Oracle® Containers for J2EE

Orion CMP Developer's Guide

10*g* Release 3 (10.1.3.1)

B28220-01

September 2006

ORACLE®

Oracle Containers for J2EE Orion CMP Developer's Guide, 10*g* Release 3 (10.1.3.1)

B28220-01

Primary Author: Liza Rekadze

Contents

Preface

This guide gets you started building EJB 2.0 entity beans with container-managed persistence for Oracle Containers for J2EE (OC4J), Release 2 (10.1.2) or earlier, using the Orion persistence manager. It includes code examples to help you develop your application.

The Orion persistence manager is deprecated. Oracle recommends that you use OC4J and the TopLink persistence manager for new development (see *Oracle Containers for J2EE Enterprise JavaBeans Developer's Guide*). Using the migration tool (see "Migrating to the TopLink Persistence Manager" on page 3-3), you can easily migrate an existing OC4J application that uses EJB 2.0 entity beans with the Orion persistence manager to use EJB 2.0 entity beans with the TopLink persistence manager.

Note: In this guide, the term *entity bean(s) with container-managed persistence* refers to the Orion entity bean(s) with container-managed persistence, assuming that all of the following conditions apply:

- You use entity beans with container-managed persistence with the Release 2 (10.1.2) or earlier of OC4J.
- Your CMP object-relational mapping is defined in `orion-ejb-jar.xml` deployment descriptor.

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

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

Audience

Anyone developing EJB 2.0 entity beans with container-managed persistence for the Release 2 (10.1.2) or earlier of OC4J using the Orion persistence manager will benefit from reading this guide. Written especially for programmers, it will also be of value to architects, systems analysts, project managers, and others interested in EJB applications deployed to OC4J.

This guide assumes that you already have a working knowledge of J2EE and the EJB 2.0 specification.

Documentation Accessibility

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

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

Accessibility of Code Examples in Documentation

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

Accessibility of Links to External Web Sites in Documentation

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

TTY Access to Oracle Support Services

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

Related Documents

For more information, see the following documents in the Oracle Containers for J2EE 10*g* Release 3 (10.1.3.1) documentation set:

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

- EJB design patterns: http://java.sun.com/blueprints/patterns/.

Conventions

The following text conventions are used in this document:

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

1

Understanding Entity Beans With Container-Managed Persistence

This chapter introduces the concept of an entity bean in general, and entity bean with container-managed persistence in particular.

This chapter includes information on the following topics:

What is an Entity Bean?

Within a J2EE SDK environment, an entity bean represents a business object in a relational database. Usually, an entity bean has an underlying table in a relational database, and every instance of the bean corresponds to a row in that database.

Entity beans possess the following characteristics:

- They are persistent
- They allow shared access
- They have primary keys
- They may participate in relationships with other entity beans

For more information about entity beans, see the following:

- *The J2EE Tutorial* at `http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBConcepts4.html#62950`
- *EJB 2.0 specification* at `http://java.sun.com/products/ejb/docs.html`

This section elaborates on the following topics specific to entity beans with container-managed persistence:

Entity Beans With Container-Managed Persistence

Terminologically, container-managed persistence is the process of the EJB container handling the database access required by an entity bean. That is, the bean's code does not contain SQL calls. This makes the bean's code oblivious to the underlying database. You do not have to implement some of the callback methods (see "Callback

Methods" on page 1-4) to manage persistence for your bean's data, because the container stores and reloads your persistent data to and from the database. In addition, you do not have to provide management for the primary key as the container provides this key for your bean (see "Configuring Primary Key" on page 5-1).

In Orion entity beans with container-managed persistence, the Orion persistence manager is used by the EJB container to persist data to the database from the entity beans. See "Preface" for details on the terminology.

For more information about persistence managers, see the following:

- "Persistence Manager" on page 3-2
- "Orion Persistence Manager" on page 3-2

An entity bean's deployment descriptor contains an abstract schema that defines the bean's persistent fields (see "Container-Managed Persistent Fields" on page 1-2) and relationships (see "Container-Managed Relationships" on page 1-3).

> **Note:** Do not confuse the terms abstract schema and physical schema: in a relational database, a physical schema consists of tables and columns, whereas an abstract schema is a provider of the information required by the container in order to produce the data access calls.

This section describes the following:

- Container-Managed Persistent Fields
- Container-Managed Relationships
- Callback Methods

Container-Managed Persistent Fields

Persistent fields of an entity bean define the state of the bean. These fields are stored in the underlying database. The container performs an automatic synchronization of the bean's state with the database at run time. At deployment time, the container usually maps the entity bean to a database table with the persistent fields mapped to the table's columns.

Persistent fields are virtual in container-managed persistence. This means that you declare them in the abstract schema, but do not make them instance variables in your entity bean class. However, you provide getters and setters for the persistent fields in your code, similar to the following:

```
public abstract String getEmployeeName() throws RemoteException

public abstract String getAddress() throws RemoteException

public abstract void setAddress(String newAddress) throws RemoteException
```

The container provides the implementation of these methods.

In the preceding example, the corresponding persistence fields are `employeeName` and `address`.

The EJB deployment descriptor in a form of an `ejb-jar.xml` file declares these fields as persistent. Each field name must be defined in a `<cmp-field><field-name>` element of the deployment descriptor, similar to the following:

```
<entity>
    ...
    <cmp-field><field-name>employeeName</cmp-field></field-name>
    <cmp-field><field-name>address</cmp-field></field-name>
    ...
</entity>
```

For more information on how to configure persistence fields, see "Configuring Container-Managed Persistent Fields" on page 5-10.

For more information on how to implement entity beans, see "Implementing an EJB 2.0 Entity Bean With Container-Managed Persistence" on page 4-1.

Container-Managed Relationships

An entity bean may be related to other entity beans. This behavior of entity beans is similar to the behavior of tables in a relational database.

The EJB container is responsible for relationships in entity beans with container-managed persistence.

OC4J supports the following types of container-managed relationships (CMR):

- One-to-one: an instance of an entity bean is related to an instance of another entity bean. For example, in a hypothetical model of a racing track with each car containing one driver, `CarEJB` and `DriverEJB` are in a one-to-one relationship.
- One-to-many: an instance of an entity bean may be related to more than one instance of another entity bean. Reusing the racing track example from the previous item, the racing track accommodates multiple competing cars, making `RacingTrackEJB` have a one-to-many relationships with `CarEJB`.
- Many-to-one: multiple instances of an entity bean may be related to one instance of another entity bean. For example, reversing the one-to-many racing track scenario, many cars compete in a single racing track, therefore `CarEJB` has a many-to-one relationship with `RacingTrackEJB`.
- Many-to-many: instances of an entity bean may be related to multiple instances of each other. For example, on a racing track, each car can be served by multiple mechanics, and every mechanic serves more than one car. This makes `CarEJB` and `MechanicEJB` have a many-to-many relationship.

An entity bean represents a "one" side of a relationship by the local bean object, whereas the "many" side is represented using a Java `Collection`.

It is the responsibility of the EJB container to maintain the referential integrity of these entity bean relationships.

For more information, see the following:

- Configuring Default Mapping of Relationship Fields to the Database
- Configuring Explicit Mapping of Relationship Fields to the Database

Direction in CMR

There is a notion of direction in CMR: a relationship may be either bidirectional or unidirectional. In a bidirectional relationship, each entity bean has a relationship field that refers to the other bean. Through the relationship field (see "Relationship Fields" on page 1-4), an entity bean's code can access its related object. In a unidirectional relationship, only one entity bean has a relationship field that refers to the other.

EJB QL queries (see "Understanding EJB QL" on page 1-6) have an ability to navigate across relationships. The direction of a relationship determines whether or not a query can navigate from one bean to another.

Relationship Fields

A relationship field identifies a related bean. A relationship field behaves similar to a foreign key in a database table.

Like persistent fields (see "Container-Managed Persistent Fields" on page 1-2), relationship fields are virtual and are defined in the enterprise bean class with access methods. But to the contrary to persistent fields, relationship field do not represent the bean's state.

For more information about relationship fields, see "Configuring Container-Managed Relationship Fields" on page 5-14.

Callback Methods

There are a number of so-called callback methods that you must implement to enable the life cycle of your entity bean with container-managed persistence. Some of these methods are defined in the `javax.ejb.EntityBean` interface, whereas the implementation of others (`ejbCreate` and `ejbPostCreate`) is mandated by the EJB 2.0 specification.

The container invokes the callback methods at the designated times; you may or may not add logic to these methods.

The following table provides each method's details, as well as lists the implementation requirements for the callback methods of the entity bean class:

Callback Method	Details and Implementation Requirements
`setEntityContext`	This method 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 the `unsetEntityContext` method.
`unsetEntityContext`	This method unsets the associated entity context and releases any resources allocated in the `setEntityContext` method.
`ejbCreate`	An entity bean can contain one or more `ejbCreate` methods. Typically, this method performs the following: ▪ Inserts the entity state into the database. ▪ Initializes the instance variables. ▪ Returns the primary key. When writing this method, you must insure the following: ▪ The access control modifier is `public`. ▪ The return type is the primary key. ▪ The arguments are legal types for the Java RMI API. ▪ The method modifier is neither `final` or `static`.

Callback Method	Details and Implementation Requirements
`ejbPostCreate`	You must write an `ejbPostCreate` method for each `ejbCreate` method in your entity bean class. The container invokes `ejbPostCreate` method immediately after it calls `ejbCreate` method. In most cases, leave your `ejbPostCreate` method empty. When writing this method, you must insure the following: ■ The number and types of arguments match a corresponding `ejbCreate` method. ■ The access control modifier is `public`. ■ The method modifier is neither `final` or `static`. ■ The return type is `void`.
`ejbActivate`	This method gives the entity bean instance a chance to acquire additional resources that it needs while it is in the ready state. You must at least provide an empty implementation of this callback method.
`ejbPassivate`	This method gives the entity bean instance the chance to release any resources that should not be held while the instance is in the pool (typically, these resources had been allocated during the execution of the `ejbActivate` method). You must at least provide an empty implementation of this callback method. You also may choose to add logic for performing any cleanup functionality.
`ejbRemove`	This method deletes the entity state from the database. The container calls this method after a client has deleted an entity bean (by invoking the remove method). You must at least provide an empty implementation of this callback method. You also may choose to add logic for performing any cleanup functionality. Note that an entity bean may also be removed directly by a database deletion. For example, if a SQL script deletes a row that contains an entity bean state, then that entity bean is removed.
`ejbLoad`	This method refreshes the instance variables from the database. If the EJB container needs to synchronize the instance variables of an entity bean with the corresponding values stored in a database, it invokes the `ejbLoad` and `ejbStore` methods. Note that the client may not call `ejbLoad` and `ejbStore` methods. 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.
`ejbStore`	This method writes the variables to the database. If the EJB container needs to synchronize the instance variables of an entity bean with the corresponding values stored in a database, it invokes the `ejbLoad` and `ejbStore` methods. Note that the client may not call `ejbStore` method. No functionality 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	Details and Implementation Requirements
`ejbFindByPrimaryKey`	As its name implies, this method accepts as an argument the primary key, which it uses to locate an entity bean. When writing this method, you must insure the following: ■ The access control modifier is `public`. ■ The method modifier is neither `final` or `static`. ■ The return type is a primary key or a collection of primary keys. 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.

Querying for an Entity Bean

You query for an instance of an entity bean using entity bean finder (see "Understanding Finder Methods" on page 1-8) or select (see "Understanding Select Methods" on page 1-8) methods.

You express your selection criteria using an appropriate query syntax (see "Understanding Query Syntax" on page 1-7).

This section describes the following:

- Understanding EJB QL
- Understanding Query Syntax
- Understanding Finder Methods
- Understanding Select Methods

Understanding EJB QL

EJB Query Language (EJB QL) is a specification language used to define semantics of finder and select methods in a portable and optimizable format. The Orion persistence manager can perform an automatic compilation of EJB QL. Your responsibility is to define EJB QL queries in the deployment descriptor of the entity bean.

Using EJB QL offers the following advantages:

- You do not need to know the database structure (such as tables and fields).
- You can construct queries using the attributes of the entity beans instead of using database tables and fields.
- You can use relationships in a query to provide navigation from attribute to attribute.
- EJB QL queries are portable because they are database-independent.
- You can specify the reference class in the `SELECT` clause.

The disadvantage of EJB QL queries is that it is difficult to use when you construct complex queries.

EJB QL has the following restrictions:

- Comments are not allowed.

- Date and time values are in milliseconds and use a Java `long`. A date or time literal should be an integer literal. To generate a millisecond value, you may use the `java.util.Calendar` class.
- In EJB 2.0 container-managed persistence does not support inheritance. For this reason, two entity beans of different types cannot be compared.

Understanding Query Syntax

You can express an entity bean query using EJB QL (see "Understanding EJB QL" on page 1-6) or SQL native to your underlying relational database.

EJB QL is preferred for it is portable and optimizable.

An EJB QL query has three clauses: `SELECT`, `FROM`, and `WHERE`. The `SELECT` and `FROM` clauses are required, but the `WHERE` clause is optional. The following is the high-level syntax of an EJB QL query:

```
EJB QL ::= select_clause from_clause [where_clause]
```

The `SELECT` clause defines the types of the objects or values returned by the query. A return type is either a local interface, a remote interface, or a persistent field.

The `FROM` clause defines the scope of the query by declaring one or more identification variables, which may be referenced in the `SELECT` and `WHERE` clauses. An identification variable represents one of the following elements:

- The abstract schema name of an entity bean.
- A member of a collection that is the multiple side of a one-to-many relationship.

The `WHERE` clause is a conditional expression that restricts the objects or values retrieved by the query. Although optional, most queries have a `WHERE` clause.

For more information about EJB QL syntax, consult *EJB 2.0 specification* at `http://java.sun.com/products/ejb/docs.html`.

The EJB QL query syntax is demonstrated by the following example: suppose, you declare the `findByZipCode` method in your entity bean's home interface to obtain all the `Employee` beans with a certain zip code:

```
public Collection findByZipCode(String zipCode) throws RemoteException,
CreateException
```

That would be expressed by using the following EJB QL statement in the deployment descriptor:

```
FROM contactInfo WHERE contactInfo.zip = ?1
```

This EJB QL statement says "select all the `Employee` beans that have a zip code equal to the `zipCode` argument." The `SELECT` clause, which indicates what to select, is not needed in the EJB QL statements for find methods. That is because the finder methods will always select the remote references of its own bean type.

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

You can use EJB QL in finder (see "Understanding Finder Methods" on page 1-8) and select (see "Understanding Select Methods" on page 1-8) methods. To use native SQL, you must use straight JDBC calls.

Understanding Finder Methods

An EJB finder is a query, as defined by the EJB 2.0 specification. An EJB finder retrieves (and returns) entity bean references, whereas a query returns Java objects.

Finders contain finder methods (`ejbFind`) that define search criteria.

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.

> **Note:** Finder methods are exposed to the client.

For information on how to define and implement finder methods, see "Implementing EJB QL Finder Methods" on page 6-1.

Understanding Select Methods

Select methods (`ejbSelect`) are used primarily to return values of container-managed persistent (see "Container-Managed Persistent Fields" on page 1-2) or relationship (see "Relationship Fields" on page 1-4) fields. All values are returned in their own object type; any primitive types are wrapped in objects that have similar functions (for example, a primitive `int` type is wrapped in an `Integer` object). An `ejbSelect` method is a query method intended for internal use within an entity bean instance. Specified in the abstract bean itself, the `ejbSelect` method is not directly exposed to the client in the home or component interface. Defined as abstract, each bean can include zero or more such methods.

Select methods have the following characteristics:

- The method name must have `ejbSelect` as its prefix.
- It must be declared as `public`.
- It must be declared as `abstract`.
- The `throws` clause must specify the `javax.ejb.FinderException`, although it may also specify application-specific exceptions as well.
- Under EJB 2.0, the `result-type-mapping` tag in the `ejb-jar.xml` file determines the return type for `ejbSelect` methods. Set the flag to `Remote` to return `EJBObject`; set it to `Local` to return `EJBLocalObject`.

For more information about select methods, see "Implementing EJB QL Select Methods" on page 6-6.

Avoiding Database Resource Contention

Entity beans concurrency and database isolation levels let you avoid the resource contention and prevent the users from overwriting each others changes to database while allowing concurrent execution.

This section discusses the following topics:

- Entity Bean Database Isolation Levels and Resource Contention
- Entity Bean Concurrency Modes and Resource Contention
- Combining Entity Bean Database Isolation Level and Concurrency Mode
- Entity Bean Concurrency Modes and Clustering

Entity Bean Database Isolation Levels and Resource Contention

The `java.sql.Connection` object represents a connection to a specific database. The `Connection` provides four database isolation levels 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.

Oracle supports the following isolation levels:

- `transaction_read_committed`: Dirty reads are prevented; nonrepeatable 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, nonrepeatable reads and phantom reads are prevented. This level includes the prohibitions in `transaction_repeatable_read` and further prohibits the occurrence of the following hypothetical situation: one transaction reads all rows that satisfy a `WHERE` condition; a second transaction inserts a row that satisfies that `WHERE` condition; the first transaction rereads for the same condition, retrieving the additional "phantom" row in the second read.

For more information about database isolation levels, see "Configuring Database Isolation Levels" on page 5-24.

Entity Bean Concurrency Modes and Resource Contention

OC4J also provides concurrency modes for handling resource contention and parallel execution within entity beans with container-managed persistence. The concurrency modes determine when to block to manage resource contention, or when to execute in parallel.

The following is the list of the available concurrency modes:

- pessimistic: Manages resource contention and does not permit parallel execution. Only one user can execute the entity bean at a given time.
- optimistic: Does not monitor resource contention. Thus, the database isolation levels manage the concurrency. Multiple users can execute the entity bean in parallel.
- read-only: The container does not permit any updates to the bean's state. Multiple users can execute the entity bean in parallel.

For more information about concurrency modes, see "Configuring Concurrency Modes" on page 5-25.

You can specify both entity bean concurrency modes and database isolation levels, if the combination affects the outcome of your resource contention. See "Combining Entity Bean Database Isolation Level and Concurrency Mode" on page 1-9 for more information.

Combining Entity Bean Database Isolation Level and Concurrency Mode

The setting of the database isolation level has no bearing on the `pessimistic` and `read-only` concurrency modes. The isolation levels only matter if an external source is modifying the database.

If you choose `optimistic` concurrency mode with `committed` database isolation level, you may lose an update. If you choose `optimistic` concurrency mode with `serializable` isolation level, you will never lose an update resulting in the constant

consistency of your data. However, you can receive an `ORA-8177` exception as a resource contention error.

Differences Between Pessimistic, Optimistic and Serializable Settings

An entity bean with the `pessimistic` concurrency mode does not permit the execution by multiple clients (either on the same or on different instances of the same primary key). Only one client at a time can execute the instance.

An entity bean with the `optimistic` concurrency mode allows multiple instances of the bean to execute in parallel. This might result in lost updates (and conflicts), because two different transactions may update the same row simultaneously.

Setting the transaction isolation level to `serializable` enables the detection of conflicts when they occur. Currently, if an update from one of the transactions raises a `SQLException`, that transaction is rolled back.

Optionally, you may set the `tx-retries` attribute of the `<entity-deployment>` element to a value greater than 1, which would result in the retry of the transaction. See Table A–1, " Attributes of the <entity-deployment> Element" on page A-4 for more information about the `<entity-deployment>` element and its attributes.

Entity Bean Concurrency Modes and Clustering

All concurrency modes behave in a similar manner whether they are used within an independent 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, when one instance tries to execute the database locks out all other instances.

For more information, see the following:

- "Clustering Overview" in the Oracle Containers for J2EE Configuration and Administration Guide
- "Application Clustering in OC4J" in the Oracle Containers for J2EE Configuration and Administration Guide
- "Oracle Application Server Cluster (OC4J) in Active-Active Topologies" in the *Oracle Application Server High Availability Guide*

When to Use an Entity Bean With Container-Managed Persistence?

You should consider using an entity bean with container-managed persistence under the following conditions:

- The bean represents a business entity, not a procedure.
- The bean's state is persistent: if the bean instance terminates or if the J2EE server shuts down, the bean's state still exists in a database.
- The EJB container that you use delegates the persisting services to the Orion persistent manager.

Note: Oracle recommends migrating your application to the TopLink persistence manager using the Oracle migration tool. See "Migrating to the TopLink Persistence Manager" on page 3-3 for more information.

2

Understanding Orion CMP Application Development

This chapter describes how you should approach the Orion CMP application development.

This chapter contains information on the following topics:

Developing, Packaging and Deploying EJB Applications

Typically, the EJB application development includes the following steps:

- Setting up the application directory structure. See "Understanding the EJB Application Directory Structure" on page 2-2 for more information.
- Implementing the component interface. See "Implementing the Entity Bean Component Interface" on page 4-3 for more information.
- Implementing the bean class. See "Implementing the Entity Bean Class" on page 4-4 for more information.
- Implementing the home interface. See "Implementing the Entity Bean Home Interface" on page 4-2 for more information.
- Writing the EJB deployment descriptor. For more information, see the following:
  - "Declaring the Component Interface in the Deployment Descriptor" on page 4-4.
  - "Defining the Entity Bean Class in the Deployment Descriptor" on page 4-6.
  - "Declaring the Home Interface in the Deployment Descriptor" on page 4-3.
  - Chapter 5, "Configuring an EJB 2.0 Entity Bean With Container-Managed Persistence" on page 5-1.
- Creating the front end.
- Writing the front-end-specific deployment descriptor (for example, Web).
- Writing the J2EE application definition (deployment descriptor) that combines the EJB module and the front end module into a J2EE application. This deployment descriptor defines all the modules of the application and the way they should be deployed. See "Packaging and Deploying EJB CMP Applications" on page 2-5 for more information.
- Compiling.

- Creating the following archive files:
  - JAR file–contains the EJB part of the application.
  - WAR file–Web part of the application.
  - EAR file– contains a complete deployable J2EE application.

  Note: You can use an Ant build file to compile all the `.java` files and create the `.jar`, `.war` and `.ear` files.

  For more information, see the following:
  - "Understanding the EJB Application Directory Structure" on page 2-2
  - "Packaging and Deploying EJB CMP Applications" on page 2-5

As a part of your application's deployment, you would have to install your application, bind the front end and start OC4J.

Understanding the EJB Application Directory Structure

Even though you can develop your application in any way you like, you are advised to use consistent naming in order to easily locate your application. One approach would be to implement your EJB application under a single parent directory structure, segregating each module of the application into its own subdirectory.

The directory structure contains all the application files, including manually coded, as well as generated files.

Example 2–1 shows the addressbook directory. You should create the directory somewhere in your home directory (for example, in `/home/jane.doe/projects/addressbook/`).

Example 2–1 Application Directory Structure

```
addressbook
addressbook/etc
addressbook/src
addressbook/src/java
addressbook/src/java/addressbook
addressbook/src/java/addressbook/ejb
addressbook/src/jsp
addressbook/src/web
```

All `.java` source files are placed under `addressbook/src/java`. The client files (JSP, in this case) are placed under `src/jsp`, and any `.html`, `.gif`, `.jpg` and `.css` files are put in `addressbook/src/web` directory.

The directory `addressbook/etc` contains all the necessary `.xml` configuration files.

Note: To create the directory structure, you can use either shell (.sh) (for UNIX and Linux systems) or batch (.bat) (for Windows and OS/2 systems) files similar to the following:

```
mkdir addressbook
mkdir addressbook\etc
mkdir addressbook\src
mkdir addressbook\src\java
mkdir addressbook\src\java\addressbook
mkdir addressbook\src\java\addressbook\ejb
mkdir addressbook\src\jsp
mkdir addressbook\src\web
```

Run your directory-making file in the directory where you plan to create the directory.

The generated files are in the `lib` and `build` directories. Note that these directories are not included in the directory structure shown in Example 2–1 as they are typically created by the build script.

The archives, when created, are placed under `addressbook/build/`.

Using EJB Development Tools

Typically, the EJB development tools include the following:

- The J2EE perspective: All of the EJB tools are accessible from the J2EE perspective. This perspective provides a layout in which the most commonly used actions, views, and wizards for J2EE and EJB development are easily accessible.
- Tools for importing existing EJB JAR files.
- Tools for creating enterprise beans and access beans: The EJB tools help you create enterprise beans (either with or without inheritance), including entity beans with container-managed persistence. The EJB deployment descriptor editor helps you set deployment descriptor and assembly properties for your enterprise beans.

  You can also accomplish complementary enterprise bean development activities, such as writing and editing business logic, importing or exporting enterprise beans, and maintaining both your enterprise bean source code and generated code using the built-in Java development tools, along with the team and versioning capabilities.

  You can also create access beans and add other attributes such as relationships. Access beans are Java bean wrappers for enterprise beans, which are typically used by client programs.
- Tools for building data persistence into enterprise beans: The EJB mapping tools help you map entity enterprise beans to back-end data stores, such as relational databases. There is support for top-down, bottom-up, and meet-in-the-middle mapping development. You can also create schemas and maps from existing EJB JAR files.
- Tools for generating deployment code: The EJB tools generate the deployment classes that allow your beans to run on an EJB server. You can launch wizards from the selected EJB projects or modules. These wizards provide lists of the enterprise beans that you can deploy (one or more at a time). These tools mask the complexities normally associated with creating deployment classes, such as generating RMI/IIOP stubs and EJB container-specific deployment code.

Support for entity beans with container-managed persistence is included in tools. The tools also let you create relational database tables for entity beans with container-managed persistence. After the deployment code is generated, you can export your enterprise beans to a JAR or EAR file for installation on an EJB server.

- Tools for validating your enterprise beans for specification compliance: The EJB tools validate that your enterprise bean code is consistent and that it conforms to the rules defined by the EJB specification.

  The EJB tools also automatically validate that access beans are constructed correctly and that they are consistent with their associated enterprise beans. Code validation usually occurs whenever you create or edit access beans.

Using JDeveloper

JDeveloper 9.0.3's features cover all aspects of EJB 2.0 development from conception to implementation.

JDeveloper is a versatile tool with many capabilities. With regards to EJB development, JDeveloper enables you to do the following:

- Develop EJB 2.0 entity beans with container-managed persistence using wizards.
- Reverse-engineer database tables as EJB 2.0 entity beans with container-managed persistence.
- Reverse engineer foreign key relationships in the database as EJB 2.0 entity beans with container-managed persistence.
- Use the EJB Module Editor (see Figure 2–2) to edit all the EJB in the `ejb-jar.xml` deployment descriptor.
- Employ the EJB Verifier to validate `ejb-jar.xml` deployment descriptor against DTD and to verify EJB classes for inconsistencies.
- Develop EJB visually in a UML class diagram with synchronization between UML, code and deployment descriptor (see Figure 2–1).
- Test EJB locally in the IDE using the built-in OC4J.
- Generate standard EJB deployment archives.
- Deploy easily to OC4J.
- Pass command-line options in deployment profiles.
- Add and assemble application deployment descriptors (see "Packaging and Deploying EJB CMP Applications" on page 2-5 and "Deploying the CMP EJB Application to OC4J" on page 2-8 for more information).
- Use the Deployment Descriptor Editor for `orion-web.xml` and `application-client.xml`.
- Import existing EAR files (see "Developing, Packaging and Deploying EJB Applications" on page 2-1) as workspaces in JDeveloper using EAR Import Wizard for OC4J.

Figure 2–1 shows a UML representation with CMR reverse-engineered from the database using JDeveloper's extensive EJB modeling capabilities.

Figure 2–1 JDeveloper-generated UML Diagram with CMR

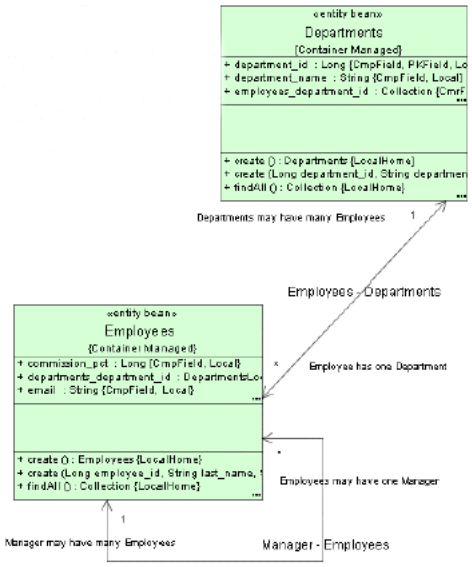
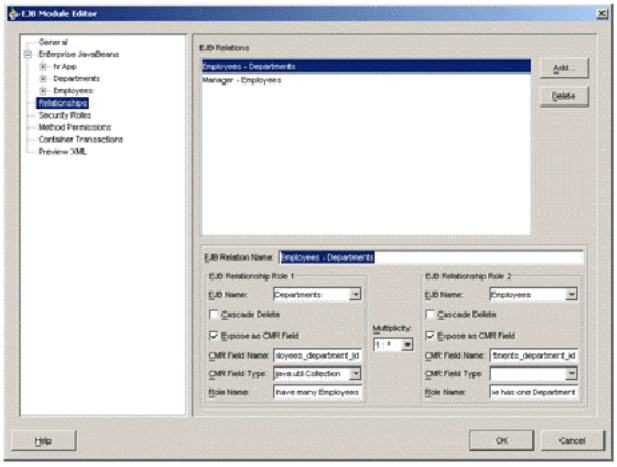


Figure 2–2 illustrates the EJB Module Editor that provides a common user interface for editing all the EJB in the `ejb-jar.xml` deployment descriptor. The Relationships panel in the editor lets you add, edit and delete the container-managed relationships between entity beans in an intuitive way.

Figure 2–2 JDeveloper's EJB Module Editor



Packaging and Deploying EJB CMP Applications

The following are the general steps for packaging and deploying an EJB application:

1. Create the deployment descriptor.

   After implementing and compiling your classes, you must create the standard EJB deployment descriptor for all entity beans with container-managed persistence in the module. The XML deployment descriptor defined in the `ejb-jar.xml` file (see "ejb-jar.xml File" on page 2-7) 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 (you can access this file at http://java.sun.com/dtd/ejb-jar_2_0.dtd).

Any OC4J services that you want to configure are also designated in the deployment descriptor. See *Oracle Containers for J2EE Services Guide* for information about the following container services:

- Data sources
- JTA
- JNDI
- JMS
- RMI and RMI/IIOP
- JCA
- Java Object Cache
- HTTPS

See *Oracle Containers for J2EE Security Guide* for information about the following container services:.

- Security
- CSv2

See *Oracle Application Server Web Services Developer's Guide* for information about Web services.

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, as Example 2–2 shows:

Example 2–2 META-INF Directory Containing the Application Deployment Descriptor

```
META-INF/
META-INF/application.xml
```

2. Archive the EJB application.

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

   For example, to archive your compiled EJB class files and XML files for the Addressbook example (see Example 2–1) into a JAR file, perform the following in the `.../addressbook/ejb` directory:

   ```
   % jar cvf addressbook-ejb.jar
   ```

   This archives all files contained in the `ejb` subdirectory within the JAR file:

   ```
   META-INF/
   META-INF/ejb-jar.xml
   addressbook/
   addressbook/ejb/AddressEntry.class
   addressbook/ejb/AddressBook.class
   addressbook/ejb/AddressEntryBean.class
   ```

3. Prepare the EJB application for assembly.

   To prepare the application for deployment, do the following:

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

The `application.xml` file acts as the manifest file for the application and contains a list of the modules that are included within your enterprise application. Use the `<module>` element defined in the `application.xml` file to designate what comprises your enterprise application, as Table 2–1 shows:

Table 2–1 Module Elements in the application.xml File

Element	Contents
`<ejb>`	EJB JAR file name.
`<web>`	WAR file name in the `<web-uri>` subelement, and its context in the `<context>` subelement.
`<java>`	Client JAR file name, if any.

As Example 2–2 demonstrates, 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, as follows:

```
META-INF/
META-INF/application.xml
addressbook-ejb.jar
addressbook-web.war
```

Because of the 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 file name.

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

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

To create the `addressbook.ear` file, execute the following in the `addressbook` directory shown in Example 2–1:

```
% jar cvf addressbook.ear
```

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

Understanding EJB Deployment Descriptor Files

The following are the EJB deployment descriptor files that you use in CMP EJB applications deployed to OC4J:

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

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

- an optional application assembly descriptor

When it is required, the `ejb-jar.xml` file describes EJB information applicable to any J2EE application server. This information may be augmented by application server-specific EJB deployment descriptor files (see "orion-ejb-jar.xml File" on page 2-8).

orion-ejb-jar.xml File The `orion-ejb-jar.xml` file is an EJB deployment descriptor file that contains all OC4J-proprietary options. This file extends the configuration that you specify in the `ejb-jar.xml` file (see "ejb-jar.xml File" on page 2-7).

For more information about `orion-ejb-jar.xml` deployment descriptor, see the following:

- Chapter 5, "Configuring an EJB 2.0 Entity Bean With Container-Managed Persistence" on page 5-1
- Appendix A, "XML Reference for orion-ejb-jar.xml Elements" on page A-1

Deploying the CMP EJB Application to OC4J

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

3

Understanding Orion CMP Support in OC4J

This chapter describes the two varieties of persistence managers that OC4J can use, and provides detailed instructions for migrating from one persistent manager (Orion) to another (TopLink).

This chapter includes information on the following topics:

EJB 2.0 Support

The OC4J EJB container provides complete support for EJB 2.0, which includes the full support of the following:

- session beans
- entity beans
- message-driven beans
- BMP
- CMP
- object-relational mapping

OC4J provides CMP implementation for entity beans supporting object-relational mapping. OC4J supports one-to-one, one-to-many, many-to-one and many-to-many object-relational mappings (see "Container-Managed Relationships" on page 1-3), including simple object-relational mapping for simple, primitive, and serializable objects, as well as complex object-relational mapping for compound objects, entity references, and collections.

You can use OC4J to persist EJB 2.0 entity beans using the Orion persistence manager (see "Orion Persistence Manager" on page 3-2).

> **Note:** The key difference between the Orion persistence manager (see "Orion Persistence Manager" on page 3-2) and the TopLink persistence manager (see "TopLink Persistence Manager" on page 3-2) is that the former is limited to the support of EJB 2.0 entity beans, whereas the latter supports EJB 2.1 entity beans and EJB 3.0 entities.

Persistence Manager

An EJB container uses the services of a persistent manager to persist data from entity beans to a database. The following is the list of responsibilities of the persistence manager:

- handle the process of persisting an entity bean with container-managed persistence automatically at run time;
- map an entity bean to the database based on a contract (abstract schema) between the bean and the persistence manager;
- implement and execute finder and select methods using EJB QL.

This section elaborates on the following topics:

Orion Persistence Manager

The main difference between the Orion persistence manager and the TopLink persistence manager (see "TopLink Persistence Manager" on page 3-2) is that the former only supports EJB 2.0 entity beans, whereas the latter supports EJB 2.1 entity beans, as well as EJB 3.0 entities.

The Orion persistence manager is deprecated and will not be supported in future releases. Oracle recommends that you use OC4J and the TopLink persistence manager for new development. Using the migration tool (see "Migrating to the TopLink Persistence Manager" on page 3-3), you can easily migrate an existing OC4J application that uses EJB 2.0 entity beans with the Orion persistence manager to use EJB 2.0 entity beans with the TopLink persistence manager (see "TopLink Persistence Manager" on page 3-2).

TopLink Persistence Manager

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

TopLink lets you build high-performance applications that store persistent object-oriented data in a relational database. It successfully transforms object-oriented data into either relational data or XML elements. Using TopLink, you can integrate persistence and object-transformation into your application, while staying focused on your primary domain problem by taking advantage of an efficient, flexible, and field-proven solution. The extensive suite of development tools that TopLink provides, including Oracle TopLink Workbench, lets you quickly capture and define object-to-data source and object-to-data representation mappings in a flexible, efficient metadata format. The TopLink runtime lets your application exploit this mapping metadata with a simple session facade that provides in-depth support for data access, queries, transactions, and caching.

The following are some of the key features of TopLink:

- Nonintrusive, flexible, metadata-based architecture
- Comprehensive visual TopLink Workbench

- Advanced mapping support and flexibility (relational, object-relational, EIS, and XML)
- Object caching support
- Query flexibility
- Just-in-time reading
- Caching
- Object-level transaction support and integration
- Locking
- Multiple performance tuning options
- Architectural flexibility

For additional information about TopLink, see the TopLink page on Oracle Technology Network at
http://www.oracle.com/technology/products/ias/toplink/index.html

> **Note:** Currently, TopLink persistence manager is the default persistence manager for OC4J. It has support for EJB 2.1 and EJB 3.0 Persistence API.

The following are some of the advantages of using OC4J with the TopLink persistence manager:

- They permit concurrent access to database tables and enable the avoidance of the database resource contention (see "Avoiding Database Resource Contention" on page 1-8).
- You can express selection criteria for an EJB 3.0 query or EJB 2.1 finder or select method using the TopLink query and expressions framework.
- You can take advantage of predefined and default finder and select methods.
- The TopLink persistence manager takes the query syntax you specify ("Understanding Query Syntax" on page 1-7) and generates SQL native to your underlying relational database.

Migrating to the TopLink Persistence Manager

Using the TopLink migration tool, you can easily migrate an existing OC4J application that uses EJB 2.0 entity beans with the Orion persistence manager to use EJB 2.0 entity beans with the TopLink persistence manager.

You can configure OC4J to use TopLink as the default persistence manager. In 10*g* Release 3 (10.1.3.1), OC4J is shipped configured to use TopLink as its default persistence manager.

> **Note:** You can only use one persistence manager for all the CMP EJB in a JAR file.

TopLink provides automated support for migrating an existing J2EE application to use TopLink as the persistence manager. In 10.1.3 release, Oracle provides a TopLink migration tool that you can use to automate this migration for Release 2 (9.0.4) or later OC4J installations. If you upgrade your OC4J to this new release, you must migrate

persistence configuration from your original `orion-ejb-jar.xml` file to the `toplink-ejb-jar.xml` file.

After using the TopLink migration tool, you may need to make some additional changes as described in "Post-Migration Changes" on page 3-9.

If you encounter problems during migration, see "Troubleshooting Your Migration" on page 3-10.

This section explains how to use the TopLink migration tool, including the following:

- Key Features of the TopLink Migration Tool
- Using the TopLink Migration Tool From the Command Line
- Post-Migration Changes
- Troubleshooting Your Migration

Key Features of the TopLink Migration Tool

Before using the TopLink migration tool, review this section to understand how the TopLink migration tool works and to determine what OC4J persistence manager metadata is, and is not, migrated.

Input and Output

The TopLink migration tool takes the following files as input:

- `ejb-jar.xml`
- `orion-ejb-jar.xml`

It migrates as much OC4J-specific persistence configuration as possible to a new `toplink-ejb-jar.xml` file and creates the following new files in a target directory you specify:

- `orion-ejb-jar.xml`
- `toplink-ejb-jar.xml`
- TopLink Workbench project file `TLCmpProject.mwp`

The `ejb-jar.xml` and `orion-ejb-jar.xml` files may be in an EAR, JAR, or just standalone XML files. If you migrate from standalone XML files (rather than an EAR or JAR file), ensure that the domain classes are accessible and included in your classpath.

The TopLink migration tool creates a new `orion-ejb-jar.xml` and `toplink-ejb-jar.xml` file to the target directory you specify in the same format as it reads the original files. For example, if you specify an EAR file as input, then the TopLink migration tool stages and creates a new EAR file that contains both the new `orion-ejb-jar.xml` and the new `toplink-ejb-jar.xml` file, but is otherwise identical to the original.

The TopLink Workbench project file is always created as a separate file.

Note: Oracle recommends that you make a backup copy of your `orion-ejb-jar.xml` file before using the TopLink migration tool.

Migration

As it operates, the TopLink migration tool logs all errors and diagnostic output to a log file named `oc4j_migration.log` in the output directory.

The TopLink migration tool processes descriptor, mapping, and query information from the input files as follows:

- It builds a TopLink descriptor object for each entity bean and migrates native persistence metadata like mapped tables, primary keys, and mappings for container-managed persistent and relationship fields.
- It builds a TopLink mapping object for every container-managed persistent and relationship field of an entity bean and migrates native persistence metadata, such as foreign key references.
- It builds a TopLink query object for each `ejbFind` or `ejbSelect` of an entity bean and migrates persistence metadata, such as customized query statements.

Table 3–1 lists OC4J `<entity-deployment>` attributes and subelements from the `orion-ejb-jar.xml` file and for each indicates whether or not the TopLink migration tool does the following:

- Retains it in the new `orion-ejb-jar.xml` file
- Migrates it to the new `toplink-ejb-jar.xml` file

In Table 3–1, elements are identified with angle brackets See Table A–1, " Attributes of the <entity-deployment> Element" on page A-4 for more information.

Table 3–1 OC4J orion-ejb-jar.xml Feature Migration

orion-ejb-jar.xml Feature	Retained in New orion-ejb-jar.xml	Migrated to New toplink-ejb-jar.xml
`<entity-deployment>`		
`clustering-schema`	✔	
`copy-by-value`	✔	
`data-source`	✔	
`location`	✔	
`max-instances`	✔	
`min-instances`	✔	
`max-tx-retries`	✔	
`disable-wrapper-cache`	✔	
`name`	✔	
`pool-cache-timeout`	✔	
`wrapper`	✔	
`local-wrapper`	✔	
`call-timeout`		✔

`exclusive-write-access`
`true`
`false`
		✔

`do-select-before-insert` `true`		

Table 3–1 (Cont.) OC4J orion-ejb-jar.xml Feature Migration

orion-ejb-jar.xml Feature	Retained in New orion-ejb-jar.xml	Migrated to New toplink-ejb-jar.xml
false		
isolation		✔
locking-mode		
pessimistic		✔
optimistic		
read-only		✔
old_pessimistic		
update-changed-fields-only		
true		✔
false		
table		✔
force-update		
true		
false		✔
data-synchronization-option		
ejbCreate		
ejbPostCreate		
batch-size		
Any value greater than 1		
<ior-security-config>	✔	
<env-entry-mapping>	✔	
<resource-ref-mapping>	✔	
<resource-env-ref-mapping>	✔	
<primkey-mapping>		✔
<cmp-field-mapping>		✔
one-to-one-join		
inner		✔
outer[1]		
shared		✔
<finder-method>		✔
<persistence-type>[2]		✔

[1] TopLink supports both `outer` and `inner` joins at run time.You can manually configure EJB descriptors with these options.

[2] The `persistence-type` attribute's table column size, if present, is discarded. For more information, see "Recovering persistence-type Table Column Size" on page 3-9.

Table 3–2 lists OC4J features and their TopLink equivalents configured by the TopLink migration tool.

Table 3–2 OC4J and TopLink Feature Comparison

Feature	orion-ejb-jar.xml	toplink-ejb-jar.xml
CMP field mapping	Direct Serialized object	Direct-to-field Serialized object
CMR field mapping	One-to-one One-to-many Many-to-many	One-to-one One-to-many Many-to-many
Partial query	Full SQL statement	SQL Call
Finder	Oracle-specific syntax	SQL Call or EJB-QL
Lazy loading (fetch group)	Lazy loading of primary key and CMP fields	Not supported. Alternatively, you can manually configure the TopLink equivalent, if appropriate.
SQL statement caching	Cache static SQL	Not supported at the bean level. TopLink supports parameterized SQL and statement caching at the session and query level.
Locking	Optimistic: database-level Pessimistic: bean instance-level	Optimistic: object-level Pessimistic: query lock at database-level
Read-only	Attempt to change throws `Exception`	Attempt to change throws `Exception`
Validity timeout	Read-only bean validity timeout before reloaded.	Cache timeout
Isolation level	Committed Serializable	Committed Serializable Not Committed Not Repeatable
Delay update until commit	Supported	Supported
Exclusive write access on bean	Default value is `false`	Assume `true`
Insert without existence check	Supported	Supported
Update changed fields only	Supported	Supported
Force update	Invoke bean life cycle `ejbStore` method even though persistent fields have not changed	Supported

Using the TopLink Migration Tool From the Command Line

To use the TopLink migration tool from the command line, you must perform the following steps:

1. Ensure that the following is in your classpath:
   - `<`*`TOPLINK_HOME`*`>/jlib/antlr.jar`

- <*TOPLINK_HOME*>/jlib/ejb.jar

> **Note:** Depending on your specific installation, the ejb.jar file could be located in <ORACLE_HOME>/j2ee/home/lib/ directory instead.

- <*TOPLINK_HOME*>/jlib/toplink.jar
- <*TOPLINK_HOME*>/jlib/cmpmigrator.jar
- <*TOPLINK_HOME*>/jlib/toplinkmw.jar
- <*TOPLINK_HOME*>/jlib/tlmwcore.jar
- <*TOPLINK_HOME*>/config
- <*ORACLE_HOME*>/lib/xmlparserv2.jar

2. If you intend to migrate from plain XML files (rather than an EAR or JAR file), ensure that the domain classes are accessible and included in your classpath.
3. Make a backup copy of your original XML files.
4. Execute the TopLink migration tool, as Example 3–1 illustrates, using the appropriate arguments listed in Table 3–3.

   The usage information for the TopLink migration tool is as follows:

   ```
   java -Dtoplink.ejbjar.schemavalidation=<true|false>
   -Dtoplink.migrationtool.generateWorkbenchProject=<true|false>
   -Dhttp.proxyHost=<proxyHost> -Dhttp.proxyPort=<proxyPort>
   oracle.toplink.tools.migration.TopLinkCMPMigrator -s<nativePM> -i<inputDir>
   -a<ear>|<jar> -x -o<outputDir> -v
   ```

   To identify the input files, you must specify one of -a or -x.

   For troubleshooting information, see "Troubleshooting Your Migration" on page 3-10.

Example 3–1 Using the TopLink Migration Tool from the Command Line

```
java -Dhttp.proxyHost=www-proxy.us.oracle.com -Dhttp.proxyPort=80
oracle.toplink.tools.migration.TopLinkCMPMigrator -sOc4j-native -iC:/mywork/in
-aEmployee.ear -oC:/mywork/out -v
```

Table 3–3 TopLink Migration Tool Arguments

Argument	Description
toplink.ejbjar.schemavalidation	The system property used to turn on schema validation if ejb-jar.xml uses XML Schema (XSD) instead of DTD. The default value is false.
toplink.migrationtool.generateWorkbenchProject	The system property used enable generation of the TopLink Workbench project. The default value is true.
<*proxyHost*>	The address of your local HTTP proxy host.
<*proxyHost*>	The port number on which your local HTTP proxy host receives HTTP requests.
-s <source>	The name of the native persistence manager from which you are migrating. For OC4J, use the name oc4j-native.

Table 3–3 (Cont.) TopLink Migration Tool Arguments

Argument	Description
`-i <input-directory>`	Fully qualified path to the input directory that contains both the OC4J `ejb-jar.xml` and `orion-ejb-jar.xml` files to migrate. Current working directory is the default.
`-a <EAR-or-JAR>`	Fully qualified path to the archive file (either an EAR or JAR) that contains both the OC4J `ejb-jar.xml` and `orion-ejb-jar.xml` files to migrate.
`-x`	Tells the TopLink migration tool that the OC4J files in the input directory to migrate from are plain XML files (not in an archive file). If you use this option, ensure that the domain classes are accessible and included in your classpath.
`-o <output-directory`	`<targetDir>` is the path to the directory into which the TopLink migration tool writes the new `orion-ejb-jar.xml`, `toplink-ejb-jar.xml`, and log files. The path may be absolute or relative to the current working directory. You must specify this argument value. Ensure that permissions are set on this directory to allow the TopLink migration tool to create files and subdirectories.
`-v`	Verbose mode. Tells the TopLink migration tool to log errors and diagnostic information to the console.

Post-Migration Changes

After you migrate the `orion-ejb-jar.xml` file persistence configuration to your `toplink-ejb-jar.xml` file, consider the following post-migration changes:

- Recovering persistence-type Table Column Size
- Updating the Unknown Primary Key Class Mapping Sequence Table
- Project Customization

Recovering persistence-type Table Column Size

In the `orion-ejb-jar.xml` file, you can specify this mapping, `cmp-field-mapping`, with a `persistence-type` attribute that provides both the type and column size as shown in Example 3–2.

Example 3–2 A cmp-field-mapping with persistence-type Specifying a Column Size

```
<cmp-field-mapping ejb-reference-home="MyOtherEntity" name="myField"
                   persistence-name="myField" persistence-type="VARCHAR2(30)">
```

The TopLink migration tool migrates the persistence type, but not the column size, because a TopLink project does not normally contain this size information.

To recover the `persistence-type` column size, do the following:

1. Perform the migration as described in "Using the TopLink Migration Tool From the Command Line" on page 3-7.
2. Launch the generated TopLink Workbench project file `TLCmpProject.mwp`.
3. Log in to your database.

Updating the Unknown Primary Key Class Mapping Sequence Table

TopLink supports the use of an unknown primary key class. Naturally, the TopLink migration tool also supports this feature.

OC4J uses a native run-time key generator to generate unique keys for `auto-id` key fields. In contrast, TopLink uses a sequencing table.

If your OC4J persistence configuration includes the use of an unknown primary key class, then the TopLink migration tool will create a sequencing table for this purpose.

Before deploying your application after migration, you must do the following:

1. Determine the largest key value generated prior to migration.
2. Manually update the counter of the TopLink migration tool-generated sequence table to a number that must be one larger than the largest key value generated prior to migration.

Project Customization

You can customize the following components of your project:

- Persistence Manager Property
- Session Event Listener

Persistence Manager Property After migrating your application, you may wish to customize the persistence manager properties in the `orion-ejb-jar.xml` file. These properties are used to configure the TopLink session that the TopLink runtime uses internally for CMP projects.

Session Event Listener After you applied the default settings to your project at deployment time, you may wish to customize the TopLink session by configuring the session event listener. The prelogin event that the session raises is particularly useful. It lets you define the custom (nondefault) specifics for the session just before the session initializes and acquires connections.

Troubleshooting Your Migration

This section describes solutions for problems you may encounter during migration, including the following:

- Log Messages
- Unexpected Relational Multiplicity

Log Messages

As it operates, the TopLink migration tool logs all errors and diagnostic output to a log file named `oc4j_migration.log` in the output directory.

In addition to these warnings, the TopLink migration tool logs an error if it encounters a problem that prevents it from completing the migration. Table 3–4 lists these problems and suggests possible solutions.

Table 3–4 TopLink Migration Tool Error Messages

Error Message	Description
There is no `ejb-jar.xml` in the input file. You must provide the `ejb-jar.xml` in order for the migration process to work.	The `ejb-jar.xml` file is missing. The TopLink migration tool stops and copies the original input files into the target directory. Verify that the `ejb-jar.xml` file is present in the specified EAR, JAR, or as a plain XML file. Empty the target directory and execute the TopLink migration tool again.

Table 3–4 (Cont.) TopLink Migration Tool Error Messages

Error Message	Description
There is no `orion-ejb-jar.xml` with native persistent metadata defined, no migration needed.	The `orion-ejb-jar.xml` file is missing. The TopLink migration tool stops and copies the original input files into the target directory. Verify that the `orion-ejb-jar.xml` file is present in the specified EAR, JAR, or as a plain XML file. Empty the target directory and execute the TopLink migration tool again.
`toplink-ejb-jar.xml` is already defined in the archive, no migration needed.	A `toplink-ejb-jar.xml` file is already present in the target directory. The TopLink migration tool stops and copies the original input files into the target directory. Remove the `toplink-ejb-jar.xml` file from the target directory. Empty the target directory and execute the TopLink migration tool again.

Unexpected Relational Multiplicity

The TopLink migration tool retrieves relationship multiplicity from the `orion-ejb-jar.xml` file and not from the OC4J `ejb-jar.xml` file.

Thus, even though the OC4J `ejb-jar.xml` file defines a relationship to be one-to-many, if the `orion-ejb-jar.xml` file defines the same relationship as many-to-many, then the TopLink migration tool will migrate the relationship as many-to-many.

4

Implementing an EJB 2.0 Entity Bean With Container-Managed Persistence

This chapter describes the ways to implement EJB 2.0 entity beans with container-managed persistence.

This chapter includes information on the following topics:

- Implementing an EJB 2.0 Entity Bean With Container-Managed Persistence

For more information, see:

- "What is an Entity Bean?" on page 1-1
- "Configuring an EJB 2.0 Entity Bean With Container-Managed Persistence" on page 5-1

Implementing an EJB 2.0 Entity Bean With Container-Managed Persistence

The process of implementing an EJB 2.0 entity bean with container-managed persistence consists of the following steps:

1. Create the bean's home interface. The home interface defines the methods that allow a client to create, find, or remove an entity bean. See "Implementing the Entity Bean Home Interface" on page 4-2 for more information.
2. Create the component (remote) interfaces for the bean. The component interfaces declare the methods that a client can invoke. See "Implementing the Entity Bean Component Interface" on page 4-3 for more information.
3. Define the primary key for the bean. The primary key identifies each entity bean instance and is a serializable class. See "Configuring Primary Key" on page 5-1 for more information.
4. Implement the bean. See "Implementing the Entity Bean Class" on page 4-4 for more information.
5. Create the bean deployment descriptor–a file that specifies properties for the bean using XML elements. It is your responsibility to identify the data within the bean that the container will manage (see "Configuring Container-Managed Persistent Fields" on page 5-10 for more information on persistence fields). If these fields describe relationships to other objects, see "Configuring Container-Managed Relationship Fields" on page 5-14.

   Any EJB container services that you might want to configure are also designated in the deployment descriptor. For information about data sources and JTA, see *Oracle*

Containers for J2EE Services Guide. For information about security, see *Oracle Containers for J2EE Security Guide*.

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 data source information.

6. 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 your entity bean to OC4J. See Chapter 2, "Understanding Orion CMP Application Development" on page 2-1 for more information.

For information on how to configure EJB 2.0 entity beans with container-managed persistence, see "Configuring an EJB 2.0 Entity Bean With Container-Managed Persistence" on page 5-1.

Implementing the Entity Bean Home Interface

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

- The remote home interface, which extends `javax.ejb.EJBHome`. This type of the home interface is provided by beans that provide a remote client view.
- The local home interface, which extends `javax.ejb.EJBLocalHome`. This type of the home interface is provided by beans that provide a local client view.

> **Note:** If an entity bean is the target of a container-managed relationship, then it must have local interfaces.

A client can locate the bean's home interface through the standard JNDI API.

The home interface must contain a `create` method, which the client invokes to create the bean instance. The entity bean can have zero or more `create` methods, each with its own defined parameters.

Entity beans must define one or more finder methods, where at least one is a `findByPrimaryKey` method. Optionally, you can define other finder methods (named `find<NAME>`) for the bean.

In addition to creation and retrieval methods, you can provide business methods within the home interface. These methods cannot access data of a particular entity object. 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.

Example 4–1 Implementing the Entity Bean Home Interface

The home interface must extend `javax.ejb.EJBHome` interface, as well as define the `create` and `findByPrimaryKey` methods.

Example 4–1 demonstrates an implementation of a local home interface that provides a method to create the remote interface. It also provides two finder methods: one to find a specific employee by an employee number, and one that finds all employees. The

`calculateSalary` 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.

```
package employee;

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

public interface EmployeeLocalHome extends EJBLocalHome {

    public EmployeeLocal create(Integer empNumber) throws CreateException;

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

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

    // Calculate the salaries of all employees
    public float calculateSalary() throws Exception;
}
```

It is the responsibility of the EJB container to create an implementation for this interface.

Declaring the Home Interface in the Deployment Descriptor

The following is the declaration of the home interface in the deployment descriptor:

```
<local-home>employee.EmployeeLocalHome</local-home>
```

Implementing the Entity Bean Component Interface

An EJB object is accessible through the bean's component interface. The component interface (also often referred to as remote interface) defines the business methods that a client may call. The business methods are implemented in the entity bean code. The following are the types of the component interface:

- The component interface that extends `javax.ejb.EJBObject`. The EJBObject interface defines the operations that let the client access the EJB object's identity and create a persistent handle for this object.
- The component interface that extends `javax.ejb.EJBLocalObject`. The EJBLocalObject interface defines the operations that let the client to access this object's identity.

Example 4–2 Implementing the Entity Bean Component Interface

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 getEmpNumber();
    public void setEmpNumber(Integer empNumber);
```

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

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

}
```

It is the responsibility of the EJB container to create an implementation for the component interface.

Declaring the Component Interface in the Deployment Descriptor

The following is the declaration of the component interface in the deployment descriptor:

```
<local>employee.EmployeeLocal</local>
```

Implementing the Entity Bean Class

An entity bean class must meet the following criteria:

- The class must implement, directly or indirectly, the `javax.ejb.EntityBean` interface.
- The class must be defined as `public` and must be `abstract`.
- The class must define a public constructor that takes no arguments.
- The class must not define the `finalize()` method.

The class may, but is not required to, implement the entity bean's component interface (see "Implementing the Entity Bean Component Interface" on page 4-3).

The entity bean class implements the following methods:

- The target methods for the methods that are declared in the home interface (see "Implementing the Entity Bean Home Interface" on page 4-2), 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.
  - Any home interface business methods, which have an `ejbHome` prefix in the bean implementation. For example, the `calculateSalary` method is implemented in the `ejbHomeCalculateSalary` method.
- The business logic methods that are declared in the component interface.
- The methods that are inherited from the `javax.ejb.EntityBean` interface (such as `ejbActivate`, `ejbPassivate`, and so forth).

It is the responsibility of the container to manage most of the target methods and the data objects. For information about the entity bean's callback methods, see "Callback Methods" on page 1-4.

Example 4–3 Implementing the Entity Bean Class

Example 4–3 demonstrates the implementation of the entity bean class.

```
package employee;

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

public abstract class EmployeeBean implements EntityBean {

  private EntityContext ctx;

  // Each persistent field has a getter and a setter
  public abstract Integer getEmpNumber();
  public abstract void setEmpNumber(Integer empNumber);

  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 ejbHomeCalculateSalary() 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();
      }
    catch (FinderException e) {
      System.out.println("Got finder Exception " + e.getMessage());
      throw new Exception(e.getMessage());
    }
  }

  public EmployeePK ejbCreate(Integer empNumber, String empName, Float salary)
  throws CreateException {
    setEmpNumber(empNumber);
    setEmpName(empName);
    setSalary(salary);
    return new EmployeePK(empNumber);
  }

  public void ejbPostCreate(Integer empNumber, 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;
  }

}
```

Defining the Entity Bean Class in the Deployment Descriptor

You define the entity bean class in the deployment descriptor with the following line:

```
<ejb-class>employee.EmployeeBean</ejb-class>
```

The following is the sample deployment descriptor for the entity bean:

```
<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>empNumber</field-name></cmp-field>
        <cmp-field><field-name>empName</field-name></cmp-field>
        <cmp-field><field-name>salary</field-name></cmp-field>
        <primkey-field>empNumber</primkey-field>
    </entity>
...
</enterprise-beans>
```

5

Configuring an EJB 2.0 Entity Bean With Container-Managed Persistence

This chapter describes the various options that you must configure in order to use an EJB 2.0 entity bean with container-managed persistence.

Table 5–1 lists these options and indicates which are basic (applicable to most applications) and which are advanced (applicable to more specialized applications).

Table 5–1 Configurable Options for an EJB 2.0 Entity Bean With Container-Managed Persistence

Options	Type
"Configuring Primary Key"	Basic
"Configuring Container-Managed Persistent Fields"	Basic
Configuring Container-Managed Relationship Fields	Basic
Configuring Database Isolation Levels	Advanced
"Configuring Concurrency Modes"	Advanced
"Configuring Exclusive Write Access to the Database"	Advanced
"Configuring Callback Methods for EJB 2.0 Entity Beans With Container-Managed Persistence"	Advanced

> **Note:** Oracle suggests that you use JDeveloper IDE for configuring your entity beans with container-managed persistence. The reason is that JDeveloper is capable of managing complex mappings between the entity beans and the database tables. See "Configuring Container-Managed Relationship Fields" on page 5-14 and "Using JDeveloper" on page 2-4 for more information.

For more information, see the following:

- "What is an Entity Bean?" on page 1-1
- "Implementing an EJB 2.0 Entity Bean With Container-Managed Persistence" on page 4-1

Configuring 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.

This section describes the following aspects of the primary key configurations:

- Configuring Primary Key Field
- Configuring Primary Key Class
- Configuring Automatic Primary Key Generation

Configuring Primary Key Field

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 the following ways:

- Define the type of the primary key to be a well-known type. The type is defined in the `<prim-key-class>` element 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`.

  The advanced option of defining the primary key is to define its type as a serializable object within a serializable `<NAME>PK` class. This class is declared in the `<prim-key-class>` element in the deployment descriptor. See "Configuring Primary Key Class" on page 5-3 for more information.

- Specify an automatically 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 automatically generated by the container. See "Configuring Automatic Primary Key Generation" on page 5-9 for more information.

Example 5–1 Defining a Primary Key of a Well-Known Type Within the Deployment Descriptor

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. Example 5–1 shows how to define the primary key (employee number) as a `java.lang.Integer`:

```
<enterprise-beans>
    <entity>
        <display-name>Employee</display-name>
        <ejb-name>EmployeeBean</ejb-name>
        <local-home>employee.EmployeeLocalHome</local-home>
        <local>employee.EmployeeLocal</local>
        <ejb-class>employee.EmployeeBean</ejb-class>
        <persistence-type>Container</persistence-type>
        <prim-key-class>java.lang.Integer</prim-key-class>
        <reentrant>False</reentrant>
        <cmp-version>2.x</cmp-version>
        <abstract-schema-name>Employee</abstract-schema-name>
        <cmp-field><field-name>empNumber</field-name></cmp-field>
        <cmp-field><field-name>empName</field-name></cmp-field>
        <cmp-field><field-name>salary</field-name></cmp-field>
        <primkey-field>empNumber</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.

Within the orion-ejb-jar.xml file, the primary key is mapped to the underlying database persistence storage by mapping the container-managed persistent field or primary key field defined in the ejb-jar.xml file to the database column name. In the following orion-ejb-jar.xml fragment, the EmpBean persistence storage is defined as the EMP table in the database that is defined in the jdbc/OracleDS data source. Following the <entity-deployment> element definition (see Table A–1, " Attributes of the <entity-deployment> Element" on page A-4), the primary key, empNumber, is mapped to the EMPNUMBER column in the EMP table, and the empName and salary persistent fields are mapped to EMPNAME and SALARY columns respectively in the EMP table:

```
<entity-deployment name="EmpBean" ...table="EMP" data-source="jdbc/OracleDS"...>
    <primkey-mapping>
        <cmp-field-mapping name="empNumber" persistence-name="EMPNUMBER" />
    </primkey-mapping>
    <cmp-field-mapping name="empName" persistence-name="EMPNAME" />
    <cmp-field-mapping name="salary" persistence-name="SALARY" />
...
```

Configuring Primary Key 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.

Example 5–2 Defining the Primary Key Class

Example 5–2 places the employee number within a primary key class.

```
package employee;

public class EmployeePK implements java.io.Serializable {

    public Integer empNumber;

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

    public EmployeePK(Integer newEmpNumber) {
        this.empNumber = newEmpNumber;
    }

}
```

Example 5–3 Declaring the Primary Key Class in the Deployment Descriptor

The primary key class is declared in the deployment descriptor within the `<prim-key-class>` element, and each of its variables are declared within a `<cmp-field><field-name>` element, as Example 5–3 shows:

```
<enterprise-beans>
    <entity>
        <description>no description</description>
        <display-name>EmployeeBean</display-name>
        <ejb-name>EmployeeBean</ejb-name>
        <local-home>employee.EmployeeLocalHome</local-home>
        <local>employee.EmployeeLocal</local>
        <ejb-class>employee.EmployeeBean</ejb-class>
        <persistence-type>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>empNumber</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 you define the primary key, the container creates a column in the entity bean table for the primary key and maps the primary key class declared in the deployment descriptor to this column.

The persistent fields are mapped in the `orion-ejb-jar.xml` in the same manner as described in the "Configuring Primary Key" section. With a complex primary key, the mapping contains more than a single field; thus, the `<cmp-field-mapping>` element of the `<primkey-mapping>` element contains another `<fields>` subelement. Every field of the primary key is defined in a separate `<cmp-field-mapping>` element within the `<fields>` element, similar to the following:

```
<primkey-mapping>
    <cmp-field-mapping>
        <fields>
            <cmp-field-mapping name="empNumber" persistence-name="EMPNUMBER" />
        </fields>
    </cmp-field-mapping>
</primkey-mapping>
```

> **Note:** If you have a complex primary key that contains a foreign key, you need to apply a special mapping. See "Configuring Foreign Key in a Composite Primary Key" on page 5-4 for more information.

Configuring Foreign Key in a Composite Primary Key

In the EJB 2.0 specification, the primary key for an entity bean must be initialized within the `ejbCreate` method. However, in this method you cannot set any relationship that this bean has to another bean. The earliest when you can set this relationship in a foreign key is in the `ejbPostCreate` method.

That said, if you have a foreign key within a composite primary key, you face the following problem: you must set all fields within the composite primary key in the `ejbCreate` method, but you cannot set the foreign key in this method.

The following hypothetical scenario models the way around this problem: an order can contain one or more items; the order bean has many items in it; each item belongs to an order. The primary key for the item is a composite primary key consisting of the item identifier and the order identifier. The order identifier is a foreign key that points to the order.

You would have to modify the deployment descriptors and bean implementation to add a placeholder persistent field that mimics the actual foreign key field. This field is set during the `ejbCreate` method. However, both the placeholder persistent field and the foreign key point to the same database column. The actual foreign key is updated during the `ejbPostCreate` method.

> **Note:** Modify the `ejb-jar.xml` file with the placeholder persistent field and the foreign key. Deploy the application with `<autocreate-tables>` element in the `orion-application.xml` file set to `false` to automatically generate the `orion-ejb-jar.xml` file, without creating any tables. Then modify the `orion-ejb-jar.xml` file to point to the correct database columns, set `<autocreate-tables>` element to `true`, and redeploy.

Example 5–4 Modifying the Deployment Descriptor and the Bean Code to Accommodate a Foreign Key Within a Primary Key

Example 5–4 demonstrates how to modify both deployment descriptors and the bean implementation.

In the order scenario, each order contains one or more items. The `OrderBean` represents the order, and the `OrderItemBean` represents the items in the order. Each item has a primary key that consists of the item number and the order number to which it belongs. Thus, the primary key for the item contains a foreign key that points to an order bean.

To adjust for a composite primary key, modify the `ejb-jar.xml` file in the following way:

1. Define a persistent field in the primary key as a placeholder for the foreign key. This placeholder should be used in the composite primary key class definition.

   In the Example 5–4, an `orderId` persistent field is defined in a `<cmp-field>` element. The `orderId` and `itemId` persistent fields are used to identify the composite primary key in the `OrderItemPK.java`.

2. Define the foreign key outside the primary key definition in its own `<cmr-field>` element in the `<relationships>` section.

   In the Example 5–4, the `belongToOrder` foreign key is defined in a `<cmr-field>` element for the `OrderItemBean`, defining the relationship from the item to the order.

```
<entity>
    <ejb-name>OrderItemBean</ejb-name>
    <local-home>OrderItemLocalHome</local-home>
    <local>OrderItemLocal</local>
    <ejb-class>OrderItemBean</ejb-class>
    ...
    <cmp-field><field-name>itemId</field-name></cmp-field>
    <cmp-field><field-name>orderId</field-name></cmp-field>
    <cmp-field><field-name>price</field-name></cmp-field>
    <prim-key-class>OrderItemPK</prim-key-class>
    ...
```

```
</entity>
<relationships>
    <ejb-relation>
        <ejb-relation-name>Order-OrderItem</ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Order-Has-OrderItems
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>OrderBean</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>items</cmr-field-name>
                <cmr-field-type>java.util.Collection</cmr-field-type>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                OrderItems-from-Order
            </ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <cascade-delete/>
            <relationship-role-source>
                <ejb-name>OrderItemBean</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>belongToOrder</cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
...
```

The `OrderItemPK.java` class defines the contents of the complex primary key, as follows:

```
public class OrderItemPK implements java.io.Serializable {

    public Integer itemID;
    public Integer orderID;

    public OrderItemPK() {
        this.itemId = null;
        this.orderId = null;
    }

    public OrderItemPK(Integer newItemId, Integer newOrderId) {
        this.itemId = newItemId;
        this.orderId = newOrderId;
    }

    public boolean equals(Object o) {
        if (o instanceof OrderItemPK) {
            OrderItemPK pk = (OrderItemPK) o;
            if (pk.itemId.intValue() == itemId.intValue() &&
                pk.orderId.intValue() == orderId.intValue()) {
                return true;
            }
        }
        return false;
    }
```

```
    public int hashCode() {
        return itemId.hashCode() * orderId.hashCode();
    }
}
```

If you consider the automatically created database tables sufficient, you do not need to modify the `orion-ejb-jar.xml` file. However, if you need to map to existing database tables, then you modify the `orion-ejb-jar.xml` file to point to these tables (see "Configuring Explicit Mapping of Relationship Fields to the Database" on page 5-18 for more information).

After the automatic generation of the `orion-ejb-jar.xml` file, copy it into your development directory. The database column names are defined in the `persistence-name` attributes in each of the persistent and relationship field name mappings. Ensure that the `persistence-name` attributes for both the placeholder persistent field and foreign key are the same.

The following is the `orion-ejb-jar.xml` file for the Order/OrderItem example:

```
<entity-deployment name="OrderItemBean"  table="ORDER_ITEM">
    <primkey-mapping>
        <cmp-field-mapping name="itemId" persistence-name="Item_ID" />
        <cmp-field-mapping name="orderId" persistence-name="Order_ID" />
    </primkey-mapping>
    <cmp-field-mapping name="price" persistence-name="Price" />
    <cmp-field-mapping name="belongToOrder">
        <entity-ref home="OrderBean">
            <cmp-field-mapping name="belongToOrder" persistence-name="Order_ID" />
        </entity-ref>
    </cmp-field-mapping>
</entity-deployment>

<entity-deployment name="OrderBean"  table="ORDER">
    <primkey-mapping>
        <cmp-field-mapping name="orderId" persistence-name="Order_ID" />
    </primkey-mapping>
    <cmp-field-mapping name="orderDesc" persistence-name="Order_Description" />
    <cmp-field-mapping name="items">
        <collection-mapping table="ORDER_ITEM">
            <primkey-mapping>
                <cmp-field-mapping name="OrderBean_orderId">
                    <entity-ref home="OrderBean">
                        <cmp-field-mapping name="OrderBean_orderId">
                    </entity-ref>
                </cmp-field-mapping>
            </primkey-mapping>
            <value-mapping type="OrderItemLocal">
                <cmp-field-mapping name="OrderItemBean_itemId">
                    <entity-ref home="OrderItemBean">
                        <cmp-field-mapping name="OrderItemBean_itemId">
                                <cmp-field-mapping name="OrderItemBean_itemId">
                            </fields>
                        </cmp-field-mapping>
                    </entity-ref>
                </cmp-field-mapping>
            </value-mapping>
            </collection-mapping>
        </cmp-field-mapping>
</entity-deployment>
```

The following takes place in the <entity-deployment> (see Table A–1, " Attributes of the <entity-deployment> Element" on page A-4) section for the OrderItemBean of the orion-ejb-jar.xml file for the Order/OrderItem example:

- The table is defined in the table attribute, which is ORDER_ITEM in this example.
- The column name for the itemId is defined in the persistence-name attribute as Item_ID.
- The column name for the placeholder persistent field, orderId, is defined in the persistence-name attribute as Order_ID.
- The foreign key, belongToOrder, is mapped to the database column, Order_ID, which is the same column as the placeholder persistent field, orderId.

Both the foreign key (belongToOrder), and the placeholder persistent field (orderId) must point to the same database column.

Finally, you must update the bean implementation to work with both the placeholder persistent field and the foreign key, as follows:

1. In the ejbCreate method, do the following:
   - Create the placeholder persistent field that takes the place of the foreign key field.
   - Set a value in the placeholder persistent field in the ejbCreate method. This value is written out to the foreign key field in the database table.
2. In the ejbPostCreate method, set the foreign key to the value in the duplicate persistent field.

> **Note:** Since the foreign key is a part of a primary key, you can only set it once.

In the Order/OrderItem example, the orderId persistent field is set in the ejbCreate method, whereas the belongToOrder relationship field is set in the ejbPostCreate method, as follows:

```
public OrderItemPK ejbCreate(OrderItem orderItem) throws CreateException {
    setItemId(orderItem.getItemId());
    setOrderId(orderItem.getOrderId());
    setPrice(orderItem.getPrice());
    return new OrderItemPK(orderItem.getItemId(),orderItem.getOrderId());
}

public void ejbPostCreate(OrderItem orderItem) throws CreateException {
    // right after the bean has been created
    try {
        Context ctx = new InitialContext();
        OrderLocalHome orderHome =
                    (OrderLocalHome)ctx.lookup("java:comp/env/OrderBean");
        OrderLocal order = orderHome.findByPrimaryKey(orderItem.getOrderId());
        setBelongToOrder(order);
    }
    catch(Exception e) {
        e.printStackTrace();
        throw new EJBException(e);
    }
}
```

The following is the code for the `OrderItem` object that is passed into the `ejbCreate` and `ejbPostCreate` methods:

```
public class OrderItem implements java.io.Serializable {

    private Integer itemId;
    private Integer orderId;
    private Double price;

    public OrderItem(Integer itemId, Integer orderId, Double price) {
        this.itemId = itemId;
        this.orderId = orderId;
        this.price = price;
    }

    public Integer getItemId() {
        return itemId;
    }

    public void setItemId(Integer itemId) {
        this.itemId = itemId;
    }

    public Integer getOrderId() {
        return orderId;
    }

    public void setOrderId(Integer orderId) {
        this.orderId = orderId;
    }

    public Double getPrice() {
        return price;
    }

    public void setPrice(Double price) {
        this.price = price;
    }

    public boolean equals(Object other) {
        if (other instanceof OrderItem) {
            OrderItem orderItem = (OrderItem)other;
            if (itemId.equals(orderItem.getItemId()) &&
                orderId.equals(orderItem.getOrderId()) &&
                price.equals(orderItem.getPrice()) ) {
                return true;
            }
        }
        return false;
    }
}
```

Configuring Automatic Primary Key Generation

If you specify a `java.lang.Object` as the primary key class type in `<prim-key-class>` element of your deployment descriptor, but do not specify the primary key name in `<primkey-field>` element, then the primary key is automatically generated by the container, 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.Object</prim-key-class>
        <reentrant>False</reentrant>
        <cmp-version>2.x</cmp-version>
        <abstract-schema-name>Employee</abstract-schema-name>
        <cmp-field><field-name>empNumber</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. This is generated in the `orion-ejb-jar.xml` for the bean, as follows:

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

Configuring Container-Managed Persistent Fields

Container-managed persistent fields represent simple data types that are persisted to database tables. These fields define the state of an entity bean. These fields are direct attributes of a bean. For more information about persistent fields, see "Container-Managed Persistent Fields" on page 1-2.

In entity beans with container-managed persistence, you define the persistent data both in the bean instance and in the deployment descriptor using the following:

- Accessor methods (getter and setter) in the bean instance: For each persistent field, both a getter and a setter method is created. The data type of the parameter returned from the getter and passed into the setter defines the simple data type of the field. The name of the field is designated by the name of the getter and setter methods.

  The following code shows a getter and a setter for the employee name persistent field. A String that is returned from the getter and passed into the setter is a 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 persistent field name. In the following example, `empName` is the persistent field name:

  ```
  public abstract String getEmpName() throws RemoteException;

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

- The deployment descriptor declares these fields as persistent. Each field name must be defined in a `<cmp-field><field-name>` element in the deployment descriptor. In the preceding example, three persistent fields are defined in the data accessor methods: `empNumber`, `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>empNumber</field-name></cmp-field>
        <cmp-field><field-name>empName</field-name></cmp-field>
        <cmp-field><field-name>salary</field-name></cmp-field>
        <primkey-field>empNumber</primkey-field>
    </entity>
...
</enterprise-beans>
```

Map these fields to the database as follows:

- Accept the defaults for these fields and avoid more deployment descriptor configuration. See "Configuring Default Mapping of Persistent Fields to the Database" on page 5-11 for more information.
- 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 "Configuring Explicit Mapping of Persistent Fields to the Database" on page 5-12 for more information.

Configuring 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 nonemulated data sources.

  Upon installation, the default database is a locally installed Oracle database that must be listening on port 1521 with a SID of `ORCL`. To customize the default database, change the first configured database to point to your database.

- Table: The container automatically 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 name is separated by an underscore character (_):

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

  – JAR file name, including the `.jar` extension. However, all dash characters (-) and periods (.) are converted to underscore characters (_) 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 that 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 are 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`.

Configuring Explicit Mapping of Persistent Fields to the Database

As "Configuring Default Mapping of Persistent Fields to the Database" 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.

To explicitly map persistent fields to the database, 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 than to create these fields.

2. Modify the `<entity-deployment>` element (see Table A–1, " Attributes of the <entity-deployment> Element" on page A-4) in the `orion-ejb-jar.xml` file to use the database table and columns you specify.

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

The explicit mapping of persistent fields is completed within an `<entity-deployment>` element t (see Table A–1, " Attributes of the <entity-deployment> Element" on page A-4). This element contains all mapping for an entity bean. The following are the attributes and elements that are specific to persistent field mapping:

```
<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>
```

Table 5–2 Deployment Descriptor's Elements for Mapping Persistent Fields to a Specific Database Table

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 container-managed persistent fields that you will map within it.
2. Configure a `<cmp-field-mapping>` element for every field within the bean that you are mapping. Each `<cmp-field-mapping>` element must contain the name of the field to be persisted, as follows:
   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 5–5 Mapping Persistent Fields to a Specific Database Table

Example 5–5 demonstrates how to map persistent fields in your bean instance to database tables and columns by mapping the employee persistence fields to the Oracle database table `EMP`:

```
<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="empNumber" persistence-name="empnumber" />
    </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, as Table 5–3 shows:

Table 5–3 Mapping of the Deployment Descriptor's Element Values by the Container

Bean	Database
`emp/EmpBean`	`EMP` table, located in `jdbc/OracleDS` in the `data-sources.xml` file
`empNumber`	`EMPNUMBER` column as primary key

Table 5–3 (Cont.) Mapping of the Deployment Descriptor's Element Values by the

Bean	Database
empName	EMPNAME column
salary	SALARY column

Configuring Container-Managed Relationship Fields

A relationship field identifies a related bean. A relationship field behaves similar to a foreign key in a database table. For more information, see "Relationship Fields" on page 1-4.

Relationship fields are virtual. You provide getters and setters for these fields in the code of your bean class, similar to the following:

```
public abstract void setContractInfo(ContractInfo contractInfo) throws
RemoteException

public abstract ContractInfo getContractInfo() throws RemoteException
```

The container provides the implementation of these methods.

In the preceding example, the corresponding relationship field is `contractInfo`.

In a deployment descriptor, this field is defined in a `<cmr-field><cmr-field-name>` element, as follows:

```
<ejb-ralation>
    ...
    <cmr-field>
        <cmr-field-name>contractInfo</cmr-field-name>
        <cmr-field-type>contractInfo</cmr-field-type>
    </cmr-field>
    ...
</ejb-ralation>
```

For more information about container-managed relationships, see "Container-Managed Relationships" on page 1-3.

As each entity bean maps to a table in a database, each of its persistent and relationship fields are saved within a database table in columns. You can map these fields to the database by performing one of the following:

- Accept the defaults for these fields. Do not configure the deployment descriptor. The tables are automatically created for the bean based on the information in the `ejb-jar.xml` file. See "Configuring Default Mapping of Relationship Fields to the Database" on page 5-14 for more information.
- Map these 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 "Configuring Explicit Mapping of Relationship Fields to the Database" on page 5-18 for more information.

Configuring Default Mapping of Relationship Fields to the Database

When you declare relationship fields in the `ejb-jar.xml` file, OC4J provides default mapping of these fields to the database at the time of the automatic generation of the `orion-ejb-jar.xml` file. The default mapping for relationship fields is the same as for the persistent fields (see "Configuring Default Mapping of Persistent Fields to the Database" on page 5-11).

Note: For all future redeployments, copy the automatically generated `orion-ejb-jar.xml` file with the indicated table name into the same directory as your `ejb-jar.xml` file from the `J2EE_Home/application-deployments` directory. If you fail to do so, different table names may be generated at each redeployment.

In summary, the default mappings include the following:

- 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 underlying database table. The name of the entity bean's underlying table is supposed to be unique, so it is constructed with the following names, where each entry is separated by an underscore character (_):
  - EJB name defined in a `<ejb-name>` element in the deployment descriptor;
  - JAR file name, including the `.jar` extension. However, all dash characters (-) and periods (.) are converted to underscore characters (_) to follow SQL conventions. For example, if the name of your JAR file is `address.jar`, then `address_jar` is appended to the name;
  - Application name as defined by you during the deployment.

  If the constructed name is greater than 30 (thirty) characters, the name is truncated at 24 (twenty-four) characters. An underscore character (_), and then 5 (five) characters made up of an alpha-numeric hash code are appended to the name for uniqueness.

  For example, if the EJB name is `AddressEntry`, the JAR file name is `addr.jar` and the application name is `address`, then the default table name would be `AddressEntry_addr_jar_address`.
- Column names in each table–The container generates columns in each table based on the `<cmp-field>` and `<cmr-field>` elements defined in the deployment descriptor. A column is created for each `<cmp-field>` element that relates to the entity bean data. In addition, a column is created for each `<cmr-field>` element that represents a relationship. In a unidirectional relationship (see "Direction in CMR" on page 1-3), only a single entity in the relationship defines a `<cmr-field>` in the deployment descriptor. In bidirectional relationship (see "Direction in CMR" on page 1-3), 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 the default one-to-one relationship, the foreign key is created in the database table for the source entity bean, 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.
  - The default for one-to-many relationship uses a foreign key.
  - The default for many-to-many relationships creates an association table (a third table). This association table contains two foreign keys, where each key points to the primary key of one of the entity tables.

  Since the `<cmp-field>` and `<cmr-field>` elements represent Java data types, they may not convert to database types in the manner you believe they should.

There is a set of rules for converting CMP types to database types that you must follow (see "Conversion of CMP Types to Database Types" on page 5-16). Note that you can modify the translation rules of converting Java data types to database data types in special database XML files that are located in `J2EE_HOME/config/database-schemas` directory. This directory includes all database files. The Oracle Database conversion file is named `oracle.xml`.

- Primary keys–Entity beans' underlying tables contain primary keys (see "Configuring Primary Key" on page 5-1). The following are the types of primary keys:
  - Defined primary key: The primary key is generated as designated in the `<primkey-field>` element as a simple data type or class. Thus, the column name is the same as the name in the `<primkey-field>` element.
  - Composite primary key: The primary key is defined within a class, and is composed of several fields. Each field within the composite primary key is represented by a column in the database table, where each field is considered part of the primary key in the table.
  - Automatically generated primary key: If you specify a `java.lang.Object` as the primary key class type in `<prim-key-class>` element, but do not specify the primary key name in `<primkey-field>` element, then the primary key is automatically generated the the container. The column is named `AUTOID`.

Conversion of CMP Types to Database 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 classes that are serializable.

This section contains information on the following:

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

Simple Data Types Table 5–4 provides a list of the supported simple data types, which you can provide in the `persistence-type` attribute, with the mapping of these types to SQL types and to Oracle database types. Note that none of these mappings are guaranteed to work on non-Oracle databases.

Table 5–4 Simple Data Types

Known Type (native)	SQL Type	Oracle Type
java.lang.String	VARCHAR(255)	VARCHAR(255)
java.lang.Integer[]	INTEGER	NUMBER(20,0)
java.lang.Long[]	INTEGER	NUMBER(20,0)
java.lang.Short[]	INTEGER	NUMBER(10,0)
java.lang.Double[]	DOUBLE PRECISION	NUMBER(30,0)
java.lang.Float[]	FLOAT	NUMBER(20,5)
java.lang.Byte[]	SMALLINT	NUMBER(10,0)
java.lang.Character[]	CHAR	CHAR(1

Table 5–4 (Cont.) Simple Data Types

Known Type (native)	SQL Type	Oracle Type
java.lang.Boolean[]	BIT	NUMBER(1,0)
java.util.Date	DATETIME	DATE
java.sql.Date	DATE	DATE
java.uti.Time	DATE	DATE
java.sql.Timestamp	TIMESTAMP	TIMESTAMP
java.lang.String	CLOB	CLOB
char[]	CLOB	CLOB
byte[]	BLOB	BLOB
java.io.Serializable (4KB limit)	LONGVARBINARY	BLOB

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

The `Date` and `Time` map to `DATE` in the database, because the `DATE` contains the time. The `Timestamp`, however, maps to `TIMESTAMP` in the database, which gives the time in nanoseconds.

Mapping `java.sql.CLOB` and `java.sql.BLOB` directly is not currently supported because these objects are not serializable. However you can map a `String` or `char[]` and `byte[]` to database column type `CLOB` and `BLOB` respectively. Mapping a `char[]` to a `CLOB` or a `byte[]` to a `BLOB` can only be done with an Oracle database. The Oracle JDBC API was modified to handle this operation.

There is a 4 KB limit when mapping a serialized object to a `BLOB` type over the JDBC Thin driver.

When `String` and `char[]` variables map to a `VARCHAR2` in the database, it can only hold up to 2K. However, you can map `String` object or `char[]` larger than 2K to a `CLOB` by doing the following:

1. The bean implementation uses the `String` or `char[]` objects.
2. The `persistence-type` attribute of the `<cmp-field-mapping>` element defines the object as a `CLOB`, as follows:

```
<cmp-field-mapping name="stringdata" persistence-name="stringdata"
                                     persistence-type="CLOB" />
```

Similarly, you can map a `byte[]` in the bean implementation to a `BLOB`, as follows:

```
<cmp-field-mapping name="bytedata" persistence-name="bytedata"
                                   persistence-type="BLOB" />
```

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

Other Entity Beans or Collections You should not define other entity beans or `Collection` objects as a CMP type–these are relationships and should be defined within a relationship field, as follows:

- A relationship to another entity bean is always defined in a `<cmr-field>` relationship.
- `Collection` objects promote a "many" side of a relationship and should be configured within a `<cmr-field>` relationship.

> **Note:** You should not use subinterfaces of `Collection` (such as `List`, for example). Use the `Collection` instead.

Configuring Explicit Mapping of Relationship Fields to the Database

As "Configuring Default Mapping of Persistent Fields to the Database" on page 5-11 discusses, the container can automatically map your relationship fields to the database tables. If you do not want to accept the defaults that OC4J provides for you, or if you need to map the fields to an existing database table, then you can manually map the relationships between entity beans to an existing database table and its columns in the `orion-ejb-jar.xml` file.

> **Note:** You need to modify elements and attributes of the `<entity-deployment>` element t (see Table A–1, " Attributes of the <entity-deployment> Element" on page A-4) in the `orion-ejb-jar.xml` file to explicitly map relationship fields. JDeveloper IDE is capable of managing complex mappings between the entity beans and the database tables. Thus, JDeveloper validates the deployment descriptors and prevents inconsistencies. You can modify the `orion-ejb-jar.xml` file on your own; however, Oracle suggests that you use JDeveloper for modifying container-managed relationships. For more information about JDeveloper, see "Using JDeveloper" on page 2-4.

To manually match an existing database to the entity bean's mappings, modify the `orion-ejb-jar.xml` file by following this procedure:

1. Deploy your entity bean with container-managed persistence with the `<autocreate-tables>` element set to `false` in the `orion-application.xml` file.
2. Copy the `orion-ejb-jar.xml` file from the `application-deployments/` directory to your development directory.
3. Modify the `<data-source>` element to point to the correct data source. Note that all beans that are associated with each other must use the same data source.
4. Modify the `table` attribute to point to the correct table. Make sure that it is the correct table for the bean that is defined in the `<entity-deployments>` element.
5. Modify the `persistence-name` attributes to point to the correct column for each bean's persistence type, whether a persistent or relationship field.
6. Set the `<autocreate-tables>` element in `orion-application.xml` file to `true`.
7. Rearchive and redeploy your application.

To manually modify mapping elements if there is no existing database to match to your entity bean's mappings, follow this procedure:

1. Deploy your bean with the <autocreate-tables> element set to false in the orion-application.xml file and the ejb-jar.xml elements configured.

   OC4J creates an orion-ejb-jar.xml file with the default mappings in it. It is easier to modify these fields than to create them.

2. Copy the container-created orion-ejb-jar.xml file from the J2EE_HOME/application-deployments directory to your development environment.

3. Modify the <entity-deployment> element (see Table A–1, " Attributes of the <entity-deployment> Element" on page A-4) in the orion-ejb-jar.xml file to use the database table and columns that you specify based on the relationship type. For more information, see"Configuring orion-ejb-jar.xml to Map Bean Relationships to Database Tables" on page 5-19.

4. Set the <autocreate-tables> element in orion-application.xml file to true.

5. Rearchive and redeploy your application.

> **Note:** If you deploy your application without setting <autocreate-tables> to false, then OC4J automatically creates the default tables. You must delete all these tables before redeploying the application. You must also delete an association table, if any.

Configuring orion-ejb-jar.xml to Map Bean Relationships to Database Tables

The relationship between the entity beans is defined in the <relationships> element in the ejb-jar.xml file. The mapping between the entity bean and the database table and columns is specified in the <entity-deployment> element in the orion-ejb-jar.xml file.

The orion-ejb-jar.xml file maps the bean relationships to database table and columns within a <cmp-field-mapping> element. The following is the XML structure of the <entity-deployment> and <cmp-field-mapping> elements for a simple one-to-one relationship:

```
<entity-deployment name="SourceBeanName" location="JNDIlocation"
                   table="TableName" data-source="DataSourceJNDIName">
...
    <cmp-field-mapping name="CMRfield_name">
        <entity-ref home="targetBeanName">
            <cmp-field-mapping name="CMRfield_name"
            persistence-name="targetBean_PKcolumn" />
        </entity-ref>
    </cmp-field-mapping>
...
```

Within the <cmp-field-mapping> element, you can define the bean's name (the source of the relationship that indicates the direction), the JNDI location, the database table to which the information is persisted, and map each of the persistent and relationship fields defined in the ejb-jar.xml file to the underlying database.

The attributes of the <entity-deployment> element (see Table A–1, " Attributes of the <entity-deployment> Element" on page A-4) define the following for the bean:

- The name attribute identifies the EJB name of the bean, which was defined in the <ejb-name> element in the ejb-jar.xml file. This name attribute connects the ejb-jar.xml file definition for the bean to its mapping to the database.

- 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. The data source must be the same for all beans that interact with each other or are associated with each other. This includes beans that are in the same application, part of the same transaction, or beans that are in a parent-child relationship.

The `<cmp-field-mapping>` element in the `orion-ejb-jar.xml` file maps the following fields to database columns:

- The `<cmp-field>` element in the `ejb-jar.xml` file defines a persistent field.
- The `<cmr-field>` element in the `ejb-jar.xml` file defines a relationship field.

Example 5–6 ejb-jar.xml and orion-ejb-jar.xml Mapping for a One-To-One Relationship

Example 5–6 demonstrates how the `<cmr-field>` element in the `ejb-jar.xml` file maps to the `<cmp-field-mapping>` element in the `orion-ejb-jar.xml` file. The `name` attribute in the `<cmp-field-mapping>` provides the link between the two XML files. You must not modify any `name` attributes.

`EJB-JAR.XML`

```
<relationship-role-source>
    <ejb-name>EmpBean</ejb-name>
</relationship-role-source>
<cmr-field>
    <cmr-field-name>address</cmr-field-name>
</cmr-field>
```

`ORION-EJB-JAR.XML`

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

To fully identify and map relationship fields, nested `<cmp-field-mapping>` elements are used. The format of the nesting depends on the type of relationship. The database column that is the primary key of the target bean is defined in the `persistence-name` attribute of the internal `<cmp-field-mapping>` element. If you have an existing database, you would be modifying the `persistence-name` attributes for each `<cmp-field-mapping>` element to match your column names.

Explicit One-to-One Relationship Mapping

In a hypothetical model, there is a one-to-one unidirectional relationship (see "Direction in CMR" on page 1-3) between a single employee (represented by `EmpBean`) and his/her address (represented by `AddressBean`). The `EmpBean` points to the `AddressBean` using the relationship field `address`. These two beans will map to the `EMP` and `ADDRESS` database tables. The `EMP` table has a foreign key named `address`, which points to the primary key of the `ADDRESS` table named `AddressPK`.

Example 5–7 Explicit One-to-One Unidirectional Relationship Mapping

The beans and their relationships are specified in both the `ejb-jar.xml` and the `orion-ejb-jar.xml` deployment descriptors. As Example 5–7 shows, in the `ejb-jar.xml` file, the one-to-one relationship between the `EmpBean` and

AddressBean is defined within a <relationships> element. The direction (see "Direction in CMR" on page 1-3) is designated by one or two <cmr-field> elements.

The mapping of the beans to their database persistent storage is defined in the orion-ejb-jar.xml file. The one-to-one relationship is mapped on both sides with an <entity-ref> element inside a <cmp-field-mapping> element. The <entity-ref> describes the target entity bean of the relationship.

EJB-JAR.XML

```
<relationships>
...
    <ejb-relation>
    ...
    <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-relation>
    <ejb-relation>
    ...
        <relationship-role-source>
            <ejb-name>AddressBean</ejb-name>
        </relationship-role-source>
        ...
    </ejb-relation>
...
```

ORION-EJB-JAR.XML

```
<entity-deployment name="EmpBean"...

    <cmp-field-mapping name="address">
        <entity-ref home="AddressBean">
            <cmp-field-mapping name="address" persistentce-name="addressPK" />
        </entity-ref>
    </cmp-field-mapping>
    ...
</entity-deployment>
<entity-deployment name="AddressBean"...
    ...
    <cmp-field-mapping name="empNumber">
        <entity-ref home="EmpBean">
            <cmp-field-mapping name="empNumber" persistentce-name="empnumber" />
        </entity-ref>
    </cmp-field-mapping>
    ...
</entity-deployment>
```

To map your bean fields to an existing database, you need to understand the fields within the <cmp-field-mapping> element in the orion-ejb-jar.xml file. This element has the following structure:

```
<cmp-field-mapping name="CMRField_name">
    <entity-ref home="targetBeanName">
        <cmp-field-mapping name="CMRfield_name"
         persistence-name="targetBean_PKcolumn" />
    </entity-ref>
```

```
</cmp-field-mapping>
```

In the preceding structure example, the following occurs:

- The name attribute of the <cmp-field-mapping> element is the same as the <cmp-field> element in the ejb-jar.xml file. Do not modify the name attribute in the <cmp-field-mapping> element.
- The target bean name is specified in the home attribute of the <entity-ref> element.
- The database column that is the primary key of the target bean is defined in the persistence-name attribute of the internal <cmp-field-mapping> element. If you have an existing database, modify the persistence-name attributes for each <cmp-field-mapping> element to match your column names.

Explicit One-to-Many Relationship Mapping

In a hypothetical model, each employee (represented by EmpBean) belongs to only one department (represented by DeptBean), 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 executes a JOIN statement on both the department and employee tables and retrieves all employees with the designated department number.

This is the default behavior. If you need to change the mappings to other database tables, then use either JDeveloper, or manually modify the orion-ejb-jar.xml file to manipulate the <collection-mapping> or <set-mapping> element.

Note: Modify elements and attributes of the <entity-deployment> element (see Table A–1, " Attributes of the <entity-deployment> Element" on page A-4) in the orion-ejb-jar.xml file to explicitly map relationship fields. JDeveloper is capable of managing the complex mapping between the entity beans and the database tables. JDeveloper validates the deployment descriptors and prevents inconsistencies. Even though you can modify the orion-ejb-jar.xml file on your own, Oracle suggests you to use JDeveloper for modifying container-managed relationships. For more information about JDeveloper, see "Using JDeveloper" on page 2-4.

Example 5–8 Explicit One-to-Many Bidirectional Relationship Mapping Using Foreign Key

Example 5–8 shows the mapping for the bidirectional relationship of one department with many employees. The "one" side of the relationship is the department; the "many" side of the relationship is the employee. Example 5–8 demonstrates how to manually modify the orion-ejb-jar.xml file for this relationship to use a foreign key.

EJB-JAR.XML

```
<relationships>
    <ejb-relation>
        <ejb-relation-name>Dept-Emps</ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>Dept-has-Emps</ejb-relationship-role-name>
```

```
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>DeptBean</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>employees</cmr-field-name>
                <cmr-field-type>java.util.Set</cmr-field-type>
            </cmr-field>
        </ejb-relationship-role>
           <ejb-relationship-role-name>Emps-have-Dept</ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <cascade-delete/>
            <relationship-role-source>
                <ejb-name>EmpBean</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>dept</cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
    </ejb-relation>
</relationships>
```

ORION-EJB-JAR.XML

```
<enterprise-beans>
    <entity-deployment name="DeptBean" data-source="jdbc/scottDS" table="DEPT">
        <primkey-mapping>
            <cmp-field-mapping name="deptno" persistence-name="DEPTNO" />  /*PK*/
        </primkey-mapping>
        <cmp-field-mapping name="dname" persistence-name="DNAME" />
        <cmp-field-mapping name="employees">
        /*points from DEPTNO column in EMP to DEPTNO in DEPT*/
1.      <collection-mapping table="EMP"> /*table with FK*/
            <primkey-mapping>
                <cmp-field-mapping name="DeptBean_deptno"> /*CMR field name*/
                    <entity-ref home="DeptBean"> /*points to DeptBean*/
2.                      <cmp-field-mapping name="DeptBean_deptno"
                        persistence-name="EDEPTNO"/>
                    </entity-ref>
                </cmp-field-mapping>
            </primkey-mapping>
            <value-mapping type="mypackage1.EmpLocal">
                <cmp-field-mapping name="EmpBean_empnumber">
                    <entity-ref home="EmpBean">
                        <cmp-field-mapping name="EmpBean_empnumber"
                        persistence-name="EMPNUMBER"/>
                    </entity-ref>
                </cmp-field-mapping>
            </value-mapping>
        </collection-mapping>
        </cmp-field-mapping>
    </entity-deployment>
    <entity-deployment name="EmpBean" data-source="jdbc/scottDS" table="EMP">
        <primkey-mapping>
            <cmp-field-mapping name="empNumber" persistence-name="EMPNUMBER"/>
        </primkey-mapping>
        <cmp-field-mapping name="empName" persistence-name="ENAME" />
        <cmp-field-mapping name="salary" persistence-name="SAL" />
        <cmp-field-mapping name="dept"> /*foreign key*/
            <entity-ref home="DeptBean">
2.              <cmp-field-mapping name="dept" persistence-name="EDEPTNO" />
```

```
            </entity-ref>
        </cmp-field-mapping>
    </entity-deployment>
</enterprise-beans>
```

In the preceding `orion-ejb-jar.xml` example, if the table identified in the `<collection-mapping>` or `<set-mapping>` element of the "one" relationship (the department) is the name of the target bean's table (the employee bean table), then the one-to-many relationship is defined with a foreign key. For example, the `table` attribute in the department definition is `EMP`.

The foreign key is defined in the database table of the "many" relationship. In the preceding example, the `EDEPTNO` foreign key column exists in the `EMP` database table. This is defined in a `persistence-name` attribute of the `<cmp-field-mapping>` element in the `EmpBean` configuration.

Thus, to manipulate the `<collection-mapping>` or `<set-mapping>` element in the `orion-ejb-jar.xml` file, modify the `<entity-deployment>` element (see Table A–1, " Attributes of the <entity-deployment> Element" on page A-4) for the "one" entity bean (which contains the `Collection`), as follows:

1. Modify the table in the `<collection-mapping>` or `<set-mapping>` table attribute in the "one" relationship to be the database table of the "many" relationship. In this example, you would modify this attribute to be the `EMP` table.
2. Modify the foreign key that points to the "one" relationship within the "many" relationship configuration. In this example, modify the `<cmp-field-mapping>` element to specify the `EDEPTNO` foreign key in the `persistence-name` attribute.

Configuring Database Isolation Levels

You can configure one of the database isolation levels (see "Entity Bean Database Isolation Levels and Resource Contention" on page 1-9) for a specific bean. That is, you can specify that when the bean starts a transaction, the database isolation level for this bean be what is specified in the OC4J-specific deployment descriptor (on parallel execution or data consistency).

> **Note:** Once set, the isolation level for the bean is valid for the entire transaction.

You can set the isolation level for each entity bean in the `isolation` attribute of the `<entity-deployment>` element (see Table A–1, " Attributes of the <entity-deployment> Element" on page A-4). You can use `committed` or `serializable` value with `committed` being the default. To change this default value to `serializable`, configure the following in the `orion-ejb-jar.xml` for the intended bean:

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

The `serializable` isolation level provides data consistency; the `committed` isolation level enables the parallel execution.

WARNING: Do not set the isolation level to `serializable` if you are using a nonemulated data source. If you do use this setting, the nonemulated data source will not work.

WARNING: There is a danger of lost updates with the `serializable` level if the `<max-tx-retries>` element value in the OC4J-specific deployment descriptor is greater than `0` (with `0` being the default). If this element is set to greater than `0`, then the container retries the update if a second blocked client receives an `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` isolation level, consider leaving the `<max-tx-retries>` element as `0` for data consistency.

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

You can specify both the entity bean concurrency mode and database isolation level, if the combination affects the outcome of your resource contention. See "Combining Entity Bean Database Isolation Level and Concurrency Mode" on page 1-9 for more information.

For more information about resource contention, see "Avoiding Database Resource Contention" on page 1-8.

Configuring Concurrency Modes

The concurrency modes determine when to block to manage resource contention, or when to execute in parallel. For more information, see "Entity Bean Concurrency Modes and Resource Contention" on page 1-9.

To set the entity bean with container-managed persistence's concurrency mode, add the appropriate concurrency value of `pessimistic`, `optimistic`, or `read-only` to the `locking-mode` attribute of the `<entity-deployment>` element (see Table A–1, " Attributes of the <entity-deployment> Element" on page A-4) in the OC4J-specific deployment descriptor (`orion-ejb-jar.xml` file). The default concurrency mode is `optimistic`. To change the concurrency mode to `pessimistic`, perform the following modifications:

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

The concurrency modes are defined on a per-bean basis and the effects of locking apply on the transaction boundaries.

Parallel execution requires the correct setting of the pool size for wrapper and bean instances.

For more information, see the following:

- "Avoiding Database Resource Contention" on page 1-8.
- "Combining Entity Bean Database Isolation Level and Concurrency Mode" on page 1-9.

Configuring Exclusive Write Access to the Database

The `exclusive-write-access` attribute of the `<entity-deployment>` element (see Table A–1, " Attributes of the <entity-deployment> Element" on page A-4) states that this is the only entity 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 manually or from another bean, the results are not automatically updated within this bean.

The default value of the `exclusive-write-access` attribute is `false`. Because of the effects of the entity bean concurrency modes (see "Configuring Concurrency Modes" on page 5-25), you can only set this element to `true` for a `read-only` entity bean. OC4J always resets this attribute to `false` for `pessimistic` and `optimistic` concurrency modes.

For more information, see the following:

- "Configuring Concurrency Modes" on page 5-25.
- "Configuring Database Isolation Levels" on page 5-24.
- "Avoiding Database Resource Contention" on page 1-8.

Configuring Callback Methods for EJB 2.0 Entity Beans With Container-Managed Persistence

In your entity bean class (see "Implementing the Entity Bean Class" on page 4-4), you have to provide the following configurations of the entity bean's callback methods (see "Callback Methods" on page 1-4):

- Provide the `ejbCreate` method with parameters matching the associated `create` method defined in the home interface (see "Implementing the Entity Bean Home Interface" on page 4-2). See the implementation of this method in the Example 4–3, "Implementing the Entity Bean Class" on page 4-4.
- Set the primary key (see "Configuring Primary Key" on page 5-1) in the `ejbCreate` method. See the implementation of this method in the Example 4–3, "Implementing the Entity Bean Class" on page 4-4
- Define the primary key relationships, if any, in the `ejbCreate` method. See the implementation of this method in the Example 4–3, "Implementing the Entity Bean Class" on page 4-4.
- Define the `javax.ejb.EntityContext` for your bean in the `setEntityContext` method (see the implementation of this method in the Example 4–3, "Implementing the Entity Bean Class" on page 4-4). You can also allocate any other resources that will exist for the lifetime of the bean within this method.

- Unset (set to `null`) the bean's associated entity context in the `setEntityContext` method. Release other resources, if any, in the `setEntityContext` method. See the implementation of this method in the Example 4–3, "Implementing the Entity Bean Class" on page 4-4).

The rest of the callback methods only require an empty implementation in the entity bean class–the container provides the full implementation for these methods. However, depending on the logic of your entity bean implementation, you may take actions similar to the following:

- Define the foreign key (see "Configuring Foreign Key in a Composite Primary Key" on page 5-4) in the `ejbPostCreate` method.
- Retrieve resources in the `ejbActivate` method and release them in the `ejbPassivate` method.

6

Implementing Query Methods for an Entity Bean With Container-Managed Persistence

You can express your entity bean queries using finder or select methods. This chapter provides details on implementing Orion EJB 2.0 EJB QL finder and select methods.

This chapter includes information on the following topics:

- Implementing EJB QL Finder Methods
- Implementing EJB QL Select Methods

For more information, see the following:

- "Querying for an Entity Bean" on page 1-6
- "Implementing an EJB 2.0 Entity Bean With Container-Managed Persistence" on page 4-1

Implementing EJB QL Finder Methods

This section describes the following:

- Specifying Finder Methods Using EJB QL Syntax
- Specifying Finder Methods Using OC4J-specific Syntax

Finder methods (`ejbFind`) define search criteria that let you query for an entity bean (see "Understanding Finder Methods" on page 1-8 for more information).

To define finder methods, follow this procedure:

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 entity bean class definition. The container returns the appropriate home interface type.
2. Define the full query or just the conditional statement (the `WHERE` clause) for the finder method in the deployment descriptor.

   You can define the query using either EJB QL syntax or OC4J-specific syntax. You can specify either a full query or only the conditional part of the query (the `WHERE` clause):

   – EJB QL syntax is defined within the `ejb-jar.xml` file. 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.

Note: In EJB 2.0, EJB QL has limited support for `GROUP BY` and `ORDER BY` functions, such as `AVERAGE` and `SUM`.

See "Specifying Finder Methods Using EJB QL Syntax" on page 6-2 for more information.

– OC4J-specific syntax is defined within the `orion-ejb-jar.xml` file. When you deploy your application, OC4J translates the EJB QL syntax into the OC4J-specific syntax, which is specified in the query attribute of the `<finder-method>` element. You can modify the statement in the query attribute for a more complex query using the OC4J syntax. The OC4J-specific query statement in the `orion-ejb-jar.xml` file takes precedence over the corresponding EJB QL statement in the `ejb-jar.xml` file.

  See "Specifying Finder Methods Using OC4J-specific Syntax" on page 6-3 for more information.

Note: Finder methods must throw `FinderException`.

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.

Specifying Finder Methods Using EJB QL Syntax

There are two steps in creating a finder method using EJB QL syntax:

1. Defining Finder Methods in the Home Interface
2. Using the Deployment Descriptor to Provide the Finder Methods Definition

Defining Finder Methods 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 `findByEmployeeNumber` method in the home interface, as follows:

```
public EmployeeLocal findByEmployeeNumber(Integer empNumber) throws
FinderException;
```

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

Using the Deployment Descriptor to Provide the Finder Methods Definition

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

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

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

Specifying Finder Methods Using OC4J-specific Syntax

There are two steps in creating a finder method using OC4J-specific syntax:

1. Adding Finder Methods to the Home Interface
2. Using the OC4J-specific Deployment Descriptor to Define Finder Methods

Adding Finder Methods to the Home Interface

You must first add the finder method to the home interface. For example, with the `Employee` entity bean, to retrieve all employees, you would define the `findAll` method within the home interface, as follows:

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

Using the OC4J-specific Deployment Descriptor to Define Finder Methods

After specifying the finder method in the home interface, modify the `orion-ejb-jar.xml` file by adding the finder method query.

The `<finder-method>` element defines all finder methods, except the `findByPrimaryKey` method. The simplest finder method to define is the `findAll` method. The `query` attribute in the `<finder-method>` element can specify a full query or just the `WHERE` clause for the query. If you want all rows retrieved, then specify an empty query (`query=""`) as it returns all records.

OC4J-specific finder methods are configured in the `orion-ejb-jar.xml` file in a `<finder-method>` element. Each `<finder-method>` element specifies a partial or full SQL statement in its `query` attribute, as follows:

```
/*the empty WHERE clause finds all*/
<finder-method query="">
```

OR

```
/*finds all records where employee equals the first input parameter*/
<finder-method query="$empName=$1">
```

If you have a `<finder-method>` element with a `query` attribute, it takes precedence over any EJB QL modifications to the same method in the `ejb-jar.xml` file.

To define a complex finder method, follow this procedure:

1. In the `ejb-jar.xml` file, define a simple query using EJB QL.
2. Deploy the application. When you deploy, OC4J translates the EJB QL statement to the OC4J-specific equivalent. The full underlying SQL statement to be executed is displayed in a comment (see Example 6–1, "OC4J-specific Syntax for findAll Method" on page 6-4).
3. Modify the `query` attribute of the `<finder-method>` element in the `orion-ejb-jar.xml` file to achieve the desired level of complexity. When you redeploy, OC4J translates the new query and outputs a new comment with the SQL statement to be executed. Check the comment to verify that you have the right syntax.

If you want to use the EJB QL syntax and you have an existing definition in `orion-ejb-jar.xml` file, then do the following:

1. Erase the `query` attribute of the `<finder-method>` element in the `orion-ejb-jar.xml` file.
2. Redeploy the application. OC4J notices that the `query` attribute of the `<finder-method>` element is not present and uses the EJB QL methodology from the `ejb-jar.xml` file instead.

WARNING: There are limitations in the way OC4J generates SQL statements based on the partial finder query when executing complex queries that involve CMR. For example, there is no notation to dereference across a CMR join. You have to set the `partial` attribute of the `<finder-method>` element to `false` and use raw SQL when dealing with columns for container-managed relationships. For more information on partial queries, see "Element Description" on page A-9 and Example 6–3 on page 6-5.

Note that this limitation is partially caused by the EJB 2.0 specification not supporting the `LIKE` clause.

Example 6–1 OC4J-specific Syntax for findAll Method

Example 6–1 demonstrates the retrieval of all records from the `EmployeeBean`. The finder method name is `findAll`. This method requires no parameters because it returns a `Collection` of all employees.

```
<finder-method query="">
<!-- Generated SQL: "select EmployeeBean.empNumber,
      EmployeeBean.empName, EmployeeBean.salary from EmployeeBean" -->
    <method>
        <ejb-name>EmployeeBean</ejb-name>
        <method-name>findAll</method-name>
            <method-params></method-params>
    </method>
</finder-method>
```

Note: If you wish to add specifics to your query, you can modify the `query` attribute with the appropriate `WHERE` clause. This clause refers to passed-in parameters using the dollar ($) symbol: the first parameter is denoted by `$1`, the second by `$2`, and so forth. All `<cmp-field>` elements that are used within the `WHERE` clause are denoted by `$<cmp-field>` name.

Example 6–2 OC4J-specific Syntax for findByName Method

Example 6–2 demonstrates the specification of a `findByName` method (which should be defined in the home interface). The name of the employee is given as in the method parameter, which is substituted for the `$1`. It is matched to the CMP name, `empName`. Thus, the `query` attribute is modified to contain `$empName=$1` for the `WHERE` clause.

```
<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>
```

> **Note:** If you have more than one method parameter, define each parameter type in successive `<method-param>` elements and refer to it in the query statement by successive `$n`, where `n` represents the number.

> **Note:** You can specify a SQL `JOIN` in the `query` attribute.

Example 6–3 OC4J-specific Syntax for the Full Query

Example 6–3 shows how to specify a full query instead of just the section after the `WHERE` clause. In this case, you would have to set the `partial` attribute to `false`, and then define the full query in the `query` attribute. The default value for the `partial` attribute is `true` (that is the reason it is not specified in the Example 6–2).

```
<finder-method partial="false" query="select * from emp where $empName=$1">
    <!-- Generated SQL: "select * from emp where emp.empName=?" -->
    <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>
```

> **Note:** When using a generated SQL statement as the basis for a partial finder query, you need to ensure the following:
>
> - SQL-specific `?` placeholders are translated to `$n` positional parameters.
> - certain XML characters, such as >, <, >=, are escaped.

Example 6–4 OC4J-specific Syntax for Enabling Lazy Loading

For entity bean finder methods, lazy loading can cause the select method (see "Understanding Select Methods" on page 1-8) to be invoked more than once. By default, lazy loading is turned off. If you are retrieving large numbers of objects, and you are accessing only a few of them, you should turn on lazy loading by setting the `lazy-loading` property to `true`, as Example 6–4 shows:

```
<finder-method partial="false" query="select * from emp where $empName=$1"
lazy-loading=true>
    <!-- Generated SQL: "select * from emp where emp.empName=?" -->
    <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>
```

Example 6–5 OC4J-specific Syntax for Setting the Fetch Size

You can define how many rows at a time the JDBC driver fetches by setting the `prefetch-size` attribute, as Example 6–5 shows:

```
<finder-method partial="false" query="select * from emp where $empName=$1"
prefetch-size="15">
    <!-- Generated SQL: "select * from emp where emp.empName=?" -->
    <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>
```

Oracle JDBC drivers include extensions that let you set the number of rows to prefetch into the client while a result set is being populated during a query. This reduces round trips to the database by fetching multiple rows of data each time data is fetched (the extra data is stored in client-side buffers for later access by the client). The number of rows to prefetch can be set as desired. The default number of rows to prefetch to the client is 10. This number is passed to the JDBC driver.

Implementing EJB QL Select Methods

Select methods (`ejbSelect`) are used to retrieve entity bean references or values of container-managed persistent or relationship fields (see "Understanding Select Methods" on page 1-8).

The format for an `ejbSelect` method definition is as follows:

```
public abstract type ejbSelect<METHOD>(...);
```

Although the select method is not based on the identity of the entity bean instance on which it is invoked, it can use the primary key of an entity bean as an argument. This creates a query that is logically scoped to a particular entity bean instance. For information on how to define the select method return type, see "Defining the Return Type for the Select Method" on page 6-8.

To define a select method, follow this procedure:

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 full query or the conditional statement alone (the `WHERE` clause) 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.

Note: You cannot modify the query statement for an ejbSelect method in the orion-ejb-jar.xml file, as you can for finder methods.

Example 6–6 Defining the Select Method in the Bean Class

Example 6–6 demonstrates the definition of a select method to retrieve all employees whose salary falls within a specified range:

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

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

Example 6–7 Providing the Select Method Definition in the Deployment Descriptor

Each select method is defined in the deployment descriptor in a <query> element. Example 6–7 shows the deployment descriptor for the ejbSelectBySalaryRange method defined in the bean class in Example 6–6:

```
<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(emp) From Employee emp
            WHERE emp.salary BETWEEN ?1 AND ?2</ejb-ql>
</query>
```

The ejbSelectBySalaryRange method has two input parameters 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. The ejbSelectBySalaryRange method evaluates the persistent field of salary within the EJB QL statement by the emp.salary. The emp represents the Employee object; the salary represents the persistent field within that object. The separating period between them indicates the relationship between the entity bean and its persistent field.

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

The ejbSelectBySalaryRange method returns objects, where the DISTINCT keyword ensures that no duplicate records are returned.

Note: Select methods must throw FinderException.

Defining the Return Type for the Select Method

The following is the list of conditions that you must consider when defining return types for your select methods:

- If the select method does not find any objects, a `FinderException` is raised.
- If you want your select method to find a single object, the container returns the same type as returned in the `ejbSelect<NAME>` method. If multiple objects are returned, a `FinderException` is raised.
- If you want your select method to find multiple objects, 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.
  - If your select method returns a 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>
    ```

  - If your select method returns a `Set` or `Collection` of CMP values, the container determines the object type from the EJB QL select statement.

A

XML Reference for orion-ejb-jar.xml Elements

This appendix describes the elements contained within the `orion-ejb-jar.dtd`–the OC4J-specific EJB deployment descriptor. 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`.

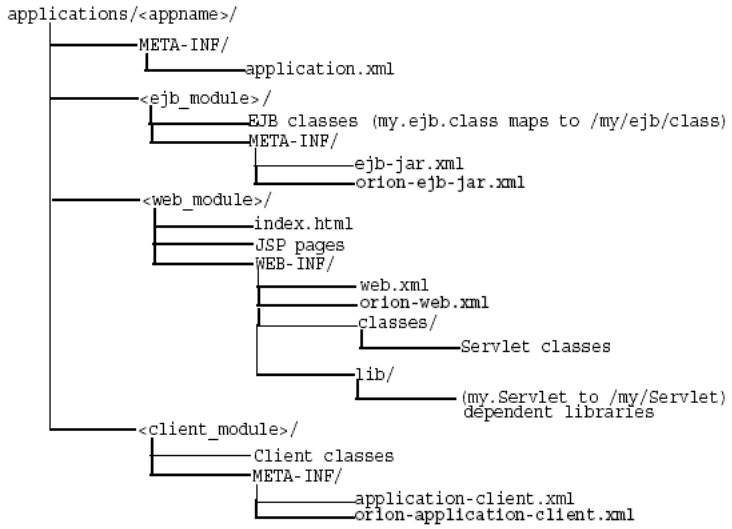
The description of this deployment descriptor has been divided into the following sections:

- Overall description of each element section–Each section of elements of this XML file is described in "OC4J-specific Deployment Descriptor for EJB" on page A-2.
- Element description–An alphabetical listing and description for each element is discussed in "Element Description" on page A-9.

Whenever you deploy an application, OC4J automatically generates the OC4J-specific XML file with the default elements. If you want to change these defaults, you must copy the `orion-ejb-jar.xml` file to where your original `ejb-jar.xml` file is located and change it in this location. If you change the XML file within the deployed location, OC4J overwrites these changes when the application is deployed again. The changes only stay constant when changed in the development directories.

You should add your OC4J-specific XML files within the recommended development structure, similar to the one that Figure A–1 shows:

Figure A–1 Development Application Directory Structure



OC4J-specific Deployment Descriptor for EJB

The OC4J-specific deployment descriptor contains extended deployment information for entity beans and the security for these beans. The major element structure of interest within this deployment descriptor is as follows:

```
<orion-ejb-jar deployment-time=... deployment-version=...>
    <enterprise-beans>
        <entity-deployment ...></entity-deployment>
    </enterprise-beans>
    <assembly-descriptor>
        <security-role-mapping ...></security-role-mapping>
        <default-method-access></default-method-access>
    </assembly-descriptor>
</orion-ejb-jar>
```

Each section under the `<orion-ejb-jar>` main tag has its own purpose. These are described in the following sections:

- Enterprise Beans Section
- Assembly Descriptor Section

Enterprise Beans Section

The `<enterprise-beans>` section defines additional deployment information for all EJB. There is a section for each type of EJB.

The following sections describe the elements of interest (entity beans with container-managed persistence) within the `<enterprise-beans>` element:

- Entity Bean Section
- Method Definition

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=... isolation=...
        location=... locking-mode=... max-instances=... min-instances=...
        max-tx-retries=... tx-retry-wait=... update-changed-fields-only=...
        name=... pool-cache-timeout=...
        table=... validity-timeout=... force-update=...
        wrapper=... local-wrapper=... delay-updates-until-commit=...
        findByPrimaryKey-lazy-loading=... >
    <ior-security-config>
        <transport-config>
            <integrity></integrity>
            <confidentiality></confidentiality>
            <establish-trust-in-target></establish-trust-in-target>
            <establish-trust-in-client></establish-trust-in-client>
        </transport-config>
        <as-context>
            <auth-method></auth-method>
            <realm></realm>
            <required></required>
        </as-context>
        <sas-context>
            <caller-propagation></caller-propagation>
        </sas-context>
    </ior-security-config>
    <primkey-mapping>
        <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
                    persistence-type=...></cmp-field-mapping>
    </primkey-mapping>
    <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
                    persistence-type=...> </cmp-field-mapping>
    <finder-method partial=... query=... lazy-loading=... prefetch-size=... >
        <method></method>
    </finder-method>
    <env-entry-mapping name=...></env-entry-mapping>
    <ejb-ref-mapping location=... name=... />
    <resource-ref-mapping location=... name=... >
        <lookup-context location=...>
            <context-attribute name=... value=... />
        </lookup-context>
    </resource-ref-mapping>
    <resource-env-ref-mapping location=... name=... />
</entity-deployment>
```

Each of the element groups are discussed in the following sections of the OC4J documentation set:

- Entity bean examples, which include the `<entity-deployment>` element, are described in Chapter 1, "Understanding Entity Beans With Container-Managed Persistence", Chapter 3, "Understanding Orion CMP Support in OC4J" and Chapter 5, "Configuring an EJB 2.0 Entity Bean With Container-Managed Persistence" of this book.

- The `<io-security-config>` element configures CSIv2 security policies for interoperability, which is discussed fully in the Interoperability chapter in the *Oracle Containers for J2EE Services Guide.*
- The `<primkey-mapping>` element maps the primary key to the container-managed persistent field it represents. See "Configuring Explicit Mapping of Persistent Fields to the Database" on page 5-12 for more information.
- The `<cmp-field-mapping>` element maps each `<cmp-field>` element to its database row. See "Configuring Explicit Mapping of Persistent Fields to the Database" on page 5-12 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 "Implementing EJB QL Finder Methods" on page 6-1.
- The `<env-entry-mapping>` element maps environment variables to JNDI names.
- The `<ejb-ref-mapping>` element maps any EJB references to JNDI names.
- The `<resource-ref-mapping>` element maps any EJB references to JNDI names.
- The `<resource-env-ref-mapping>` element is used to map an administered object for a resource.

Table A–1 lists the attributes for the `<entity-deployment>` element:

Table A–1 Attributes of the <entity-deployment> Element

Attribute	Description
`call-timeout`	Specifies the maximum time to wait for any resource to make a business/life cycle method invocation. This is not a timeout of how long a business method invocation can take. If the timeout is reached, a `TimeOutException` is thrown. This excludes database connections. The default value is 90000 milliseconds. Set to 0 if you want the timeout to be forever. See the EJB section in the *Oracle Application Server Performance Guide* for more information.
`clustering-schema`	Do not use. Not applicable.
`copy-by-value`	Indicates whether or not to copy (clone) all the incoming and outgoing parameters in EJB calls. Set to `false` if you are certain that your application does not assume copy-by-value semantics for a speed-up. The default value is `true`.
`data-source`	Specifies the name of the data source used.

Table A–1 (Cont.) Attributes of the <entity-deployment> Element

Attribute	Description
`exclusive-write-access`	Defines whether or not the EJB server has exclusive write (update) access to the database back-end. This can be used only for entity beans that use a `read-only` locking mode (see "Configuring Concurrency Modes" on page 5-25). 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 2.0 specification). When `exclusive-write-access=true`, this is commit option A. The default value 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 (see "Entity Bean Concurrency Modes and Clustering" on page 1-10). The `exclusive-write-access` can be `false` with `read-only` locking, but `read-only` will not have any performance impact if `exclusive-write-access=false`, since `ejbStore` methods are already skipped when no fields have been changed. To see a performance advantage and avoid the execution of `ejbLoad` methods for `read-only` beans, you must also set `exclusive-write-access=true`. See "Configuring Exclusive Write Access to the Database" on page 5-26 for more information.
`do-select-before-insert`	Specifies whether or not to avoid executing a select before an insert. Set it to `false` if you choose to avoid executing a select before an insert. The extra select normally checks to see if the entity already exists before doing the insert to avoid duplicates. If a unique key constraint is defined for the entity, then you should set this to `false`. If there is no unique key constraint, setting this to `false` leads to not detecting a duplicate insert. To prevent duplicate inserts in this case, leave it set to `true`. For performance, Oracle recommends setting this to `false` to avoid the extra select before insert. The default value is `true`.
`location`	Specifies the JNDI name to which this bean will be bound.
`isolation`	Specifies the isolation level (see "Configuring Database Isolation Levels" on page 5-24) for database actions. The valid values for Oracle databases are `serializable` and `committed`, with `committed` being the default. Non-Oracle databases can be the following: `none`, `committed`, `serializable`, `uncommitted`, and `repeatable_read`. For more information, see "Entity Bean Database Isolation Levels and Resource Contention" on page 1-9, "Configuring Database Isolation Levels" on page 5-24 and *Oracle Application Server Performance Guide.*
`locking-mode`	Specifies the concurrency mode (see "Configuring Concurrency Modes" on page 5-25 and *Oracle Application Server Performance Guide).* The concurrency modes are as follows: `pessimistic`, `optimistic`, `read-only`.

Table A–1 (Cont.) Attributes of the <entity-deployment> Element

Attribute	Description
`max-instances`	Specifies the number of maximum bean implementation instances to be kept instantiated or pooled. The default value is 0, which means infinite.
`min-instances`	Specifies the number of minimum bean implementation instances to be kept instantiated or pooled. The default value is 0.
`max-tx-retries`	Specifies the number of times to retry a transaction that was rolled back due to system-level failures. Generally, you should add retries only where errors are seen that could be resolved through retries. For example, if you are using serializable isolation and you want to retry the transaction automatically if there is a conflict, you might want to use retries. However, if the bean wants to be notified when there is a conflict, then you should leave `max-tx-retries=0`. The default value is 0. See the EJB section in the *Oracle Application Server Performance Guide* for more information.
`tx-retry-wait`	Specifies the time to wait in seconds between retrying the transaction. The default value is 60.
`update-changed-fields-only`	Specifies whether the container updates only modified fields or all fields to persistence storage for entity beans with container-managed persistence when `ejbStore` method is invoked. The default value is `true`, which specifies to only update modified fields.
`name`	Specifies the name of the bean, which matches the name of a bean in the assembly section of the EJB deployment descriptor.
`pool-cache-timeout`	Specifies 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 value is 60.
`table`	Specifies the name of the table in the database.

Table A–1 (Cont.) Attributes of the <entity-deployment> Element

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 concurrency mode of read-only and when exclusive-write-access=true (default). If the data is never being modified externally (and, therefore, you have set exclusive-write-access=true), you should set this to 0 or -1 to disable this option, since the data in the cache will always be valid for read-only beans that are never modified externally. If the bean is generally not modified externally, so you are using exclusive-write-access=true, yet occasionally the table is updated so you need to update the cache sometimes, 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 bean's life cycle by invoking the ejbStore method. This manages data in transient fields and sets appropriate persistent fields during the ejbStore method. For example, an image might be kept in one format in memory, but stored in a different format in the database. The default value is false.
wrapper	Specifies the name of the OC4J remote home wrapper class for this bean. This is an internal server value and should not be edited.
local-wrapper	Specifies the 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	Specifies whether or not to defer the flushing of transactional data until commit time. The default value is true. Set this value to false to update persistence data after completion of every EJB method invocation, except ejbRemove method and the finder methods.

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 the *Oracle Containers for J2EE Services Guide*.

The <jem-deployment> section provides additional deployment information for an active bean 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 you control or restrict the usage of the asynchronous methods defined on the AC4J bean at deployment time. 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 the *Oracle Containers for J2EE Services Guide*.

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>
```

You can use one of the following styles:

1. When referring to all the methods of the specified 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.

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>
```

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 execute the AC4J beans are identified in one or more `<caller>` elements.

<caller>
Each caller identity that is allowed to execute methods on the AC4J bean is 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 persistent 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 field's 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 of the field. Valid values vary from database to database.

<collection-mapping>
A relational mapping of a `Collection` type. A `Collection` consists of *n* unordered items (order is not specified and not relevant). The field containing the mapping must be of type `java.util.Collection`.

Attributes:

- `table`–The name of the table in the database.

<context-attribute>
An attribute that is sent to the context. The only mandatory attribute in JNDI is the `java.naming.factory.initial`, which is the class name of the context factory implementation.

Attributes:

- `name`–The name of the attribute.
- `value`–The value of the attribute.

<data-bus>
The name and URL of a specific Databus for an OC4J object.

Attributes:

- `data-bus-name`–The user-defined name of the Databus.
- `url`–The URL of the Databus, which is similar to a JDBC URL.

<default-method-access>
The default method access policy for methods not tied to a `method-permission`.

<description>
A short description.

<ejb-name>
An enterprise bean's name. This name is assigned by the `ejb-jar.xml` file producer to name the enterprise bean in deployment descriptor. The name must be unique among the names of the enterprise beans in the same `ejb-jar.xml` 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>` element 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>
Declares a reference to another enterprise bean's home. This element ties this to a JNDI location at deployment time.

Attributes:

- location–The JNDI location, in which to search for the EJB home.
- name–The ejb-ref's name. Matches the name of an ejb-ref in ejb-jar.xml file.

<enterprise-beans>
Lists the beans contained in this EJB JAR file.

<entity-deployment>
Contains the deployment information for an entity bean. See Table A–1 for list and description of attributes of this element.

<entity-ref>
Specifies the configuration for persisting an entity reference using its primary key. The subtag of this tag is the specification of how to persist the primary key.

Attributes:

- home–The JNDI location of the EJBHome, in which to search for the bean.

<env-entry-mapping>
Overrides the value of an <env-entry> element in the assembly descriptor. It is used to keep the EAR clean from deployment-specific values. The body of the element represents 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 a zero-argument constructor.

<finder-method>
The definition of a container-managed finder method. This defines the selection criteria in a findBy<CRITERION> method in the bean's home.

Attributes:

- partial–Specifies whether or not the 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 persistent fields. This is useful when performing advanced queries involving table joins.
- query–Defines the query part of a SQL statement. This is the section following the WHERE keyword in the statement. Special tokens are $number, which denotes a method argument number, and $name, which denotes a persistent field name. For instance, the query for findByAge(int age) would be "$1 = $age" (assuming the persistent field is named age).

- `lazy-loading`–For entity bean finder methods, lazy loading can cause the select method to be invoked more than once. To turn on lazy loading and enforce only a single execution of this finder method, set this property to `true`. The default value is `false`.
- `prefetch-size`–Oracle JDBC drivers include extensions that let you set the number of rows to prefetch into the client, while a result set is being populated during a query. This reduces round trips to the database by fetching multiple rows of data each time data is fetched–the extra data is stored in client-side buffers for later access by the client. You can set any number of rows to prefetch. The default number of rows to prefetch to the client is 10. The number set here is passed to the JDBC driver. See the *Oracle Database JDBC Developer's Guide and Reference* for more information on using prefetch with a JDBC driver.

<group>
A group that this `<security-role-mapping>` implies. That is, all members of the specified group are included in this role.

Attributes:

- `name`–The name of the group.

<ior-security-config>
The `<ior-security-config>` element configures CSIv2 security policies for interoperability, which is discussed fully in the Interoperability chapter in the *Oracle Containers for J2EE Services Guide*.

<jem-deployment>
Specifies an active enterprise bean for deployment into the AC4J container.

Attributes:

- `jem-name`–The AC4J name that is used to identify the enterprise bean within the AC4J calls.
- `ejb-name`–The name of the enterprise bean defined in the `ejb-jar.xml` file as an active bean.

<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. 1 is the default value.

<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. For example, `com.acme.Product`, `java.lang.String`, and so on.

<method>
Specifies the methods (and, possibly, parameters of that method) of the bean.

<method-intf>
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. This element must be either home or remote.

<method-name>
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>
Contains the fully qualified Java type name of a method parameter.

<method-params>
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 a bean. It is used to specify initial deployment properties. After each deployment, the deployment descriptor 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. This is an internal server value, do not edit it.
- `deployment-version`–The version of OC4J with which this JAR was deployed. If it is not matching the current version, then it will be redeployed. This is an internal server value, do not edit it.

<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 EJB 2.0 specification, and the type of the containing object has to have an empty constructor. This is also designated within the EJB 2.0 specification.

<resource-ref-mapping>
Declares a reference to an external resource, such as a data source. This element ties the data source to a JNDI location at deployment time.

Attributes:

- `location`–The JNDI location from which to search the resource factory.
- `name`–The `<resource-ref>` element name in `ejb-jar.xml` file.

<resource-env-ref-mapping>
Maps an administered object for a resource. These objects are retrieved at the same time from JNDI. This element maps the destination object.

Attributes:

- `location`–The JNDI location from which to search the administered resource.
- `name`–The `<resource-env-ref>` element name in `ejb-jar.xml` file.

<role-name>
The security role under which the AC4J EJB methods are run when using the `<run-as-specified-identity>` element.

<run-as-specified-identity>
Specifies 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 run-time mapping of a role to groups and users. Maps to a security role of the same name in the assembly descriptor.

Attributes:

- `impliesAll`–Indicates whether or not this mapping implies all users. The default value is `false`.
- `name`–The name of the role.

<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>
Specifies that all methods of an AC4J bean execute under the caller's identity.

<user>
A user that this `<security-role-mapping>` element implies.

Attributes:

- `name`–The name of the user.

<value-mapping>
Specifies a mapping of the primary key part of a set of fields.

Attributes:

- `immutable`–Identifies whether or not the value can be trusted to be immutable, once added to the `Collection`. Setting this to `true` will optimize database operations extensively. The default value is `true` for `<set-mapping>` element, and `false` for `<collection-mapping>` element

- `type`–The fully qualified class name of the type of the value. For example, `com.acme.OrderEntry`, `java.lang.String`, and so on.

Index

N

O

P

Q

R

S

T

U

V

W