employee;

public class EmpPK implements java.io.Serializable
{
  public Integer empNo;
  public String countryCode;

  //constructor
  public EmpPK ( ) { }
}
```

The primary key class is declared within the `<prim-key-class>` element and its variables, each within a `<cmp-field><field-name>` element in the XML deployment descriptor, as follows:

```
<enterprise-beans>
      <entity>
         <display-name>Employee</display-name>
         <ejb-name>EmployeeBean</ejb-name>
         <home>employee.EmployeeHome</home>
         <remote>employee.Employee</remote>
         <ejb-class>employee.EmployeeBean</ejb-class>
         <persistence-type>Container</persistence-type>
         <prim-key-class>employee.EmpPK</prim-key-class>
         <reentrant>False</reentrant>
         <cmp-field><field-name>empNo</field-name></cmp-field>
         <cmp-field><field-name>countryCode</field-name></cmp-field>
      </entity>
      ...
</enterprise-beans>
```

Deploying the Entity Bean

Archive your EJB into a JAR file. You deploy the entity bean in the same way as the session bean, which "Prepare the EJB Application for Assembly" on page 2-13 and "Deploy the Enterprise Application to OC4J" on page 2-15 explain in detail.

Advanced CMP Entity Beans

This section discusses how to implement your bean beyond the simple CMP entity bean. It includes the following sections:

- EJB 1.1 Advanced Finder Methods
- EJB 1.1 Object-Relational Mapping of Persistent Fields

EJB 1.1 Advanced Finder Methods

Specifying the `findByPrimaryKey` method is easy to do in OC4J. All the fields for defining a simple or complex primary key are specified within the `ejb-jar.xml` deployment descriptor. However, if you want to define other finder methods in a CMP entity bean, you must do the following:

1. Add the finder method to the home interface.
2. Add the EJB 1.1 finder method definition to the OC4J-specific deployment descriptor—the `orion-ejb-jar.xml` file.

Add the Finder Method to Home Interface

You must first add the finder method to the home interface. For example, with the employee entity bean, if we wanted to retrieve all employees, the `findAll` method would be defined within the home interface, as follows:

```
public Collection findAll() throws FinderException, RemoteException;
```

Add the EJB 1.1 Finder Method Definition to the OC4J-Specific Deployment Descriptor

After specifying the finder method in the home interface, modify the `orion-ejb-jar.xml` file with the EJB 1.1 finder method specifics. The container identifies the correct query necessary for retrieving the required fields.

The EJB 1.1 `<finder-method>` element defines all finder methods—excluding the `findByPrimaryKey` method. The simplest finder method to define is the `findByAll` method. The `query` attribute in the `<finder-method>` element specifies the `WHERE` clause for the query. If you want all rows retrieved, then an empty query (`query=""`) returns all records.

The following example retrieves all records from the `EmployeeBean`. The method name is `findAll`, and it requires no parameters because it returns a `Collection` of all employees.

```
<finder-method query="">
  <method>
     <ejb-name>EmployeeBean</ejb-name>
     <method-name>findAll</method-name>
         <method-params></method-params>
     </method>
</finder-method>
```

After deploying the application with this bean, OC4J adds the following statement of what query it invokes as a comment in the finder method definition:

```
<finder-method query="">
<!-- Generated SQL: "select EmployeeBean.empNo, EmployeeBean.empName,
    EmployeeBean.salary from EmployeeBean" -->
  <method>
     <ejb-name>EmployeeBean</ejb-name>
     <method-name>findAll</method-name>
         <method-params></method-params>
     </method>
</finder-method>
```

Verify that it is the type of query that you expect.

To be more specific, modify the `query` attribute with the appropriate `WHERE` clause. This clause refers to passed in parameters using the '$' symbol: the first parameter is denoted by $1, the second by $2. All `<cmp-field>` elements that are used within the `WHERE` clause are denoted by `$<cmp-field>` name.

The following example specifies a `findByName` method (which should be defined in the home interface) where the name of the employee is given as in the method parameter, which is substituted for the $1. It is matched to the CMP name, "`empName`". Thus, our `query` attribute is modified to contain the following for the `WHERE` clause: "`$empname=$1`".

```
<finder-method query="$empname = $1">
  <method>
    <ejb-name>EmployeeBean</ejb-name>
    <method-name>findByName</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
    </method-params>
  </method>
</finder-method>
```

If you have more than one method parameter, each parameter type is defined in successive `<method-param>` elements and referred to in the query statement by successive $*n*, where *n* represents the number.

Note: You can also specify a SQL JOIN in the `query` attribute.

If you wanted to specify a full query and not just the section after the WHERE clause, specify the `partial` attribute to FALSE and then define the full query in the `query` attribute. The default value for `partial` is true, which is why it is not specified on the previous finder-method example.

```
<finder-method partial="false"
               query="select * from EMP where $empName = $1">
        <!-- Generated SQL: "select * from EMP where EMP.ENAME = ?" -->
        <method>
                <ejb-name>EmployeeBean</ejb-name>
                <method-name>findByName</method-name>
                <method-params>
                        <method-param>java.lang.String</method-param>
                </method-params>
        </method>
</finder-method>
```

Specifying the full SQL query is useful for complex SQL statements.

EJB 1.1 Object-Relational Mapping of Persistent Fields

As "Persistent Data" on page A-7 discusses, your persistent data can be automatically mapped to a database table by the container. However, if the data represented by your bean is more complex or you do not want to accept the defaults that OC4J provides for you, then map the CMP designated fields to an existing database table and its applicable rows within the `orion-ejb-jar.xml` file. Once mapped, the container provides the persistence storage of the CMP data to the indicated table and rows.

Before configuring the object-relational mapping, add the `DataSource` used for the destination within the `<resource-ref>` element in the `ejb-jar.xml` file.

Mapping EJB 1.1 CMP Fields to a Database Table and Its Columns

Configure the following within the `orion-ejb-jar.xml` file:

1. Configure the `<entity-deployment>` element for every entity bean that contains CMP fields that will be mapped within it.
2. Configure a `<cmp-field-mapping>` element for every field within the bean that is mapped. Each `<cmp-field-mapping>` element must contain the name of the field to be persisted.
   a. Configure the primary key in the `<primkey-mapping>` element contained within its own `<cmp-field-mapping>` element.
   b. Configure simple data types (such as a primitive, simple object, or serializable object) that are mapped to a single field within a single `<cmp-field-mapping>` element. The name and database field are fully defined within the element attributes.
   c. Configure complex data types using one of the many sub-elements of the `<cmp-field-mapping>` element. These can be one of the following:
      * If you define an object as your complex data type, then specify each field or property within the object in the `<fields>` or `<properties>` element.
      * If you specify a field defined in another entity bean, then define the home interface of this entity bean in the `<entity-ref>` element.
      * If you define a `List`, `Collection`, `Set`, or `Map` of fields, then define these fields within the `<list-mapping>`, `<collection-mapping>`, `<set-mapping>`, `<map-mapping>` elements.

Examples for simple and complex O-R mappings are shown in the following sections:

- EJB 1.1 One-to-One Mapping Example
- EJB 1.1 One-to-Many Mapping Example

EJB 1.1 One-to-One Mapping Example The following example demonstrates how to map EJB 1.1 persistent data fields in your bean instance to database tables and columns by mapping the employee persistence data fields to the Oracle database table `EMP`.

- The bean is identified in the `<entity-deployment>` `name` attribute. The JNDI name for this bean is defined in the `location` attribute.

- The database table name is defined in the `table` attribute. And the database is specified in the `data-source` attribute, which should be identical to the `<ejb-location>` name of a `DataSource` defined in the `data-sources.xml` file.
- The bean primary key, `empNo`, is mapped to the database table column, `EMPNO`, within the `<primkey-mapping>` element.
- The bean persistent data fields, `empName` and `salary`, are mapped to the database table columns `ENAME` and `SAL` within the `<cmp-field-mapping>` element.

```
<entity-deployment name="EmpBean" location="emp/EmpBean"
      wrapper="EmpHome_EntityHomeWrapper2" max-tx-retries="3"
      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="ename" />
      <cmp-field-mapping name="salary" persistence-name="sal" />
  ...
</entity-deployment>
```

After deployment, OC4J maps this to the following:

Bean	Database
`emp/EmpBean`	EMP table, located at `jdbc/OracleDS` in the `data-sources.xml` file
`empNo`	`EMPNO` column as primary key
`empName`	`ENAME` column
`salary`	`SAL` column

EJB 1.1 One-to-Many Mapping Example If you have two beans that access two tables for their data, you must map the persistent data from both beans to the respective tables.

We added a department number to our employee example. Each employee belongs to a department; each department has multiple employees. The container will handle this object-relational mapping; however, you must specify how the data is stored.

The employee data maps to the employee database table; the department data is mapped to the database department table. The employee database table also contains a foreign key of the department number to link this information together.

The XML configuration for the employee bean, `EmpBean`, is as follows:

- Same XML configuration details for the employee bean as stated above, with the addition of the definition of the department number as part of the employee entity bean.
- The department number is defined in the bean instance as `deptno`, which relates to the department number defined in the `DeptBean`. Thus, the `DeptBean`, its `deptno` field, and its mapping to the database column, `deptno`, is configured within a `<cmp-field-mapping><entity-ref>` element, as follows:

```
<entity-deployment name="EmpBean" location="emp/EmpBean"
wrapper="EmpHome_EntityHomeWrapper2" max-tx-retries="3" 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="ename" />
   <cmp-field-mapping name="salary" persistence-name="sal" />
   <cmp-field-mapping name="dept">
       <entity-ref home="dept/DeptBean">
               <cmp-field-mapping name="dept" persistence-name="deptno" />
        </entity-ref>
   </cmp-field-mapping>
   <finder-method query="">
    <!-- Generated SQL: "select EMP.empno, EMP.ename, EMP.sal,
        EMP.deptno from EMP" -->
      <method>
       <ejb-name>EmpBean</ejb-name>
        <method-name>findAll</method-name>
        <method-params></method-params>
      </method>
   </finder-method>
</entity-deployment>
```

> **Note:** This definition within the `EmpBean` configuration refers to the definition of the `deptno` within the `DeptBean` configuration.

The XML configuration for the department bean, `DeptBean`, is as follows:

- The bean is identified in the `<entity-deployment>` `name` attribute. The JNDI name for this bean is defined in the `location` attribute.
- The database table name is defined in the `table` attribute. And the database is specified in the `data-source` attribute, which should be identical to the `<ejb-location>` name of a `DataSource` defined in the `data-sources.xml` file.
- The bean primary key, `deptNo`, is mapped to the `dept` database table in its `DEPTNO` column within the `<primkey-mapping>` element.
- The bean persistent data field, `deptName`, is mapped to the `DEPT` database table in its `DNAME` column within a `<cmp-field-mapping>` element.
- The bean persistent data field, `employees`, is actually a bean—the employee bean. Thus, the example uses the `<collection-mapping>` element to specify all fields within the employee bean. A `Collection` containing the employee information is returned. See the bold text in the example below.
  - The employees field maps to the `EmpBean` entity bean. Its home interface reference is defined in the `home` attribute of the `<entity-ref>` element.
  - The primary key used to retrieve the employees is defined as `deptNo` within the `<primkey-mapping>` element and is mapped to the database column `DEPTNO`.
  - All fields that are of interest to the department bean are defined within `<cmp-field-mapping>` elements. The bean instance fields within the `EmpBean` that are of interest are `empNo`, `empName`, and `salary`. Their respective database columns are also specified: `EMPNO`, `ENAME`, and `SAL.` The database table itself is defined in the `EmpBean` `<entity-deployment>` definition.

```
<entity-deployment name="DeptBean" location="dept/DeptBean"
wrapper="DeptHome_EntityHomeWrapper2" max-tx-retries="3" table="dept"
data-source="jdbc/OracleDS">
   <primkey-mapping>
        <cmp-field-mapping name="deptNo" persistence-name="deptno" />
   </primkey-mapping>
   <cmp-field-mapping name="deptName" persistence-name="dname" />
   <cmp-field-mapping name="employees">
   <collection-mapping table="emp">
       <primkey-mapping>
           <cmp-field-mapping name="deptNo" persistence-name="deptno" />
       </primkey-mapping>
       <value-mapping type="emp.Emp">
```

```
       <cmp-field-mapping>
          <entity-ref home="emp/EmpBean">
           <cmp-field-mapping name="empNo" persistence-name="empno"/>
           <cmp-field-mapping name="empName" persistence-name="ename"/>
           <cmp-field-mapping name="salary" persistence-name="sal"/>
          </entity-ref>
        </cmp-field-mapping>
    </value-mapping>
</collection-mapping>
</cmp-field-mapping>
...
</entity-deployment>
```

B

OC4J-Specific DTD Reference

This appendix describes the elements contained within the OC4J-specific EJB deployment descriptor: `orion-ejb-jar.dtd`. This appendix covers the structure and briefly describes the elements in this DTD; however, most of these elements are fully described in other sections of this book.

The DTD is located at `http://xmlns.oracle.com/ias/dtds/orion-ejb-jar.dtd`.

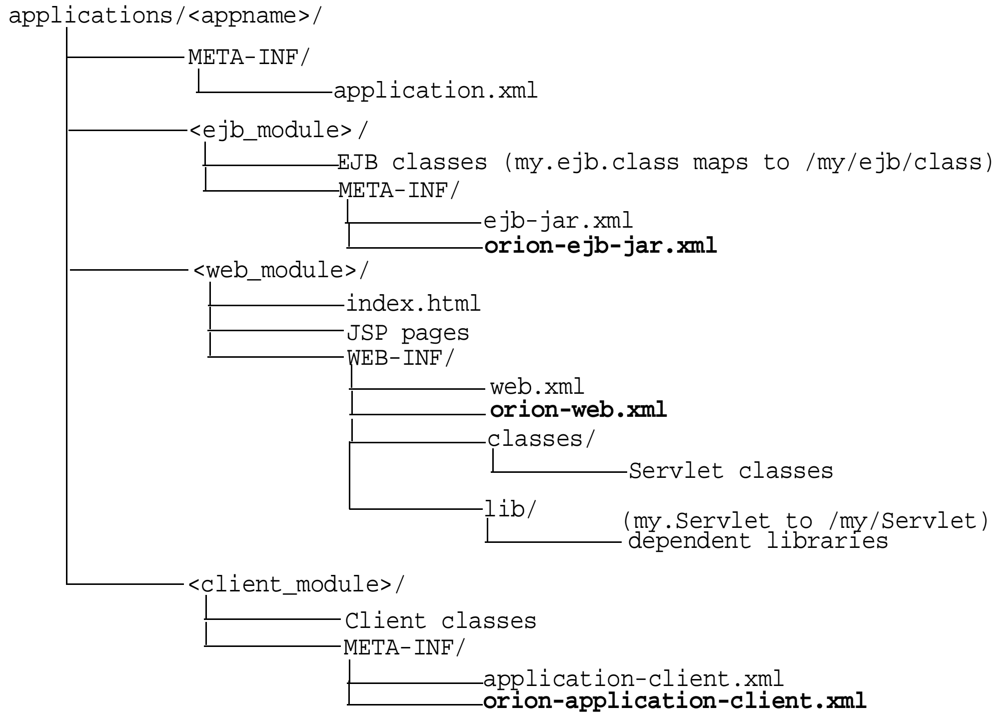
The description of this deployment descriptor has been divided into the following sections:

- Overall description of each element section—Each section of elements of this XML file is described in "OC4J-Specific Deployment Descriptor for EJBs" on page B-3.
- Element description—An alphabetical listing and description for each element is discussed in "Element Description" on page B-21.

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 simply 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 B–1.

Figure B–1 Development Application Directory Structure



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>
   <session-deployment ...></session-deployment>
   <entity-deployment ...></entity-deployment>
   <message-driven-deployment ...></message-driven-deployment>
   <jem-deployment ...></jem-deployment>
   <jem-server-extension ...></jem-server-extension>
 </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;

- Session Bean Section
- Entity Bean Section
- Message Driven Bean Section
- EJB 1.1 CMP Field Mapping Section
- Method Definition

Session Bean Section

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=...
      name=... persistence-filename=... timeout=... 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>
```

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, is described in "Create the Deployment Descriptor" on page 2-11 in Chapter 2, "An EJB Primer For OC4J".
- The <ior-security-config> element is an interoperability element, which is discussed fully in the Interoperability chapter in the *Oracle9iAS Containers for J2EE Services Guide.*

- The `<env-entry-mapping>` element maps environment variables to JNDI names and is discussed in "Environment variables" on page 8-15.
- The `<ejb-ref-mapping>` element maps any EJB references to JNDI names and is discussed in "Environment References To Other Enterprise JavaBeans" on page 8-20.
- The `<resource-ref-mapping>` element maps any EJB references to JNDI names and is discussed in "Environment References To Resource Manager Connection Factory References" on page 8-20.
- 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>` elment is used to declare the destination. Thus, the `<resource-env-ref-mapping>` element maps the destination object. See "Using Logical Names in the JMS JNDI Lookup" on page 7-15 for more information.

The attributes for the `<session-deployment>` element are as follows:

Table B–1 Attributes for <session-deployment> Element

Attribute	Description
`pool-cache-timeout`	The pool-cache-timeout applies for stateless session EJBs. This parameter specifies how long to keep stateless sessions cached in the pool. For stateless session beans, if you specify a pool-cache-timeout, then at every pool-cache-timeout interval, all beans 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`	This parameter specifies the maximum time to wait for any resource to make a business/life-cycle method invocation. This is not a timeout for how long a business method invocation can take. If the timeout is reached, a `TimedOutException` is thrown. This excludes database connections. Default Values: 90000 milliseconds. Set to 0 if you want the timeout to be forever. See the EJB section in the *Oracle9i Application Server Performance Guide* for more information.

Table B–1 Attributes for <session-deployment> Element (Cont.)

Attribute	Description
`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'.
`location`	The JNDI-name to which this bean will be bound.
`max-instances`	The number of maximum bean implementation instances to be kept instantiated or pooled. The default is 100. This setting is valid for stateless session beans only.
`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`	This parameter specifies the number of times to retry a transaction that was rolled back due to system-level failures. Generally, we recommend that you start by setting max-tx-retries to 0 and adding 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 set max-tx-retries=0. Default Value: 3. See the EJB section in the *Oracle9i Application Server Performance Guide* for more information.
`name`	The name of the bean, which matches the name of a bean in the assembly section of the EJB deployment descriptor (`ejb-jar.xml`).
`persistence-filename`	Path to the file where sessions are stored across restarts.

Table B–1 Attributes for <session-deployment> Element (Cont.)

Attribute	Description
`timeout`	The timeout applies for stateful session EJBs. If the value is zero or negative, then all timeouts are disabled. The timeout parameter is an inactivity timeout for stateful session beans. 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. Default Value: 30 (minutes)
`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.

Entity Bean Section

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=... instance-cache-timeout=... isolation=...
      location=... locking-mode=... max-instances=... min-instances=...
      max-tx-retries=... update-chnaged-fields-only=...
      disable-wrapper-cache=... name=... pool-cache-timeout=...
      table=... validity-timeout=... force-update=...
      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>
    <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=... >
     <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>
```

Each of the element groups are discussed in the following sections of the OC4J documentation set:

- Entity bean examples, which includes the `<entity-deployment>` element, are described in Chapter 3, "CMP Entity Beans", Chapter 4, "Entity Relationship Mapping", Chapter 5, "EJB Query Language", and Chapter 6, "BMP Entity Beans".
- The `<ior-security-config>` element configures CSIv2 security policies for interoperability, which is discussed fully in the Interoperability chapter in the *Oracle9iAS Containers for J2EE Services Guide.*
- The `<primkey-mapping>` element maps the primary key to the CMP field it represents. See "Explicit Mapping of Persistent Fields to the Database" on page 3-15 for more information.

- The `<cmp-field-mapping>` element maps each `<cmp-field>` element to its database row. See "Explicit Mapping of Persistent Fields to the Database" on page 3-15 for more information.
- The `<finder-method>` element is used to create finder methods for EJB 1.1 entity beans. To create EJB 2.0 finder methods, see "Entity Relationship Mapping". To continue to use EJB 1.1 finder methods with this element, see "EJB 1.1 Advanced Finder Methods" on page A-11.
- The `<env-entry-mapping>` element maps environment variables to JNDI names and is discussed in "Environment variables" on page 8-15.
- The `<ejb-ref-mapping>` element maps any EJB references to JNDI names and is discussed in "Environment References To Other Enterprise JavaBeans" on page 8-20.
- The `<resource-ref-mapping>` element maps any EJB references to JNDI names and is discussed in "Environment References To Resource Manager Connection Factory References" on page 8-20.
- 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>` elment is used to declare the destination. Thus, the `<resource-env-ref-mapping>` element maps the destination object. See "Using Logical Names in the JMS JNDI Lookup" on page 7-15 for more information.

The attributes for the `<entity-deployment>` element are as follows:

Table B–2 Attributes for <entity-deployment> Element

Attribute	Description
`call-timeout`	This parameter specifies the maximum time to wait for any resource to make a business/life-cycle method invocation. This is not a timeout for how long a business method invocation can take. If the timeout is reached, a `TimedOutException` is thrown. This excludes database connections. Default Values: 90000 milliseconds. Set to 0 if you want the timeout to be forever. See the EJB section in the *Oracle9i Application Server Performance Guide* for more information.

Table B–2 Attributes for <entity-deployment> Element (Cont.)

Attribute	Description
clustering-schema	Do not use. Not needed in this release.
copy-by-value	Whether or not to copy (clone) all the incoming and outgoing parameters in EJB calls. Set to 'false' if you are certain that your application does not assume copy-by-value semantics for a speed-up. The default is 'true'.
data-source	The name of the data source used if using container-managed persistence.
exclusive-write-access	Whether or not the EJB-server has exclusive write (update) access to the database backend. 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. 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. Default is false for beans with locking-mode=optimistic or pessimistic and true for locking-mode=read-only. 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. See "Exclusive Write Access to the Database" on page 8-12 for more information.
do-select-before-insert	Recommend setting to false to avoid the extra select before insert which checks if the entity already exists before doing the insert. This will then detect a duplicate, if there is one, during the insert. Default Value: true
instance-cache-timeout	The amount of time in seconds that entity wrapper instances are assigned to an identity. If you specify 'never', you retain the wrapper instances until they are garbage collected. The default is 60 seconds.
location	The JNDI-name to which this bean will be bound.

Table B–2 Attributes for <entity-deployment> Element (Cont.)

Attribute	Description
`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 "Entity Bean Concurrency and Database Isolation Modes" on page 8-10 and *Oracle9i Application Server Performance Guide* .
`locking-mode`	The concurrency modes configure when to block to manage resource contention or when to execute in parallel. For more information, see "Entity Bean Concurrency and Database Isolation Modes" on page 8-10 and *Oracle9i 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 100. See "Configuring Pool Sizes For Entity Beans" on page 8-13 for more information.
`min-instances`	The number of minimum bean implementation instances to be kept instantiated or pooled. The default is 0. See "Configuring Pool Sizes For Entity Beans" on page 8-13 for more information.

Table B–2 Attributes for <entity-deployment> Element (Cont.)

Attribute	Description
`max-tx-retries`	This parameter specifies the number of times to retry a transaction that was rolled back due to system-level failures. Generally, we recommend that you start by setting max-tx-retries to 0 and adding 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 set max-tx-retries=0. Default Value: 3. See the EJB section in the *Oracle9i Application Server Performance Guide* for more information.
`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. See "Techniques for Updating Persistence" on page 8-14 for more information.
`disble-wrapper-cache`	If true, a pool of wrapper instances is not maintained. The default is true. See "Configuring Pool Sizes For Entity Beans" on page 8-13 for more information.
`name`	The name of the bean, which matches the name of a bean in the assembly section of the EJB deployment descriptor (`ejb-jar.xml`).
`pool-cache-timeout`	The amount of time in seconds that the bean implementation instances are to be kept in the "pooled" (unassigned) state, specifying '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.

Table B–2 Attributes for <entity-deployment> Element (Cont.)

Attribute	Description
`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.
`force-update`	If OC4J does not believe that any of the persistence data has changed, the `force-update` attribute set to true means that OC4J will still execute the EJB lifecycle by invoking the `ejbStore` method. This manages data in transient fields and sets appropriate persistent fields during the `ejbStore` method. For example, an image might be kept in one format in memory, but stored in a different format in the database. The default 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 `ejbRemove`() and the finder methods.

Message Driven Bean Section

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=...>
  <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-driven-deployment>
```

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 Chapter 7, "Message-Driven Beans".
- The `<env-entry-mapping>` element maps environment variables to JNDI names and is discussed in "Environment variables" on page 8-15.
- The `<ejb-ref-mapping>` element maps any EJB references to JNDI names and is discussed in "Environment References To Other Enterprise JavaBeans" on page 8-20.
- The `<resource-ref-mapping>` element maps any EJB references to JNDI names and is discussed in "Environment References To Resource Manager Connection Factory References" on page 8-20.
- 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>` elment is used to declare the destination. Thus, the `<resource-env-ref-mapping>` element maps the destination object. See "Using Logical Names in the JMS JNDI Lookup" on page 7-15 for more information.

The attributes for the `<message-driven-deployment>` element are as follows:

Table B–3 Attributes for <message-driven-deployment> Element

Attribute	Description
cache-timeout	Do not use this element.
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 + "TopicConnectionFactories" or "QueueConnectionFactories" + user defined name. The xxxConnectionFactories details what type of factory is being defined. For more information, see "OC4J-Specific Deployment Descriptor" on page 7-12.
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. For more information, see "OC4J-Specific Deployment Descriptor" on page 7-12.
max-instances	Do not use this element. Use listener-threads instead
min-instances	Do not use this element.
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).
subscription-name	If this is a topic, the subscription name is defined in the subscription-name attribute. For more information, see "OC4J-Specific Deployment Descriptor" on page 7-12.
listener-threads	The listener threads are used to concurrently consume JMS messages. The default is one thread. For more information, see "OC4J-Specific Deployment Descriptor" on page 7-12.
transaction-timeout	This attribute controls the transaction timeout interval for any container-managed transactional MDB. The default is one day. If the transaction has not completed in this timeframe, the transaction is rolled back. For more information, see "OC4J-Specific Deployment Descriptor" on page 7-12.

AC4J Active EJB Section

The `<jem-server-extension>` section defines the JNDI name of the database where the AC4J Databus is installed. The `<jem-server-extension>` contains the following structure:

```
<jem-server-extension data-source-location=... scheduling-threads=...>
  <description></description>
  <data-bus data-bus-name=... url=.../>
</jem-server-extension>
```

For more information on this element, see Chapter 10, "Active Components for Java".

The `<jem-deployment>` section provides additional deployment information for an active EJB deployed within this JAR file. The `<jem-deployment>` section contains the following structure:

```
<jem-deployment jem-name=... ejb-name=...>
  <description></description>
  <data-bus data-bus-name=... url=.../>
  <called-by>
    <caller caller-identity=.../>
  </called-by>
  <security-identity>
   <description></description>
   <use-caller-identity></use-caller-identity>
  </security-identity>
</jem-deployment>
```

The `called-by` element lets the application deployer to control or restrict the usage of the asynchronous methods defined on the AC4J bean. In the following example "CLIUSER", "SVRUSER" and "XTRAUSER" can invoke all methods defined on `AC4JBeanA`, which corresponds to the EJB with name="`ABean`". If "USER1" or "USER2" invoke this `AC4JBeanA`, then the container throws `SecurityException`.

```
<jem-deployment jem-name="AC4JBeanA" ejb-name="ABean">
  <called-by>
    <caller  caller-identity="CLIUSER"/>
    <caller  caller-identity="SVRUSER"/>
    <caller  caller-identity="XTRAUSER"/>
  </called-by>
</jem-deployment>
```

If the application deployer defines a security-role for the ABean EJB with role="USER1", then "USER1" can invoke all the methods on the ABean EJB

synchronously. However, "USER1" can not invoke the same asynchronous methods in `AC4JBeanA` unless the `called-by` element is defined for "USER1".

For more information on this element, see Chapter 10, "Active Components for Java".

EJB 1.1 CMP Field Mapping Section

If you still use EJB 1.1 CMP entity beans, you use the following elements to map the CMP fields to the database. See "Mapping EJB 1.1 CMP Fields to a Database Table and Its Columns" on page A-14 for a discussion on mapping EJB 1.1 CMP data fields.

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>
   <list-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>
   </list-mapping>
   <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>
   <map-mapping table=...>
      <primkey-mapping>
         <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
             persistence-type=...></cmp-field-mapping>
      </primkey-mapping>
      <map-key-mapping type=...>
         <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
             persistence-type=...></cmp-field-mapping>
      </map-key-mapping>
      <value-mapping immutable="true|false" type=...>
         <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
             persistence-type=...></cmp-field-mapping>
      </value-mapping>
   </map-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 "OC4J-Specific Deployment Descriptor" on page 7-12 and "Specifying Logical Roles in the EJB Deployment Descriptor" on page 8-29 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 8-34.
- The `<default-method-access>` element is described in "Specifying a Default Role Mapping for Undefined Methods" on page 8-36.

Element Description

<**assembly-descriptor**>

The mapping of the assembly descriptor elements.

<**called-by**>

Enables the application deployer to control or restrict the usage of the asynchronous methods defined on the AC4J bean. You specify the user identity that is allowed to execute all methods of the bean in this element. The identities that can be execute the AC4J beans are identified in one or more `<caller>` elements.

<**caller**>

Each caller identity allowed to execute methods on the AC4J bean are defined in a single `<caller>` element.

Attributes:

- caller-identity - The security role that is allowed to execute the AC4J bean methods.

<**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 isnt specified and not relevant). The field containing the mapping must be of type java.util.Collection.

Attiributes:

- 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 AC4J 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.
- disable-wrapper-cache - If true, a pool of wrapper instances is not maintained. The default is true. See "Configuring Pool Sizes For Entity Beans" on page 8-13 for more information.
- do-select-before insert - Recommend setting to false to avoid the extra select before insert which checks if the entity already exists before doing the insert. This will then detect a duplicate, if there is one, during the insert. Default Value: true.
- exclusive-write-access - Whether or not the EJB-server has exclusive write (update) access to the database backend. 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 "Exclusive Write Access to the Database" on page 8-12 for more information.
- instance-cache-timeout - The amount of time in seconds that entity wrapper instances are assigned to an identity. If you specify 'never', you retain the wrapper instances until they are garbage collected. The default is 60 seconds.

- 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 "Entity Bean Concurrency and Database Isolation Modes" on page 8-10 and *Oracle9i 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 "Entity Bean Concurrency and Database Isolation Modes" on page 8-10 and *Oracle9i 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 100. See "Configuring Pool Sizes For Entity Beans" on page 8-13 for more information.
- min-instances - The number of minimum bean implementation instances to be kept instantiated or pooled. The default is 0. See "Configuring Pool Sizes For Entity Beans" on page 8-13 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 3. Consider 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 set this value to 0 and add retries only where errors are seen that could be resolved through a retry.
- 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. See "Techniques for Updating Persistence" on page 8-14 for more information.
- wrapper - Name of the OC4J remote home wrapper class for this bean. (internal server attribute, do not edit)

<entity-ref>

Specified the configuration for persisting an entity reference via it's 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 advances 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".

<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 *Oracle9iAS Containers for J2EE Services Guide.*

<jem-deployment>

Specifies an active EJB for deployment into the AC4J container.

Attributes:

- jem-name - An AC4J name that is used to identify the bean within the AC4J calls
- ejb-name - Identifies the EJB defined in the `ejb-jar.xml` file as an active EJB.

<jem-server-extension>

Describes the database server where the Databus is installed

Attributes:

- data-source-location - Provides the JNDI data source definition of the database where the Databus exists. The data source is configured in the `data-sources.xml` file.
- scheduling-threads - If greater than 1, then multiple OC4J threads can act in parallel. Default is 1.

<list-mapping>

Specifies a relational mapping of a List type. A List is a sequential (where order/index is important) Collection of items. The field containing the mapping must be of type java.util.List or the legacy types java.util.Vector or Type[].

Attributes:

- table - The name of the table in the database.

<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 etc.

<map-mapping>

Specifies a relational mapping of a Map type. A Map consists of n unique keys and their mapping to values. The field containing the mapping must be of type java.util.Map or the legacy types java.util.Hashtable or java.util.Properties.

Attributes:

- table - The name of the table in the database.

<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 + `"TopicConnectionFactories"` or `"QueueConnectionFactories"` + user defined name. The `xxxConnectionFactories` 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.
- transaction-timeout: This attribute controls the transaction timeout interval for any container-managed transactional MDB. The default is one day. If the transaction has not completed in this timeframe, the transaction is rolled back.

<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 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>` elment is used to declare the destination. Thus, the `<resource-env-ref-mapping>` element maps the destination object. See "Using Logical Names in the JMS JNDI Lookup" on page 7-15 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`.

<role-name>

The security role that the AC4J EJB methods are run under when using the `<run-as-specified-identity>` element.

<run-as-specified-identity>

You can specify that all methods of an AC4J 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 AC4J EJB methods under the specified security identity.

<security-identity>

Describes if the AC4J Databus should use the caller or run-as identity for the AC4J bean security.

<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". This is the default.
- 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 - The number of maximum bean implementation instances to be kept instantiated or pooled. The default is 100. This applies only to stateless session beans.
- min-instances - The number of minimum bean implementation instances to be kept instantiated or pooled. The default is 0. This applies only to stateless session beans.
- max-tx-retries—The number of times to retry a transaction that was rolled back due to system-level failures. The default is 3. 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 set this value to 0 and add retries only where errors are seen that could be resolved through a retry.

- 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—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.

<use-caller-identity>
You can specify that all methods of an AC4J EJB execute under the caller's identity.

<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/Map`. Setting this to true will optimize database

operations extensively. The default value is "true" for set-mapping and map-mappings and "false" for collection-mapping and list-mapping.

- type - The fully qualified class name of the type of the value. Examples are `com.acme.OrderEntry`, `java.lang.String`, and so on.

C

Third Party Licenses

This appendix includes the Third Party License for all the third party products included with Oracle9*i* Application Server. Topics include:

- Apache HTTP Server
- Apache JServ

Apache HTTP Server

Under the terms of the Apache license, Oracle is required to provide the following notices. However, the Oracle program license that accompanied this product determines your right to use the Oracle program, including the Apache software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the Apache software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or Apache.

The Apache Software License

```
/* ====================================================================
 * The Apache Software License, Version 1.1
 *
 * Copyright (c) 2000 The Apache Software Foundation.  All rights
 * reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 *
 * 3. The end-user documentation included with the redistribution,
 *    if any, must include the following acknowledgment:
 *       "This product includes software developed by the
 *        Apache Software Foundation (http://www.apache.org/)."
 *    Alternately, this acknowledgment may appear in the software itself,
 *    if and wherever such third-party acknowledgments normally appear.
 *
 * 4. The names "Apache" and "Apache Software Foundation" must
 *    not be used to endorse or promote products derived from this
 *    software without prior written permission. For written
 *    permission, please contact apache@apache.org.
 *
 * 5. Products derived from this software may not be called "Apache",
 *    nor may "Apache" appear in their name, without prior written
```

```
*    permission of the Apache Software Foundation.
*
* THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED
* WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
* OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
* DISCLAIMED.  IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR
* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
* LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
* USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
* ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
* OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
* OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
* ====================================================================
*
* This software consists of voluntary contributions made by many
* individuals on behalf of the Apache Software Foundation.  For more
* information on the Apache Software Foundation, please see
* <http://www.apache.org/>.
*
* Portions of this software are based upon public domain software
* originally written at the National Center for Supercomputing Applications,
* University of Illinois, Urbana-Champaign.
*/
```

Apache JServ

Under the terms of the Apache license, Oracle is required to provide the following notices. However, the Oracle program license that accompanied this product determines your right to use the Oracle program, including the Apache software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the Apache software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or Apache.

Apache JServ Public License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistribution of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistribution in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- All advertising materials mentioning features or use of this software must display the following acknowledgment:

  This product includes software developed by the Java Apache Project for use in the Apache JServ servlet engine project (http://java.apache.org/).
- The names "Apache JServ", "Apache JServ Servlet Engine" and "Java Apache Project" must not be used to endorse or promote products derived from this software without prior written permission.
- Products derived from this software may not be called "Apache JServ" nor may "Apache" nor "Apache JServ" appear in their names without prior written permission of the Java Apache Project.
- Redistribution of any form whatsoever must retain the following acknowledgment:

  This product includes software developed by the Java Apache Project for use in the Apache JServ servlet engine project (http://java.apache.org/).

THIS SOFTWARE IS PROVIDED BY THE JAVA APACHE PROJECT "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE JAVA

APACHE PROJECT OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Index

Symbols

<abstract-schema-name> element, 5-4, 5-7
<assembly-descriptor> element, B-20, B-21
<caller> element, B-21
<cascade-delete/> element, 4-9
<cmp-field-mapping> element, 4-24, 4-25, 4-27, 4-31, B-9, B-21
<cmr-field> element, 3-19, 4-6, 4-12, 4-27, 4-31
<cmr-field-name> element, 4-4, 4-6
<cmr-field-type> element, 4-6
<collection-mapping> element, 4-26, 4-27, 4-30, B-21
<context-attribute> element, B-22
<default-method-access> element, 8-36, B-20, B-22
<delay-updates-until-commit> attribute, B-23
<description> element, B-22
<ejb> element, 2-13
<ejb-link> element, 8-18, 8-19
<ejb-location> element, 6-11
<ejb-mapping> element, 8-19
<ejb-module> element, 8-8
<ejb-name> element, 8-19, B-22
<ejb-ql>, 5-4
<ejb-ql> element, 5-8
<ejb-ref> element, 8-2, 8-6, 8-19
<ejb-ref-mapping> element, 8-19, B-5, B-9, B-14, B-22
<ejb-ref-name> element, 8-2, 8-19, 8-20
<ejb-ref-type> element, 8-19
<ejb-relation> element, 4-6
<ejb-relation-name> element, 4-6
<ejb-relationship-role> element, 4-6
<ejb-relationship-role-name> element, 4-6
<enterprise-beans> element, B-3, B-23
<entity-deployment> element, 4-20, 4-24, 8-10, 8-12, B-7, B-8, B-23
<entity-ref> element, B-25
<env-entry> element, 8-15
<env-entry-mapping> element, B-5, B-9, B-14, B-25
<env-entry-name> element, 8-15
<env-entry-type> element, 8-15
<env-entry-value> element, 8-15
<fields> element, B-26
<finder-method> element, 5-3, B-9, B-26
<group> element, B-26
<home> element, 8-19
<ior-security-config> element, B-4, B-8, B-26
<java> element, 2-13
<jem-deployment> element, B-16, B-27
<jem-server-extension> element, B-16, B-27
<jndi-name> element, 8-19, 8-23, 8-25
<list-mapping> element, B-27
<lookup-context> element, B-27
<map-key-mapping> element, B-27
<map-mapping> element, B-28
<mapping> element, 8-19, 8-23, 8-25
<max-tx-retries> element, 8-11
<message-driven> element, 7-10
<message-driven-deployment> element, B-13, B-14, B-28
<method> element, B-18, B-19, B-28
  defined, 8-31
<method-intf> element, B-29
<method-name> element, 5-4, B-29
<method-param> element, 5-8, B-29
<method-params> element, B-29
<method-permission> element, 8-28, 8-29, 8-31

<module> element, 2-13
<multiplicity> element, 4-6
<orion-ejb-jar> element, B-3, B-29
<persistence-type> element, 6-11
<prim-key-class> element, 3-9, 6-4, A-8
<primkey-mapping> element, 4-28, B-8, B-29
<properties> element, B-29
<query> element, 5-2, 5-3, 5-4, 5-8
<query> element., 5-6
<relationship-role-source> element, 4-6
<relationships> element, 4-5, 4-26
<remote> element, 8-19
<res-auth> element, 8-23, 8-26
<resource-env-ref> element, 7-15
<resource-env-ref-mapping> element, B-5, B-9, B-14, B-30
<resource-ref> element, 6-11, 7-15
<resource-ref-mapping> element, 8-23, 8-25, B-5, B-9, B-14, B-30
<res-ref-name> element, 8-23, 8-25
<res-type> element, 8-23, 8-26
<result-type-mapping> element, 5-4
<role-link> element, 8-28, 8-29, 8-30
<role-name> element, 8-28, 8-29
<run-as> element, 8-33
<security-identity> element, 8-33
<security-role> element, 8-28, 8-29
<security-role-mapping> element, 8-34, 8-35, B-20, B-30
<security-role-ref> element, 8-28, 8-29
<session-deployment> element, B-4, B-31
<set-mapping> element, B-32
<unchecked/> element, 8-33
<use-caller-identity/> element, 8-34
<user> element, B-32
<value-mapping> element, B-32
<value-mapping> element, 4-28
<web> element, 2-13

A

AC4J, 10-1 to 10-62
Active Components for Java, see AC4J
application.xml file, 2-13, 7-4
  example, 2-14
  overview, 2-13
archiving
  directions, 2-12
  EAR file, 2-14
  EJBs, 2-12
associateUsingThirdTable option, 4-30

B

bean
  accessing remotely, 1-8
  activation, 1-12
  creating, 2-4, 3-3, A-2
  environment, 1-14
  implementation, 2-8
  interface, 1-7
  overview, 1-1
  passivation, 1-12
  removal, 2-15
  steps for invocation, 1-8
bean-managed persistent, see BMP
BMP
  create database tables, 6-12
  creation process, 6-2
  defined, 6-1
  deployment descriptor, 6-11
  ejbCreate implementation, 6-3
  home and remote interfaces, 6-3
  implementation details, 6-3
  persistence, 1-20

C

cache-timeout attribute, B-15
called-by attribute, B-21
caller-identity attribute, B-21
call-timeout attribute, B-5, B-9, B-23, B-31
ClassCastException, 8-8, 8-44
clustering, 9-1 to 9-9
  concurrency mode effect, 8-13
  deploying application to all nodes, 9-8
clustering-schema attribute, B-23
CMP
  data types, 3-17
  overview, 1-21

persistence update configuration, 8-14
CMR
cardinality, 4-6
cascade delete option, 4-9
default mapping, 4-11
define get/set methods, 4-4
deployment descriptor, 4-5
direction, 4-6
explicit relationship mapping, 4-19
many-to-many, 4-3, 4-7, 4-14
many-to-one, 4-3, 4-6
mapping relationships, 4-10
one-to-many, 4-3, 4-6, 4-14, 4-26, 4-30
one-to-one, 4-2, 4-6, 4-13, 4-22
relationship definition, 4-3
types of relationships, 4-2
Collections, 3-19
command-line options
performance settings, 8-38
concurrency modes, 8-10
clustering, 8-13
connection-factory-location attribute, B-28
context
session, 1-14
transaction, 1-14
copy-by-value attribute, B-6, B-10, B-23, B-31
create method, 2-15, 3-3, 3-4, 3-5, 6-2, A-2, A-3
EJBHome interface, 1-9, 2-4
CreateException, 2-5, 2-6

D

data types, 3-17
data-bus attribute, B-22
data-source attribute, B-10, B-23
DataSource object, 8-21
data-source-location attribute, B-27
data-sources.xml file, 6-11, 6-12
DBMS_AQADM package, 7-5
deadlock
recovery, 8-44
dedicated.connection setting, 8-38
dedicated.rmicontext setting, 8-38
DefineColumnType setting, 8-38
delay-updates-until-commit attribute, B-13
deployment descriptor, 1-9, 2-11, 3-3, 6-3, A-2
BMP, 6-11
EJB QL, 5-4
EJB reference, 8-16
entity bean, A-10, B-7
environment variables, 8-15
JDBC DataSource, 8-20
MDB, 7-3
message-driven bean, B-13
security, 8-28, 8-29, 8-35
session bean, B-5
destination-location attribute, B-28
disable-wrapper-cache attribute, 8-14, B-23
disble-wrapper-cache attribute, B-12
DNS round-robin, 8-7, 8-43
do-select-before insert attribute, B-23
do-select-before-insert attribute, B-10
DTD file, 2-11
dynamic cluster discovery, 9-3

E

EAR file, 2-1
creation, 2-14
EJB
archive, 2-12
client
setting JMS port, 8-6
setting RMI port, 8-6
clustering, 9-1 to 9-9
creating, 2-2, 2-4, 2-8, 3-3, A-2
deployment descriptor, 2-11
development suggestions, 2-2
difference between session and entity, 1-25
home interface, 2-4
JAR file, 3-4, 6-3, 7-4, A-2
local interface, 2-7
overview, 1-1
parameter passing, 1-10
referencing other EJBs, 8-8, 8-44
remote interface, 2-6
replication, 9-7
security, 8-27
setting pool size, 8-13
standalone client, 8-6

EJB QL
  ?1, 5-8
  deployment descriptor, 5-4
  DISTINCT keyword, 5-8
  documentation, 5-1
  finder method
    example, 5-5
    overview, 5-2
  input parameter syntax, 5-8
  overview, 5-2
  query methods, 5-2
  select method
    example, 5-7
    overview, 5-3
  statement example, 5-5, 5-6
EJB Query Language, see EJB QL
ejbActivate method, 1-12, 1-19, 6-3, 6-9, 6-10
EJBContext interface, 1-13
ejbCreate method, 1-18, 1-19, 1-21, 2-4, 3-3, 6-2, 6-3, A-2
  initializing primary key, 6-3
  MDB, 7-3
  SessionBean interface, 1-12
EJBException, 2-5, 2-6, 2-7
ejbFindByPrimaryKey method, 1-21, 6-3, 6-7, A-2
EJBHome interface, 2-4, 2-5, 3-3, 3-4, 6-2, A-2
  create method, 3-3, 3-4, 3-5, 6-2, A-2, A-3
  findByPrimaryKey method, 3-3, 3-5, 6-2, A-2, A-3
ejb-jar.xml file, 2-11, 6-11
ejbLoad method, 1-18, 1-20, 1-21, 1-22, 6-3, 6-9
EJBLocalHome interface, 2-4, 2-6, 3-3, 3-4, 6-2
EJBLocalObject interface, 2-4, 2-7, 3-3, 3-5, 6-2
ejb-name attribute, B-27
EJBObject interface, 2-4, 2-6, 3-3, 3-5, 6-2, A-2, A-4
ejbPassivate method, 1-12, 1-19, 6-3, 6-9
ejbPostCreate method, 1-18, 1-21, 3-3, 6-2, A-2
ejb-reference-home attribute, B-21
ejbRemove method, 1-12, 1-18, 1-20, 1-21, 6-10
  MDB, 7-3
ejbStore method, 1-18, 1-20, 1-21, 6-2, 6-8
Enterprise Archive file, see EAR file
Enterprise Java Beans, see EJB
entity bean
  class implementation, 3-6, A-4
  clustering, 9-4
  context information, 1-19
  creating, 1-19, 3-3, 3-4, A-2, A-3
  deploy, A-10
  deployment descriptor, B-7
  finder methods, 3-4, 6-3, A-3
  home interface, 3-4, A-3
  overview, 1-11, 1-16
  persistent data, 1-17, 1-20
  primary key, 1-17
  relationships, see CMR
  remote interface, 3-5, A-4
  removing, 1-20
EntityBean interface, 1-10, 1-17, 1-21, 2-4, 3-3, 6-2, A-2
  ejbActivate method, 1-19, 6-3
  ejbCreate method, 1-18, 1-19, 1-21
  ejbFindByPrimaryKey method, 1-21, A-2
  ejbLoad method, 1-18, 1-20, 1-21, 1-22, 6-3
  ejbPassivate method, 1-19, 6-3
  ejbPostCreate method, 1-18
  ejbRemove method, 1-18, 1-20, 1-21
  ejbStore method, 1-18, 1-20, 1-21, 6-2
  setEntityContext method, 1-18, 1-19, 1-22
  unsetEntityContext method, 1-19
environment references
  URL, 8-25
environment, retrieval, 1-14
exclusive-write-access attribute, 8-12, B-10, B-23

F

findByPrimaryKey method, 3-3, 6-2, A-2
finder method
  backwards compatibility, 5-3
  EJB QL example, 5-5
  overview, 5-2
finder methods, 6-3
  BMP, 6-7
  entity bean, 3-4, A-3
  findByPrimaryKey method, 3-5, A-3
force-update attribute, B-13

G

getEJBHome method, 1-14
getEnvironment method, 1-14
getRollbackOnly method, 1-14
getUserTransaction method, 1-14
global-thread-pool element, 8-39

H

home interface
  creating, 2-4, 3-3, 6-2, A-2
  lookup, 2-15
  overview, 1-8, 1-9

I

immutable attribute, B-32
impliesAll attribute, 8-36, B-31
InitialContext, 8-38
instance-cache-timeout attribute, B-10, B-23
isCallerInRole method, 8-29
isolation attribute, 8-10, B-11, B-24
isolation modes, 8-10

J

JAR
  archiving command, 2-12
jar command, 2-12
JAR file, 3-4, 6-3, 7-4, A-2
  EJB, 2-12
Java mail
  Session object, 8-22
jem-name attribute, B-27
JMS
  handled by MDB, 1-23
  port, 8-6
JNDI
  lookup, 2-15

L

listener-threads attribute, 7-12, B-15, B-28
Lists, 3-19
load balancing, 9-9
  DNS round-robin, 8-7
LoadBalanceOnLookup property, 9-9
local home interface
  example, 2-6
local interface
  creating, 2-7
  example, 2-8
local-wrapper attribute, B-7, B-13, B-24, B-31
location attribute, B-6, B-10, B-24, B-27, B-30, B-31
locking-mode attribute, 8-12, B-11, B-24

M

mail
  Session object, 8-22
mapping
  relationships, 4-19
max-instances attribute, 8-14, B-6, B-11, B-15, B-24, B-31
max-tx-retries attribute, B-6, B-12, B-24, B-31
MDB
  configuration, 7-10
  creation, 7-3
  deployment descriptor, 7-3
  overview, 1-11, 1-23, 7-1
  performance, 7-12, B-28
  transaction timeout, 7-13, B-28
message-driven bean
  deployment descriptor, B-13
Message-Driven Beans, see MDB
MessageDrivenBean interface, 1-24, 7-3
  setMessageDrivenContext method, 7-3
MessageListener interface, 1-24, 7-3
  onMessage method, 7-3
min-instances attribute, 8-14, B-6, B-11, B-15, B-24, B-31

N

name attribute, B-6, B-12, B-15, B-24, B-28, B-31, B-32
narrowing, 2-15

O

onMessage method, 1-24, 7-3
optimisitic concurrency mode, 8-11
optimistic concurrency mode, B-11, B-24
ORA-8177 exception, 8-13
oracle.dms.gate setting, 8-38

P

packaging
  referenced EJB classes, 8-8, 8-44
parameters
  object types, 1-10
  passing conventions, 1-10
parent application, 8-9
partial attribute, B-26
pass by reference, 1-10
pass by value, 1-10
performance setting
  command-line options, 8-38
  dedicated.connection, 8-38
  dedicated.rmicontext, 8-38
  DefineColumnType, 8-38
  DNS load balancing option, 8-7, 8-43
  oracle.dms.gate, 8-38
  statement caching, 8-42
  task manager granularity, 8-42
  thread pools, 8-39
performance settings, 8-38 to 8-43
permissions, 8-27
persistence
  bean-managed, 1-20
  container-managed, 1-21
  container-managed vs. bean-managed, 1-22
  create database tables, 6-12
  data management, 1-19
  field modification, 8-14
  managing, 3-4, A-2
  managing in BMP, 6-3
  overview, 1-17
persistence-filename attribute, B-6, B-32
persistence-name attribute, 4-25, 4-31, B-21
persistence-type attribute, B-21
pessimistic concurrency mode, B-11, B-24
pessimistic conncurrency mode, 8-11
pool
  setting size, 8-13
pool-cache-timeout attribute, B-5, B-12, B-25, B-31
PortableRemoteObject
  narrow method, 2-15
primary key, 3-3, 6-2, A-2
  complex class, 6-6
  complex definition, 6-4
  creating, 6-3
  entity bean, 1-21, 3-9, A-8
  management, 1-19
  overview, 1-17, 3-9, A-8
  simple definition, 6-4
PropertyPermission, 8-27

Q

query attribute, B-26

R

read-only concurrency mode, 8-12, B-11, B-24
remote home interface
  example, 2-5
remote interface
  business methods, 2-15
  creating, 2-4, 2-6, 3-3, 6-2, A-2
  example, 2-7
  overview, 1-8, 1-9
RemoteException, 2-5, 2-7
remove method, 2-15
  EJBHome interface, 1-9
RMI
  port, 8-6
RMILBInitialContextFactory, 8-7, 8-43
runAs security identity, 8-33
RuntimePermission, 8-27

S

scheduling-threads attribute, B-27
security, 8-27
  permissions, 8-27
SecurityException, B-16

security-identity element, B-30
select method
  EJB QL example, 5-7
  overview, 5-3
Serializable interface, 1-11
session bean
  class implementation, 1-10
  context, 1-12
  deployment descriptor, B-4, B-5
  local home interface, 2-6
  methods, 1-12
  overview, 1-11
  remote home interface, 2-5
  removing, 1-12
  stateful, 1-8, 1-15
  stateless, 1-8, 1-14
Session object, 8-22
SessionBean interface, 1-10
  EJB, 1-11, 2-4
  ejbActivate method, 1-12
  ejbCreate method, 1-12
  ejbPassivate method, 1-12
  ejbRemove method, 1-12
  setSessionContext method, 1-12
SessionContext
  interface, 1-13
setEntityContext method, 1-18, 1-19, 1-22
setMessageDrivenContext method, 1-24, 7-3
setRollbackOnly method, 1-14
setSessionContext method, 1-12, 1-20
setStmtCacheSize method, 8-42
SocketPermission, 8-27
stateful session bean
  clustering, 9-4
  overview, 1-15
stateless session bean
  clustering, 9-4
  overview, 1-14
statement caching
  DataSource
    statement caching, 8-42
static cluster discovery, 9-2
stmt-cache-size attribute, 8-42
subscription-name attribute, B-15, B-28

T

table attribute, B-12, B-25, B-27, B-28
task manager granularity, 8-42
taskmanager-granularity attribute, 8-42
thread
  pooling, 8-39
TimedOutException, B-5, B-9
timeout attribute, B-7, B-32
transaction
  commit, 1-14
  context propagation, 1-14
  retrieve status, 1-14
  rollback, 1-14
TRANSACTION_READ_COMMITTED, 8-10
TRANSACTION_SERIALIZABLE, 8-10
transaction-timeout attribute, 7-13, B-15, B-28
type attribute, B-28, B-33

U

unsetEntityContext method, 1-19, 1-22
update-changed-fields-only attribute, 8-14, B-12, B-25

V

validity-timeout attribute, B-13, B-25

W

wrapper attribute, B-7, B-13, B-25, B-32

X

XML
  BMP, 6-11
  deployment descriptor, 3-3, 6-3, A-2