# Oracle9*i*AS Containers for J2EE

Enterprise JavaBeans Developer's Guide and Reference

Release 2 (9.0.2)

February 2002

Part No. A95881-01

ORACLE®

Oracle9*i*AS Containers for J2EE Enterprise JavaBeans Developer’s Guide and Reference, Release 2 (9.0.2)

Part No. A95881-01

Primary Author: Sheryl Maring

# Contents

# Send Us Your Comments

**Oracle9*i*AS Containers for J2EE Enterprise JavaBeans Developer's Guide and Reference, Release 2 (9.0.2)**

**Part No. A95881-01**

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

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

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

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

Please indicate if you would like a reply.

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

# Preface

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

## Who Should Read This Guide?

Anyone developing Enterprise JavaBeans for OC4J will benefit from reading this guide. Written especially for programmers, it will also be of value to architects, systems analysts, project managers, and others interested in EJB applications. To use this guide effectively, you must have a working knowledge of J2EE.

## Prerequisite Reading

Before consulting this Guide, you should read the following:

- Any J2EE book that enables you to understand the basics of J2EE programming.
- The *Oracle9iAS Containers for J2EE User's Guide*. This guide helps you to understand the minimum requirements for a J2EE application in the OC4J environment.
- The Sun Microsystems EJB 1.1 specification as a supplement to this guide. This guide assumes that you already have a base understanding of the EJB 1.1 specification details.

## Suggested Reading

### Books

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

### Online Sources

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

`http://www.sun.com`

The current 1.1 EJB specification is available at:

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

Another popular Java Web site is:

`http://www.gamelan.com`

For Java API documentation, see:

`http://www.javasoft.com`

## How This Guide Is Organized

This guide consists of the following:

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

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

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

Chapter 4, "BMP Entity Beans", discusses a BMP entity bean.

Chapter 5, "Message-Driven Beans", discusses an MDB entity bean.

Chapter 6, "Advanced EJB Subjects", discusses advanced issues for EJBs.

Chapter 7, "EJB Clustering", discusses how to cluster EJBs across OC4J nodes.

Chapter 8, "Active Components For Java", introduces a new methodology to merge the advantages of both asynchronous and request/response communication.

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

## Documentation Accessibility

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

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

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

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

## Notational Conventions

This guide follows these conventions:

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

Java code examples follow these conventions:

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

# 1 EJB Overview

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

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

This chapter includes the following topics:

## Invoking Enterprise JavaBeans

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

- Session beans can be stateful or stateless and are used for business logic functionality.
  - Stateless session beans are used for business services. They do not retain client state across calls.
  - Stateful session beans do maintain state across client calls. Thus, these beans manage business functions for a specific client for the life of that client.
- Entity beans are normally used for managing persistent data.
- Message-driven beans are used for receiving messages from a JMS queue or topic.

An EJB has two client interfaces:

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

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

***Figure 1–1 Events In A Stateless Session Bean***

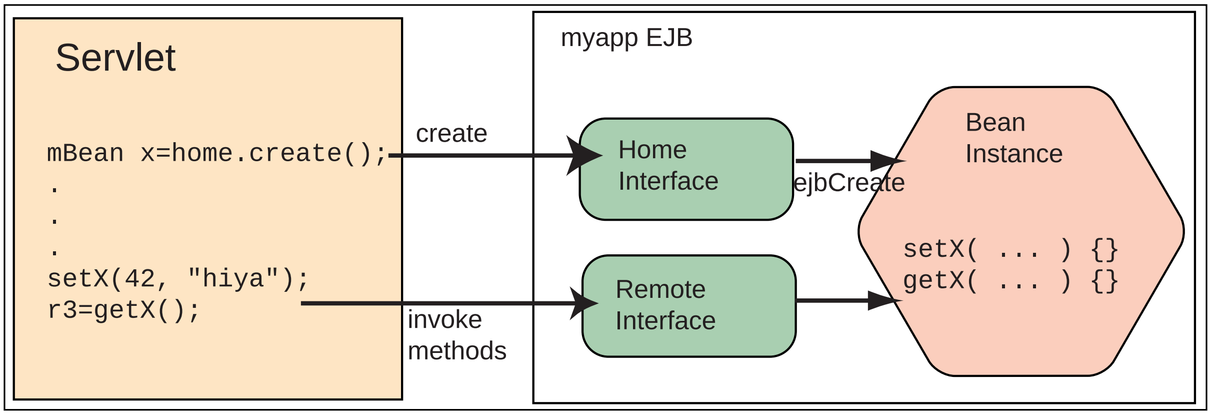


Figure 1–1 demonstrates a stateless session bean and corresponds to the following steps:

1. The client, which can be a standalone Java client, servlet, JSP, or an applet, retrieves the home interface of the bean—normally through JNDI.
2. The client invokes the `create` method on the home interface reference (home object). This creates the bean instance and returns a reference to the remote interface of the bean.
3. The client invokes a method defined in the remote interface, which delegates the method call to the corresponding method in the bean instance (through a stub).
4. The client can destroy the bean instance by invoking the `remove` method that is defined in the remote interface. Some beans, such as stateless session beans, cannot call the `remove` method. In this case, the container removes the bean.

## Implementing an EJB

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

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

| Component | Description |
|---|---|
| The home interface | Specifies the interface to an object that the container itself implements: the *home object*. The home interface contains the life cycle methods, such as the `create()` methods that specify how a bean is created. |
| The remote interface | Specifies the business methods that you implement in the bean. The bean must also implement additional container service methods. The EJB container invokes these methods at different times in the life cycle of a bean. |
| The bean implementation | Contains the Java code that implements the methods defined in the home interface (life cycle methods), remote interface (business methods), and the required container methods (container callback functions). |

| Component | Description |
|---|---|
| The deployment descriptor | Specifies attributes of the bean for deployment. These designate configuration specifics, such as environment, interface names, transactional support, type of EJB, and persistence information. |

## Bean Implementation

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

## Parameter Passing

When you implement an EJB or write the client code that calls EJB methods, you must be aware of the parameter-passing conventions used with EJBs.

A parameter that you pass to a bean method—or a return value from a bean method—can be any Java type that is serializable. Java primitive types, such as `int`, `double`, are serializable. Any non-remote object that implements the `java.io.Serializable` interface can be passed. A non-remote object that is passed as a parameter to a bean or returned from a bean is passed by *value*, not by reference. So, for example, if you call a bean method as follows:

```
public class theNumber {
  int x;
}
...
bean.method1(theNumber);
```

then `method1()` in the bean receives a copy of `theNumber`. If the bean changes the value of `theNumber` object on the server, this change is not reflected back to the client, because of pass-by-value semantics.

If the non-remote object is complex—such as a class containing several fields—only the non-static and non-transient fields are copied.

When passing a remote object as a parameter, the stub for the remote object is passed. A remote object passed as a parameter must extend remote interfaces.

The next section demonstrates parameter passing to a bean, and remote objects as return values.

## Parameter Objects

The `EmployeeBean` `getEmployee` method returns an `EmpRecord` object, so this object must be defined somewhere in the application. In this example, an `EmpRecord` class is included in the same package as the EJB interfaces.

The class is declared as `public` and must implement the `java.io.Serializable` interface so that it can be passed back to the client by value, as a serialized remote object. The declaration is as follows:

```
package employee;

public class EmpRecord implements java.io.Serializable {
  public String ename;
  public int empno;
  public double sal;
}
```

**Note:** The `java.io.Serializable` interface specifies no methods; it just indicates that the class is serializable. Therefore, there is no need to implement extra methods in the `EmpRecord` class.

## Types of EJBs

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

- Session Beans
- Entity Beans
- Message-Driven Beans

### Session Beans

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

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

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

```
public interface javax.ejb.SessionBean extends javax.ejb.EnterpriseBean {
   public abstract void ejbActivate();
   public abstract void ejbPassivate();
   public abstract void ejbRemove();
   public abstract void setSessionContext(SessionContext ctx);
}
```

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

| | |
|---|---|
| `ejbCreate()` | The container invokes this method right before it creates the bean. Stateless session beans must do nothing in this method. Stateful session beans can initiate state in this method. |
| `ejbActivate()` | The container invokes this method right after it reactivates the bean. |
| `ejbPassivate()` | The container invokes this method right before it passivates the bean. |
| `ejbRemove()` | A container invokes this method before it ends the life of the session object. This method performs any required clean-up—for example, closing external resources such as file handles. |
| `setSessionContext (SessionContext ctx)` | This method associates a bean instance with its context information. The container calls this method after the bean creation. The enterprise bean can store the reference to the context object in an instance variable, for use in transaction management. Beans that manage their own transactions can use the session context to get the transaction context. |

### Using setSessionContext

You use this method to obtain a reference to the context of the bean. Session beans have session contexts that the container maintains and makes available to the beans. The bean may use the methods in the session context to make callback requests to the container.

The container invokes `setSessionContext` method, after it first instantiates the bean, to enable the bean to retrieve the session context. The container will never call this method from within a transaction context. If the bean does not save the session context at this point, the bean will never gain access to the session context.

When the container calls this method, it passes the reference of the `SessionContext` object to the bean. The bean can then store the reference for later use. The following example shows the bean saving the session context in the **`sessctx`** variable.

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

public class myBean implements SessionBean {
   SessionContext sessctx;

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

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

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

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

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

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

***Table 1–1 SessionContext Operations***

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

There are two types of session beans:

- Stateless Session Beans—Stateless session beans do not share state or identity between method invocations. They are useful mainly in middle-tier application servers that provide a pool of beans to process frequent and brief requests.
- Stateful Session Beans—Stateful session beans are useful for conversational sessions, in which it is necessary to maintain state, such as instance variable values or transactional state, between method invocations. These session beans are mapped to a single client for the life of that client.

## Stateless Session Beans

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

The EJB container maintains a pool of these stateless beans to service multiple clients. An instance is taken out of the pool when a client sends a request. There is no need to initialize the bean with any information. There is implemented only a single `create/ejbCreate` with no parameters—containing no initialization for the bean within these methods. There is no need to implement any actions within the `remove/ejbRemove`, `ejbPassivate`, `ejbActivate`, and `setSessionContext` methods. In addition, there is no need for the intended use for these methods in a stateless session bean. Instead, these methods are used mostly for EJBs with state—for stateful session beans and entity beans. Thus, these methods should be empty or extremely simple.

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

## Stateful Session Beans

A stateful session bean maintains its state between method calls. Thus, there is one instance of a stateful session bean created for each client. Each stateful session bean contains an identity and a one-to-one mapping with an individual client. The state of this type of bean is maintained across several calls through serialization of its state, called passivation. This is why the state that you passivate must be serializable. However, this information does not survive system crashes.

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

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

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

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

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

## Entity Beans

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

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

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

### Uniquely Identified by a Primary Key

Each entity bean has a persistent identity associated with it. That is, the entity bean contains a unique identity that can be retrieved if you have the primary key—given the primary key, a client can retrieve the entity bean. If the bean is not available, the container instantiates the bean and repopulates the persistent data for you.

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

### Managing Persistent Data

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

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

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

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

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

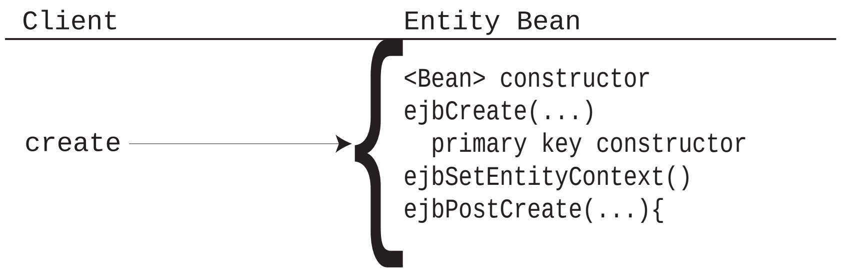
The container expects these methods to have the following functionality:

| | |
|---|---|
| ▪ `ejbCreate` | You must implement an `ejbCreate` method corresponding to each `create` method declared in the home interface. When the client invokes the `create` method, the container first invokes the constructor to instantiate the object, then it invokes the corresponding `ejbCreate` method. The `ejbCreate` method performs the following:<br>▪ creates any persistent storage for its data, such as database rows<br>▪ intializes a unique primary key and returns it |
| ▪ `ejbPostCreate` | The container invokes this method after the environment is set. For each `ejbCreate` method, an `ejbPostCreate` method must exist with the same arguments. This method can be used to initialize parameters within or from the entity context. |
| ▪ `ejbRemove` | The container invokes this method before it ends the life of the session object. This method can perform any required clean-up, for example closing external resources such as file handles. |
| ▪ `ejbStore` | The container invokes this method right before a transaction commits. It saves the persistent data to an outside resource, such as a database. |
| ▪ `ejbLoad` | The container invokes this method when the data should be reinitialized from the database. This normally occurs after activation of an entity bean. |
| ▪ `setEntityContext` | Associates the bean instance with context information. The container calls this method after the bean creation. The enterprise bean can store the reference to the context object in an instance variable, for use in transaction management. Beans that manage their own transactions can use the session context to get the transaction context.<br><br>You can also allocate any resources that will exist for the lifetime of the bean within this method. You should release these resources in `unsetEntityContext`. |

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

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

***Figure 1–2 Creating the Entity Bean***



```
<Bean> constructor
ejbCreate(...)
  primary key constructor
ejbSetEntityContext()
ejbPostCreate(...){
```

**Using setEntityContext** An entity bean instance uses this method to retain a reference to its context. Entity beans have contexts that the container maintains and makes available to the beans. The bean may use the methods in the entity context to retrieve information about the bean, such as security, and transactional role. Refer to the Enterprise JavaBeans specification from Sun Microsystems for the full range of information that you can retrieve about the bean from the context.

The container invokes the `setEntityContext` method, after it first instantiates the bean, to enable the bean to retrieve the context. The container will never call this method from within a transaction context. If the bean does not save the context at this point, the bean will never gain access to the context.

---

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

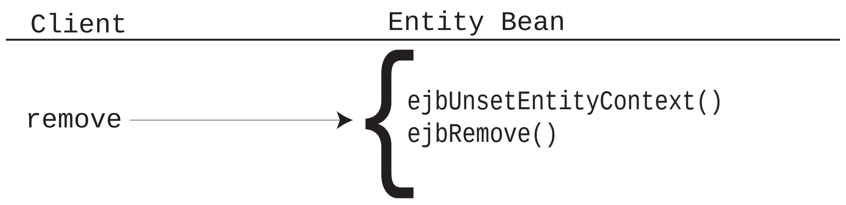
---

When the container calls this method, it passes the reference of the `EntityContext` object to the bean. The bean can then store the reference for later use. The following example shows the bean saving the context in the **`this.ctx`** variable.

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

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

***Figure 1–3 Removing the Entity Bean***



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

- The `ejbStore` method is called by the container before the object is passivated or whenever a transaction is about to end. Its purpose is to save the persistent data to an outside resource, such as a database.
- The `ejbLoad` method is called by the container before the object is activated or whenever a transaction has begun, or when an entity bean is instantiated. Its purpose is to restore any persistent data that exists for this particular bean instance.

### Container-Managed Persistence

You can choose to have the container manage your persistent data for the bean. You do not have to implement some of the callback methods to manage persistence for your bean's data, because the container stores and reloads your persistent data to and from the database. When you use container-managed persistence, the container invokes a persistence manager class that provides the persistence management business logic. In addition, you do not have to provide management for the primary key: the container provides this key for the bean.

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

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

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

| Callback Method | Functionality Required |
|---|---|
| `ejbLoad` | No functionality is required for restoring persistent data within this method. The persistence manager restores all persistent data for you. However, you must provide at least an empty implementation. |
| `setEntityContext` | Associates the bean instance with context information. The container calls this method after the bean creation. The enterprise bean can store the reference to the context object in an instance variable, for use in transaction management. Beans that manage their own transactions can use the session context to get the transaction context.<br><br>You can also allocate any resources that will exist for the lifetime of the bean within this method. You should release these resources in `unsetEntityContext`. |
| `unsetEntityContext` | Unset the associated entity context and release any resources allocated in `setEntityContext`. |

## Differences Between Bean and Container-Managed Persistence

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

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

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

## Message-Driven Beans

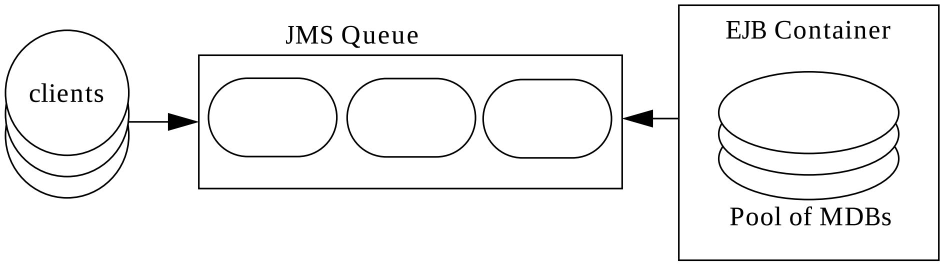
Message-Driven Beans (MDB) provide an easier method to implement asychronous communication than using straight JMS. MDBs were created to receive asynchronous JMS messages. The container handles much of the setup required for JMS queues and topics. It sends all messages to the interested MDB.

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

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

A MDB is similar to a stateless session bean because it does not save conversational state and is used for handling multiple incoming requests. Instead of handling direct requests from a client, MDBs handle requests placed on a queue. Figure 1–4 demonstrates this by showing how clients place requests on a queue. The container takes the requests off of the queue and gives the request to an MDB in its pool.

***Figure 1–4 Message Driven Beans***



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

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

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

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

## Difference Between Session and Entity Beans

The major differences between session and entity beans are that entity beans involve a framework for persistent data management, a persistent identity, and complex business logic. The following table illustrates the different interfaces for session and entity beans. Notice that the difference between the two types of EJBs exists within the bean class and the primary key. All of the persistent data management is done within the bean class methods.

| | Entity Bean | Session Bean |
|---|---|---|
| Remote interface | Extends `javax.ejb.EJBObject` | Extends `javax.ejb.EJBObject` |
| Home interface | Extends `javax.ejb.EJBHome` | Extends `javax.ejb.EJBHome` |
| Bean class | Extends `javax.ejb.EntityBean` | Extends `javax.ejb.SessionBean` |
| Primary key | Used to identify and retrieve specific bean instances | Not used for session beans. Stateful session beans do have an identity, but it is not externalized. |

2

# An EJB Primer For OC4J

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

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

Developing and deploying EJB applications with OC4J includes the following:

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

## Develop EJBs

The development of EJB components for the OC4J environment is identical to development in any other standard J2EE environment. The steps for developing EJBs are as follows:

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

### Create the Development Directory

You can develop your application in any manner. We encourage you to use consistent naming for locating your application easily. One method would be to implement your enterprise Java application under a single parent directory structure, separating each module of the application into their own sub-directories.

Our employee example was developed using the directory structure mentioned in the *OC4J User's Guide*. Notice in Figure 2–1 that the EJB and Web modules exist under the `employee` application parent directory and are developed separately in their own directory.

***Figure 2–1 Employee Directory Structure***

```
.../employee/
    |
    |---------META-INF/
    |          |__________application.xml
    |
    |---------<ejb_module>/
    |          |__________EJB classes (Employee.class, ...)
    |          |__________META-INF/
    |                      |__________ejb-jar.xml
    |
    |---------<web_module>/
               |__________index.html
               |__________JSP pages
               |__________WEB-INF/
                           |__________web.xml
                           |__________classes/
                           |           |__________Servlet classes
                           |                     (EmployeeServlet.class)
                           |__________lib/
                                       |__________ dependent libraries
```

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

## Implement the Enterprise JavaBeans

When you implement an EJB, create the following:

1. A home interface for the bean. The home interface extends `javax.ejb.EJBHome`. It defines the `create` method for your bean. If the bean is an entity bean, it also defines the finder method(s) for that bean.
2. A remote interface for the bean. The remote interface declares the methods that a client can invoke. It extends `javax.ejb.EJBObject`.
3. The bean implementation that includes the following:
   a. the implementation of the business methods that are declared in the remote interface

b. the container callback methods that are inherited from either the `javax.ejb.SessionBean` or `javax.ejb.EntityBean` interfaces

c. the `ejbCreate` method with parameters matching those of the `create` method as defined in the home interface

### Creating the Home Interface

The home interface is used to create and destroy the bean instance; thus, it defines the `create` method for your bean. Each type of EJB can define the `create` method in the following ways:

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

For each `create` method, a corresponding `ejbCreate` method is defined in the bean implementation. The client invokes the `create` method that is declared within the home interface. The container turns around and calls the `ejbCreate` method—with the appropriate parameter signature—within your bean implementation. You can use the parameter arguments to initialize the state of the new EJB object.

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

### Example

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

```
package employee;

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

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

### Creating the Remote Interface

The remote interface defines the business methods of the bean that the client can invoke.

1. The remote interface of the bean must extend the `javax.ejb.EJBObject` interface and its methods must throw the `java.rmi.RemoteException` exception.
2. You must declare the remote interface and its methods as `public`, because clients that invoke these methods are remote.
3. The remote interface, all its method parameters, and return types must be serializable. In general, any object that is passed between the client and the EJB must be serializable, because RMI marshals and unmarshals the object on both ends.
4. Any exception can be thrown to the client, as long as it is serializable. Runtime exceptions, including `EJBException` and `RemoteException`, are transferred back to the client as remote runtime exceptions.

### Example

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

```
package employee;

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

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

  public EmpRecord getEmployee(Integer empNo)
    throws RemoteException;

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

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

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

### Implementing the Bean

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

1. The bean methods defined in the remote interface. The signature for each of these methods must match the signature in the remote interface.

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

2. The methods defined in the home interface are inherited from the `SessionBean` or `EntityBean` interface. The container uses these methods for

controlling the life cycle of the bean. These include the `ejb<Action>` methods, such as `ejbActivate`, `ejbPassivate`, and so on.

3. The `ejbCreate` methods that correspond to the `create` method(s) that are declared in the home interface. The container invokes the appropriate `ejbCreate` method when the client invokes the corresponding `create` method.
4. Any methods that are private to the bean or package used for facilitating the business logic. This includes private methods that your public methods use for completing the tasks requested of them.

### Accessing the Bean

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

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

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

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

**Example** The following example is executed from a servlet, which can also be executed from a JSP or JavaBean, that is co-located in the same container with the stateless session bean. Thus, the JNDI lookup does not require JNDI properties, such as the factory, location, or security parameters.

**Note:** The JNDI name is specified in the `<ejb-ref>` element in the EJB client XML configuration file—in this case, the servlet `web.xml` file—as follows:

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

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

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

  public void init() throws ServletException
  {
    //Retrieve the initial context for JNDI
    Context context = new InitialContext();

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

    //Narrow the returned object to be an EmployeeHome object
    home =
        (EmployeeHome) PortableRemoteObject.narrow(homeObject,
                                                   EmployeeHome.class);

    // Create the remote Employee bean instance and return a reference
    // to the remote interface to this bean.
    empBean =
       (Employee) PortableRemoteObject.narrow(home.create(), Employee.class);
  }
  public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
          throws ServletException, IOException
  {
    response.setContentType("text/html");
    ServletOutputStream out = response.getOutputStream();

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

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

    Iterator iterator = emps.iterator();

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

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

## Create the Deployment Descriptor

After implementing and compiling your classes, you must create the standard J2EE EJB deployment descriptor for all beans in the module. The XML deployment descriptor (defined in the `ejb-jar.xml` file) describes the application components and provides additional information to enable the container to manage the application. The structure for this file is mandated in the DTD file, which is provided at " `http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd`".

The following example shows the sections that are necessary for the `Employee` example.

***Example 2–1 XML Deployment Descriptor for Employee Bean***

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">

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

## Archive the EJB Application

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

**Note:** If you have included a Web application as part of this enterprise Java application, follow the instructions for building the Web application in the *Oracle9iAS Containers for J2EE User's Guide*. Then, modify the `*-web-site.xml` file, and archive all Web application files into a WAR file.

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

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

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

## Prepare the EJB Application for Assembly

Before deploying, perform the following:

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

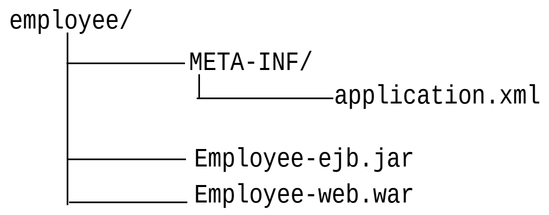
### Modify Application.XML

The `application.xml` file acts as the manifest file for the application and contains a list of the modules that are included within your enterprise application. You use each `<module>` element defined in the `application.xml` file to designate what comprises your enterprise application. Each module describes one of three things: EJB JAR, Web WAR, and any client files. Respectively, modify the `<ejb>`, the `<web>`, and the `<java>` elements in separate `<module>` elements.

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

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

***Figure 2–2 Archive Directory Format***



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

```
<?xml version="1.0"?>
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE
Application 1.2//EN" "http://java.sun.com/j2ee/dtds/application_1_
2.dtd">
<application>
 <module>
  <ejb>Employee-ejb.jar</ejb>
 </module>
 <module>
  <web>
    <web-uri>Employee-web.war</web-uri>
    <context-root>/employee</context-root>
  </web>
 </module>
</application>
```

## Create the EAR File

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

To create the `Employee.EAR` file, execute the following in the employee directory that is shown in Figure 2–2:

```
% jar cvfM Employee.EAR .
```

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

## Deploy the Enterprise Application to OC4J

OC4J is aware of and deploys your application when it is configured within the `server.xml` file. There are three methods to provide application information within the `server.xml` file:

- modify configuration using OEM—this is the recommended procedure
- using `admin.jar` to modify the `server.xml` file
- updating the `server.xml` file manually

Oracle recommends the following:

- Use OEM if you are executing in a production environment. The DCM component of OEM must know about your application configuration.
- Use `admin.jar` or update the XML file manually only if you are executing in standalone mode. If you do update the XML files through either of these methods and your OC4J instance is part of the OEM managed environment, you must notify DCM of the XML changes through the following command:

  ```
  dcmctl updateConfig -ct oc4j
  ```

The sections below describe the OC4J methods. See *Oracle9i Application Server Administrator's Guide* and *Oracle Enterprise Manager Administrator's Guide* for information on OEM.

### Using ADMIN.JAR To Modify SERVER.XML

OC4J contains a command-line deployment tool for deploying J2EE applications—the `admin.jar` command. The options for this command are listed in the *Oracle9iAS Containers for J2EE User's Guide*.

To deploy a J2EE application with the EAR file to a remote node, execute `admin.jar`, as follows:

```
java -jar admin.jar ormi://<host><:port>
      <username> <password> -deploy -file <path/filename>
      -deploymentName <appname> -targetpath <path/destination>
```

where

- The `<host><:port>` is the host and port of the OC4J server.
- The `<username><password>` is the administration username and password for the OC4J server.

- The `-file <path/filename>` indicates the local directory and filename for the EAR file.
- The `-deploymentName <appname>` variable is the name of the application.
- The `-targetpath <path/destination>` indicates what path on the server node to deploy the EAR file into. The default path is the `applications/` directory. Oracle recommends that you provide a target path.

---

**Note:** If you have a Web application within the EAR file, bind the Web application using the `admin.jar -bindWebApp` option.

---

## Updating SERVER.XML Manually

In `server.xml`, add a new or modify the existing `<application name=... path=... auto-start="true" />` entry for each J2EE application. The path should be the full directory path and EAR filename. For our employee example, add the following to the `server.xml` file:

```
<application name="employee"
          path="/private/applications/Employee.EAR"
          auto-start="true" />
```

If you included a Web application portion, you must do the following to bind the Web application to the Web server. In `*-web-site.xml`, add a <`web-app` ...> entry for each Web application. The `<application>` variable should be the same value as provided in the `server.xml` file. The `<name>` should be the WAR file, without the WAR extension, for the Web application.

For Web application binding for the `employee` Web application, add the following:

```
<web-app application="employee" name="Employee-web"
    root="/employee" />
```

## Verifying Deployment

OC4J detects the addition of your application to `server.xml`. The OC4J server displays a message that your application has been deployed. This is the extent of installation in OC4J.

If the server does not notice your application in a timely manner, simply start (or restart) OC4J, and it will locate your application immediately.

If you modified `server.xml` using `admin.jar` or manual edit, and you are not executing OC4J in standalone mode, notify the DCM component within OEM of your XML file changes by executing the following:

```
dcmctl updateConfig -ct oc4j
```

For more information on DCM, see the DCM Appendix in the *Oracle9i Application Server Administrator's Guide.*

3

# CMP Entity Beans

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

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

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

This chapter demonstrates the following:

- Creating Entity Beans—Demonstrates how to create a simple container-managed persistent entity bean.
- Advanced CMP Entity Beans—Demonstrates advanced configuration for finder methods, object-relational mapping, and so on.

See Chapter 4, "BMP Entity Beans", for an example of how to create a simple bean-managed persistent entity bean.

## Creating Entity Beans

To create an entity bean, perform the following steps:

1. Create a remote interface for the bean. The remote interface declares the methods that a client can invoke. It must extend `javax.ejb.EJBObject`.
2. Create a home interface for the bean. The home interface must extend `javax.ejb.EJBHome`. It defines the `create` and finder methods, including `findByPrimaryKey`, for your bean.
3. Define the primary key for the bean. The primary key identifies each entity bean instance. The primary key must be either a well-known class, such as `java.lang.String`, or defined within its own class.
4. Implement the bean. This includes the following:
    a. The implementation for the methods that are declared in your remote interface.
    b. The methods that are defined in the `javax.ejb.EntityBean` interface.
    c. The methods that match the methods that are declared in your home interface. This includes the following:
        * the `ejbCreate` and `ejbPostCreate` methods with parameters matching the associated `create` method defined in the home interface
        * an `ejbFindByPrimary` key method, which corresponds to the `findByPrimaryKey` method of the home interface
        * any other finder methods that were defined in the home interface
5. Create the bean deployment descriptor. The deployment descriptor specifies properties for the bean through XML elements. This step is where you identify the data within the bean that is to be managed by the container.
6. If the persistent data is saved to or restored from a database and you are not using the defaults provided by the container, then you must ensure that the correct tables exist for the bean. In the extreme default scenario, the container will actually create the table and columns for your data based on deployment descriptor and datasource information.
7. Create an EJB JAR file containing the bean, the remote and home interfaces, and the deployment descriptor. Once created, configure the `application.xml` file, create an EAR file, and install the EJB in OC4J.

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

## Home Interface

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

#### *Example 3–1 Entity Bean Employee Home Interface*

To demonstrate an entity bean, this example creates a bean that manages a purchase order. The entity bean contains a list of items that were ordered by the customer.

The home interface extends `javax.ejb.EJBHome` and defines the `create` and `findByPrimaryKey` methods.

```
package employee;

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

public interface EmployeeHome extends EJBHome
{

  public Employee create(Integer empNo)
    throws CreateException, RemoteException;

  // Find an existing employee
  public Employee findByPrimaryKey (Integer empNo)
    throws FinderException, RemoteException;

  //Find all employees
  public Collection findAll()
    throws FinderException, RemoteException;
}
```

## Remote Interface

The entity bean remote interface is the interface that the customer sees and invokes methods upon. It extends `javax.ejb.EJBObject` and defines the business logic methods. For our employee entity bean, the remote interface contains methods for adding and removing employees, and retrieving and setting employee information.

```
package employee;

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

public interface Employee extends EJBObject
{
  // getter remote methods
  public Integer getEmpNo() throws RemoteException;
  public String getEmpName() throws RemoteException;
  public Float getSalary() throws RemoteException;

  // setter remote methods
  public void setEmpName(String newEmpName) throws RemoteException;
  public void setSalary(Float newSalary) throws RemoteException;
}
```

## Entity Bean Class

The entity bean class must implement the following methods:

- the target methods for the methods that are declared in the home interface, which includes the `ejbCreate` method and any finder methods, including `ejbFindByPrimaryKey`
- the business logic methods that are declared in the remote interface
- the methods that are inherited from the `EntityBean` interface

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

```
package employee;

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

public class EmployeeBean extends Object implements EntityBean
{
```

```
public Integer empNo;
public String empName;
public Float salary;
public EntityContext entityContext;

public EmployeeBean()
{
  // Constructor. Do not initialize anything in this method.
  // All initialization should be performed in the ejbCreate method.
}

public Integer getEmpNo()
{
  return empNo;
}

public String getEmpName()
{
  return empName;
}

public Float getSalary()
{
  return salary;
}

public void setEmpName(String empName)
{
  this.empName = empName;
}

public void setSalary(Float salary) {
  this.salary = salary;
}

public Integer ejbCreate(Integer empNo)
  throws CreateException, RemoteException
{
  this.empNo = empNo;
  return empNo;
}

public void ejbPostCreate(Integer empNo)
  throws CreateException, RemoteException
```

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

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

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

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

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

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

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

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

## Persistent Data

In CMP entity beans, you define the persistent data both in the bean instance and in the deployment descriptor. The declaration of the data fields in the bean instance creates the resources for the fields. The deployment descriptor defines these fields as persistent.

In our employee example, the data fields are defined in the bean instance, as follows:

```
public Integer empNo;
public String empName;
public Float salary;
```

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

```
<enterprise-beans>
      <entity>
         <display-name>Employee</display-name>
         <ejb-name>EmployeeBean</ejb-name>
         <home>employee.EmployeeHome</home>
         <remote>employee.Employee</remote>
         <ejb-class>employee.EmployeeBean</ejb-class>
         <persistence-type>Container</persistence-type>
         <prim-key-class>java.lang.Integer</prim-key-class>
         <reentrant>False</reentrant>
         <cmp-field><field-name>empNo</field-name></cmp-field>
         <cmp-field><field-name>empName</field-name></cmp-field>
         <cmp-field><field-name>salary</field-name></cmp-field>
         <primkey-field>empNo</primkey-field>
      </entity>
...
</enterprise-beans>
```

In most cases, you map the persistent data fields to columns in a table that exists in a designated database. However, you can accept the defaults for these fields—thus, avoiding more deployment descriptor configuration.

OC4J contains some defaults for mapping these fields to a database and its table.

- Database—The default database is the first database that is defined in the `data-sources.xml` file. The JNDI name that is used for a pooled database is defined in the `<ejb-location>` element.

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

---

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

---

To customize the default database, change the first configured database (including its `<ejb-location>`) within the `data-sources.xml` to point to your database.

- Table with correct column names—The container creates a default table with the same name as the bean name (defined in `<ejb-name>`), with columns having the same name as the `<cmp-field>` elements in the designated database. The data types for the database, translating Java data types to database data types, are defined in the specific database XML file, such as `oracle.xml`.

If you want to designate another database or generate a table that has a different naming convention, see "Object-Relational Mapping of Persistent Fields" on page 3-13 for a description of how to customize your database, table, and column names.

## Primary Key

Each entity bean instance has a primary key that uniquely identifies it from other instances. You must declare the primary key (or the fields contained within a complex primary key) as a container-managed persistent field in the deployment descriptor. All fields within the primary key are restricted to either primitive, serializable, or types that can be mapped to SQL types. You can define your primary key in one of two ways:

- Define the type of the primary key to be a well-known type. The type is defined in the `<prim-key-class>` in the deployment descriptor. The data field that is identified as the persistent primary key is identified in the `<primkey-field>` element in the deployment descriptor. The primary key variable that is declared within the bean class must be declared as `public`.
- Define the type of the primary key as a serializable object within a <*`name`*>`PK` class that is serializable. This class is declared in the `<prim-key-class>` element in the deployment descriptor. This is an advanced method for defining

a primary key, so it is discussed in "Defining the Primary Key in a Class" on page 3-9.

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

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

```
<enterprise-beans>
      <entity>
         <display-name>Employee</display-name>
         <ejb-name>EmployeeBean</ejb-name>
         <home>employee.EmployeeHome</home>
         <remote>employee.Employee</remote>
         <ejb-class>employee.EmployeeBean</ejb-class>
         <persistence-type>Container</persistence-type>
         <prim-key-class>java.lang.Integer</prim-key-class>
         <reentrant>False</reentrant>
         <cmp-field><field-name>empNo</field-name></cmp-field>
         <cmp-field><field-name>empName</field-name></cmp-field>
         <cmp-field><field-name>salary</field-name></cmp-field>
         <primkey-field>empNo</primkey-field>
      </entity>
...
</enterprise-beans>
```

## Defining the Primary Key in a Class

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

The primary key variables must adhere to the following:

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

Within the primary key class, you implement a constructor for creating a primary key instance. Once defined in this manner, the container manages the primary key, as well as storing the persistent data.

The following example is a complex primary key made up of employee number and country code. Our company is so large that it reuses employee numbers in different countries. Thus, the combination of both the employee number and the country code uniquely identifies each employee.

```
package employee;

public class EmpPK implements java.io.Serializable
{
  public Integer empNo;
  public String countryCode;

  //constructor
  public EmpPK ( ) { }
}
```

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

```
<enterprise-beans>
      <entity>
         <display-name>Employee</display-name>
         <ejb-name>EmployeeBean</ejb-name>
         <home>employee.EmployeeHome</home>
         <remote>employee.Employee</remote>
         <ejb-class>employee.EmployeeBean</ejb-class>
         <persistence-type>Container</persistence-type>
         <prim-key-class>employee.EmpPK</prim-key-class>
         <reentrant>False</reentrant>
         <cmp-field><field-name>empNo</field-name></cmp-field>
         <cmp-field><field-name>countryCode</field-name></cmp-field>
      </entity>
      ...
</enterprise-beans>
```

## Deploying the Entity Bean

Archive your EJB into a JAR file. You deploy the entity bean in the same way as the session bean, which "Prepare the EJB Application for Assembly" on page 2-11 and "Deploy the Enterprise Application to OC4J" on page 2-13 explain in detail.

## Advanced CMP Entity Beans

This section discusses how to implement your bean beyond the simple CMP entity bean. It includes the following sections:

- Advanced Finder Methods
- Object-Relational Mapping of Persistent Fields

### Advanced Finder Methods

Specifying the `findByPrimaryKey` method is easy to do in OC4J. All the fields for defining a simple or complex primary key are specified within the `ejb-jar.xml` deployment descriptor. However, if you want to define other finder methods in a CMP entity bean, you must do the following:

1. Add the finder method to the home interface.
2. Add the finder method definition to the OC4J-specific deployment descriptor—the `orion-ejb-jar.xml` file.

#### Add the Finder Method to Home Interface

You must first add the finder method to the home interface. For example, with the employee entity bean, if we wanted to retrieve all employees, the `findAll` method would be defined within the home interface, as follows:

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

#### Add the Finder Method Definition to OC4J-Specific Deployment Descriptor

After specifying the finder method in the home interface, modify the `orion-ejb-jar.xml` file with the finder method specifics. The container identifies the correct query necessary for retrieving the required fields.

The `<finder-method>` element defines all finder methods—excluding the `findByPrimaryKey` method. The simplest finder method to define is the `findByAll` method. The `query` attribute in the `<finder-method>` element specifies the `WHERE` clause for the query. If you want all rows retrieved, then an empty query (`query=""`) returns all records.

The following example retrieves all records from the `EmployeeBean`. The method name is `findAll`, and it requires no parameters because it returns a `Collection` of all employees.

```
<finder-method query="">
```

```
  <method>
     <ejb-name>EmployeeBean</ejb-name>
     <method-name>findAll</method-name>
         <method-params></method-params>
     </method>
</finder-method>
```

After deploying the application with this bean, OC4J adds the following statement of what query it invokes as a comment in the finder method definition:

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

Verify that it is the type of query that you expect.

To be more specific, modify the `query` attribute with the appropriate `WHERE` clause. This clause refers to passed in parameters using the '$' symbol: the first parameter is denoted by $1, the second by $2. All `<cmp-field>` elements that are used within the `WHERE` clause are denoted by `$<cmp-field>` name.

The following example specifies a `findByName` method (which should be defined in the home interface) where the name of the employee is given as in the method parameter, which is substituted for the $1. It is matched to the CMP name, "`empName`". Thus, our `query` attribute is modified to contain the following for the `WHERE` clause: "`$empname=$1`".

```
<finder-method query="$empname = $1">
  <method>
    <ejb-name>EmployeeBean</ejb-name>
    <method-name>findByName</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
    </method-params>
  </method>
</finder-method>
```

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

---

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

---

If you wanted to specify a full query and not just the section after the WHERE clause, specify the `partial` attribute to FALSE and then define the full query in the `query` attribute. The default value for `partial` is true, which is why it is not specified on the previous finder-method example.

```
<finder-method partial="false"
               query="select * from EMP where $empName = $1">
        <!-- Generated SQL: "select * from EMP where EMP.ENAME = ?" -->
        <method>
                <ejb-name>EmployeeBean</ejb-name>
                <method-name>findByName</method-name>
                <method-params>
                        <method-param>java.lang.String</method-param>
                </method-params>
        </method>
</finder-method>
```

Specifying the full SQL query is useful for complex SQL statements.

## Object-Relational Mapping of Persistent Fields

As "Persistent Data" on page 3-7 discusses, your persistent data can be automatically mapped to a database table by the container. However, if the data represented by your bean is more complex or you do not want to accept the defaults that OC4J provides for you, then map the CMP designated fields to an existing database table and its applicable rows within the `orion-ejb-jar.xml` file. Once mapped, the container provides the persistence storage of the CMP data to the indicated table and rows.

Before configuring the object-relational mapping, add the `DataSource` used for the destination within the `<resource-ref>` element in the `ejb-jar.xml` file.

### Mapping CMP Fields to a Database Table and Its Columns

Configure the following within the `orion-ejb-jar.xml` file:

1. Configure the `<entity-deployment>` element for every entity bean that contains CMP fields that will be mapped within it.
2. Configure a `<cmp-field-mapping>` element for every field within the bean that is mapped. Each `<cmp-field-mapping>` element must contain the name of the field to be persisted.
    a. Configure the primary key in the `<primkey-mapping>` element contained within its own `<cmp-field-mapping>` element.
    b. Configure simple data types (such as a primitive, simple object, or serializable object) that are mapped to a single field within a single `<cmp-field-mapping>` element. The name and database field are fully defined within the element attributes.
    c. Configure complex data types using one of the many sub-elements of the `<cmp-field-mapping>` element. These can be one of the following:
        * If you define an object as your complex data type, then specify each field or property within the object in the `<fields>` or `<properties>` element.
        * If you specify a field defined in another entity bean, then define the home interface of this entity bean in the `<entity-ref>` element.
        * If you define a `List`, `Collection`, `Set`, or `Map` of fields, then define these fields within the `<list-mapping>`, `<collection-mapping>`, `<set-mapping>`, `<map-mapping>` elements.

Examples for simple and complex O-R mappings are shown in the following sections:

- One-to-One Mapping Example
- One-to-Many Mapping Example

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

- The bean is identified in the `<entity-deployment>` `name` attribute. The JNDI name for this bean is defined in the `location` attribute.
- The database table name is defined in the `table` attribute. And the database is specified in the `data-source` attribute, which should be identical to the `<ejb-location>` name of a `DataSource` defined in the `data-sources.xml` file.

- The bean primary key, empNo, is mapped to the database table column, EMPNO, within the <primkey-mapping> element.
- The bean persistent data fields, empName and salary, are mapped to the database table columns ENAME and SAL within the <cmp-field-mapping> element.

```
<entity-deployment name="EmpBean" location="emp/EmpBean"
  wrapper="EmpHome_EntityHomeWrapper2" max-tx-retries="3"
  table="emp" data-source="jdbc/OracleDS">
  <primkey-mapping>
    <cmp-field-mapping name="empNo" persistence-name="empno" />
  </primkey-mapping>
  <cmp-field-mapping name="empName" persistence-name="ename" />
  <cmp-field-mapping name="salary" persistence-name="sal" />
  ...
</entity-deployment>
```

After deployment, OC4J maps this to the following:

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

**One-to-Many Mapping Example** If you have two beans that access two tables for their data, you must map the persistent data from both beans to the respective tables.

We added a department number to our employee example. Each employee belongs to a department; each department has multiple employees. The container will handle this object-relational mapping; however, you must specify how the data is stored.

The employee data maps to the employee database table; the department data is mapped to the database department table. The employee database table also contains a foreign key of the department number to link this information together.

The XML configuration for the employee bean, EmpBean, is as follows:

- Same XML configuration details for the employee bean as stated above, with the addition of the definition of the department number as part of the employee entity bean.
- The department number is defined in the bean instance as `deptno`, which relates to the department number defined in the `DeptBean`. Thus, the `DeptBean`, its `deptno` field, and its mapping to the database column, `deptno`, is configured within a `<cmp-field-mapping><entity-ref>` element, as follows:

```
<entity-deployment name="EmpBean" location="emp/EmpBean"
wrapper="EmpHome_EntityHomeWrapper2" max-tx-retries="3" table="emp"
data-source="jdbc/OracleDS">
   <primkey-mapping>
         <cmp-field-mapping name="empNo" persistence-name="empno" />
   </primkey-mapping>
   <cmp-field-mapping name="empName" persistence-name="ename" />
   <cmp-field-mapping name="salary" persistence-name="sal" />
   <cmp-field-mapping name="dept">
       <entity-ref home="dept/DeptBean">
               <cmp-field-mapping name="dept" persistence-name="deptno" />
        </entity-ref>
   </cmp-field-mapping>
   <finder-method query="">
    <!-- Generated SQL: "select EMP.empno, EMP.ename, EMP.sal,
        EMP.deptno from EMP" -->
      <method>
       <ejb-name>EmpBean</ejb-name>
        <method-name>findAll</method-name>
        <method-params></method-params>
      </method>
   </finder-method>
</entity-deployment>
```

**Note:** This definition within the `EmpBean` configuration refers to the definition of the `deptno` within the `DeptBean` configuration.

The XML configuration for the department bean, `DeptBean`, is as follows:

- The bean is identified in the `<entity-deployment>` `name` attribute. The JNDI name for this bean is defined in the `location` attribute.
- The database table name is defined in the `table` attribute. And the database is specified in the `data-source` attribute, which should be identical to the

`<ejb-location>` name of a `DataSource` defined in the `data-sources.xml` file.

- The bean primary key, `deptNo`, is mapped to the `dept` database table in its `DEPTNO` column within the `<primkey-mapping>` element.
- The bean persistent data field, `deptName`, is mapped to the `DEPT` database table in its `DNAME` column within a `<cmp-field-mapping>` element.
- The bean persistent data field, `employees`, is actually a bean—the employee bean. Thus, the example uses the `<collection-mapping>` element to specify all fields within the employee bean. A `Collection` containing the employee information is returned. See the bold text in the example below.
  - The employees field maps to the `EmpBean` entity bean. Its home interface reference is defined in the `home` attribute of the `<entity-ref>` element.
  - The primary key used to retrieve the employees is defined as `deptNo` within the `<primkey-mapping>` element and is mapped to the database column `DEPTNO`.
  - All fields that are of interest to the department bean are defined within `<cmp-field-mapping>` elements. The bean instance fields within the `EmpBean` that are of interest are `empNo`, `empName`, and `salary`. Their respective database columns are also specified: `EMPNO`, `ENAME`, and `SAL.` The database table itself is defined in the `EmpBean` `<entity-deployment>` definition.

```
<entity-deployment name="DeptBean" location="dept/DeptBean"
wrapper="DeptHome_EntityHomeWrapper2" max-tx-retries="3" table="dept"
data-source="jdbc/OracleDS">
   <primkey-mapping>
        <cmp-field-mapping name="deptNo" persistence-name="deptno" />
   </primkey-mapping>
   <cmp-field-mapping name="deptName" persistence-name="dname" />
   <cmp-field-mapping name="employees">
   <collection-mapping table="emp">
       <primkey-mapping>
           <cmp-field-mapping name="deptNo" persistence-name="deptno" />
       </primkey-mapping>
       <value-mapping type="emp.Emp">
           <cmp-field-mapping>
              <entity-ref home="emp/EmpBean">
               <cmp-field-mapping name="empNo" persistence-name="empno"/>
               <cmp-field-mapping name="empName" persistence-name="ename"/>
               <cmp-field-mapping name="salary" persistence-name="sal"/>
```

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

# 4 BMP Entity Beans

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

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

The following sections discuss how to implement data persistence:

- Creating BMP Entity Beans
- Remote and Home Interface
- BMP Entity Bean Implementation
- Create Database Table and Columns for Entity Data

## Creating BMP Entity Beans

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

1. Create a remote interface for the bean. The remote interface declares the methods that a client can invoke. It must extend `javax.ejb.EJBObject`.
2. Create a home interface for the bean. The home interface must extend `javax.ejb.EJBHome`. It defines the `create` and finder methods, including `findByPrimaryKey`, for your bean.
3. Define the primary key for the bean. The primary key identifies each entity bean instance. The primary key must be either a well-known class, such as `java.lang.String`, or defined within its own class.
4. Implement the bean. This includes the following:
   a. The implementation for the methods declared in your remote interface.
   b. The methods that match the methods that are declared in your home interface. This includes the following:
      * The `ejbCreate`, which must create the persistent data, and `ejbPostCreate` methods with parameters matching each of the `create` methods defined in the home interface.
      * An `ejbFindByPrimary` key method, which corresponds to the `findByPrimaryKey` method of the home interface, retrieves the primary key and validates that it exists.
      * Any other finder methods that were defined in the home interface.
   c. The methods defined in the `javax.ejb.EntityBean` interface. The `ejbCreate`, `ejbPostCreate`, and `ejbFindByPrimaryKey` are already mentioned above. The other methods are as follows:
      * Persistent saving of the data within the `ejbStore` method.
      * Restoring the persistent data to the bean within your implementation of the `ejbLoad` method.
      * Passivation of the bean instance within the `ejbPassivate` method.
      * Activation of the passivated bean instance within the `ejbActivate` method.
5. If the persistent data is saved to or restored from a database, you must ensure that the correct tables exist for the bean.

6. Create the bean deployment descriptor. The deployment descriptor specifies properties for the bean through XML properties.
7. Create an EJB JAR file containing the bean, the remote and home interfaces, and the deployment descriptor. Once created, configure `application.xml`, create an EAR file, and install the EJB in OC4J.

## Remote and Home Interface

The BMP entity bean definition of the remote and home interfaces are identical to the CMP entity bean. For examples of how the remote and home interface are implemented, see "Creating Entity Beans" on page 3-2.

## BMP Entity Bean Implementation

Because the container is no longer managing the primary key nor the saving of the persistent data, the bean callback functions must include the implementation logic for these functions. The container invokes the `ejbCreate`, `ejbFindByPrimaryKey`, other finder methods, `ejbStore`, and `ejbLoad` methods when it is appropriate.

### The ejbCreate Implementation

The `ejbCreate` method is responsible primarily for the creation of the primary key. This includes creating the primary key, creating the persistent data representation for the key, initializing the key to a unique value, and returning this key to the container. The container maps the key to the entity bean reference.

The following example shows the `ejbCreate` method for the employee example, which initializes the primary key, `empNo`. It should automatically generate a primary key that is the next available number in the employee number sequence. However, for this example to be simple, the `ejbCreate` method requires that the user provide the unique employee number.

In addition, because the full data for the employee is provided within this method, the data is saved within the context variables of this instance. After initialization, it returns this key to the container.

```
// The create methods takes care of generating a new empNo and returns
// its primary key to the container
public Integer ejbCreate (Integer empNo, String empName, Float salary)
          throws CreateException, RemoteException
{
 this.empNo = empNo;
 this.empName = empName;
 this.salary = salary;
 return (empNo);
}
```

The deployment descriptor defines only the primary key class in the `<prim-key-class>` element. Because the bean is saving the data, there is no definition of persistence data in the deployment descriptor. Note that the deployment descriptor does define the database the bean uses in the `<resource-ref>` element. For more information on database configuration, see "Modify XML Deployment Descriptors" on page 4-10.

```
<enterprise-beans>
   <entity>
      <display-name>EmployeeBean</display-name>
      <ejb-name>EmployeeBean</ejb-name>
      <home>employee.EmployeeHome</home>
      <remote>employee.Employee</remote>
      <ejb-class>employee.EmployeeBean</ejb-class>
      <persistence-type>Bean</persistence-type>
      <prim-key-class>java.lang.Integer</prim-key-class>
      <reentrant>False</reentrant>
      <resource-ref>
            <res-ref-name>jdbc/OracleDS</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Application</res-auth>
      </resource-ref>
   </entity>
</enterprise-beans>
```

Alternatively, you can create a complex primary key based on several data types. You define a complex primary key within its own class, as follows:

```
package employee;

public class EmployeePK implements java.io.Serializable
{
  public Integer empNo;
  public String empName;
  public Float salary;

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

  public EmployeePK(Integer empNo, String empName, Float salary)
  {
    this.empNo = empNo;
    this.empName = empName;
    this.salary = salary;
  }

}
```

For a primary key class, you define the class in the `<prim-key-class>` element, which is the same for the simple primary key definition.

```
<enterprise-beans>
   <entity>
      <display-name>EmployeeBean</display-name>
      <ejb-name>EmployeeBean</ejb-name>
      <home>employee.EmployeeHome</home>
      <remote>employee.Employee</remote>
      <ejb-class>employee.EmployeeBean</ejb-class>
      <persistence-type>Bean</persistence-type>
      <prim-key-class>employee.EmployeePK</prim-key-class>
      <reentrant>False</reentrant>
      <resource-ref>
            <res-ref-name>jdbc/OracleDS</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Application</res-auth>
      </resource-ref>
   </entity>
</enterprise-beans>
```

The employee example requires that the employee number is given to the bean by the user. Another method would be to generate the employee number by computing the next available employee number, and use this in combination with the employee's name and office location.

After defining the complex primary key class, you would create your primary key within the `ejbCreate` method, as follows:

```
public EmployeePK ejbCreate(Integer empNo, String empName, Float salary)
    throws CreateException, RemoteException
{
  pk = new EmployeePK(empNo, empName, salary);
  ...
}
```

The other task that the `ejbCreate` (or `ejbPostCreate`) should handle is allocating any resources necessary for the life of the bean. For this example, because we already have the information for the employee, the `ejbCreate` performs the following:

1. Retrieves a connection to the database. This connection remains open for the life of the bean. It is used to update employee information within the database. It should be released in `ejbPassivate` and `ejbRemove`, and reallocated in `ejbActivate`.
2. Updates the database with the employee information.

This is executed, as follows:

```
public EmployeePK ejbCreate(Integer empNo, String empName, Float salary)
    throws CreateException, RemoteException
{
  pk = new EmployeePK(empNo, empName, salary);
  conn = getConnection(dsName);
  ps = conn.prepareStatement(INSERT INTO EMPLOYEEBEAN (EmpNo, EmpName, SAL)
      VALUES ( this.empNo.intValue(), this.empName, this.salary.floatValue());
  ps.close();
  return pk;
}
```

## The ejbFindByPrimaryKey Implementation

The `ejbFindByPrimaryKey` implementation is a requirement for all BMP entity beans. Its primary responsibility is to ensure that the primary key is valid. Once it is validated, it returns the primary key to the container, which uses the key to return the remote interface reference to the user.

This sample verifies that the employee number is valid and returns the primary key, which is the employee number, to the container. A more complex verification would be necessary if the primary key was a class.

```
public Integer ejbFindByPrimaryKey(Integer empNoPK)
    throws FinderException, RemoteException
{
  if (empNoPK == null) {
      throw new FinderException("Primary key cannot be null");
  }
  ps = conn.prepareStatement(SELECT EMPNO FROM EMPLOYEEBEAN
              WHERE EMPNO = ?);
  ps.setInt(1, empNoPK.intValue());
  ps.executeQuery();
  ResultSet rs = ps.getResultSet();
  if (rs.next()) {
   /*PK is validated because it exists already*/
  } else {
  throw new FinderException("Failed to select this PK");
 }

 ps.close();

 return empNoPK;
}
```

## Other Finder Methods

You can create other finder methods beyond the single `ejbFindByPrimaryKey`.

To create other finder methods, do the following:

1. Add the finder method to the home interface.
2. Implement the finder method in the BMP bean implementation.

These finder methods need only to gather the primary keys for all of the entity beans that should be returned to the user. The container maps the primary keys to references to each entity bean within either a `Collection` (if multiple references are returned) or to the single class type.

The following example shows the implementation of a finder method that returns all employee records.

```
public Collection ejbFindAll() throws FinderException, RemoteException
{
  Vector recs = new Vector();

  ps = conn.prepareStatement(SELECT EMPNO FROM EMPLOYEEBEAN);
  ps.executeQuery();
  ResultSet rs = ps.getResultSet();

  int i = 0;

  while (rs.next())
  {
   retEmpNo = new Integer(rs.getInt(1));
   recs.add(retEmpNo);
  }

 ps.close();
 return recs;
}
```

## The ejbStore Implementation

The container invokes the `ejbStore` method when the persistent data should be saved to the database. This includes whenever the primary key is "dirtied", or before the container passivates the bean instance or removes the instance. The BMP bean is responsible for ensuring that all data is stored to some resource, such as a database, within this method.

```
public void ejbStore() throws RemoteException
{
 //Container invokes this method to instruct the instance to
 //synchronize its state by storing it to the underlying database
 ps = conn.prepareStatement(UPDATE EMPLOYEEBEAN SET EMPNAME=?,
              SALARY=? WHERE EMPNO=?);
 ps.setString(1, this.empName);
 ps.setFloat(2, this.salary.floatValue());
 ps.setInt(3, this.empNo.intValue());
 if (ps.executeUpdate() != 1) {
        throw new RemoteException("Failed to update record");
 }
 ps.close();
}
```

## The ejbLoad Implementation

The container invokes the `ejbLoad` method after activating the bean instance. The purpose of this method is to repopulate the persistent data with the saved state. For most `ejbLoad` methods, this implies reading the data from a database into the instance data variables.

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

## The ejbPassivate Implementation

The `ejbPassivate` method is invoked directly before the bean instance is serialized for future use. Normally, this is invoked when the instance has not been used in a while. It will be re-activated, through the `ejbActivate` method, the next time the user invokes a method on this instance.

Before the bean is passivated, you should release all resources and release any static information that would be too large to be serialized. Any large, static information that can be easily regenerated within the `ejbActivate` method should be released in this method.

In our example, the only resource that cannot be serialized is the open database connection. It is closed in this method and reopened in the `ejbActivate` method.

```
public void ejbPassivate()
{
 // Container invokes this method on an instance before the instance
 // becomes disassociated with a specific EJB object
 conn.close();
}
```

## The ejbActivate Implementation

As the `ejbPassivate` method section states, the container invokes this method when the bean instance is reactivated. That is, the user has asked to invoke a method on this instance. This method is used to open resources and rebuild static information that was released in the `ejbPassivate` method.

Our employee example opens the database connection where the employee information is stored.

```
public void ejbActivate() throws RemoteException
{
 // Container invokes this method when the instance is taken out
 // of the pool of available instances to become associated with
 // a specific EJB object
 conn = getConnection(dsName);
}
```

### The ejbRemove Implementation

The container invokes the `ejbRemove` method before removing the bean instance itself or by placing the instance back into the bean pool. This means that the information that was represented by this entity bean should be removed—both by the instance being destroyed and removed from within persistent storage. The employee example removes the employee and all associated information from the database before the instance is destroyed. Close the database connection.

```
public void ejbRemove() throws RemoveException, RemoteException
{
 //Container invokes this method befor it removes the EJB object
 //that is currently associated with the instance
 ps = conn.prepareStatement(DELETE FROM EMPLOYEEBEAN WHERE EMPNO=?);
 ps.setInt(1, this.empNo.intValue());
 if (ps.executeUpdate() != 1) {
        throw new RemoteException("Failed to delete record");
 }
 ps.close();
 conn.close();
}
```

## Modify XML Deployment Descriptors

In addition to the configuration described in "Creating Entity Beans" on page 3-2, you must modify and add the following to your `ejb-jar.xml` deployment descriptor:

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

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

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

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

```
<data-source
  class="com.evermind.sql.DriverManagerDataSource"
  name="Oracle"
  location="jdbc/OracleCoreDS"
  pooled-location="jdbc/pool/OraclePoolDS"
  ejb-location="jdbc/OracleDS"
  xa-location="jdbc/xa/OracleXADS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  url="jdbc:oracle:thin:@localhost:5521:orcl"
  username="scott"
  password="tiger"
  max-connections="300"
  min-connections="5"
  max-connect-attempts="10"
  connection-retry-interval="1"
  inactivity-timeout="30"
  wait-timeout="30"
/>
```

## Create Database Table and Columns for Entity Data

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

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

| Table | Columns |
|---|---|
| EMPLOYEEBEAN | ■ employee number (EMPNO)<br>■ employee name (EMPNAME)<br>■ salary (SALARY) |

The following shows the SQL commands that create these fields.

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

# 5 Message-Driven Beans

A Message-Driven Bean (MDB) is a JMS message listener that can reliably consume messages from a queue or a durable subscription. The advantage of using an MDB instead of a JMS message listener is because you can use the asynchronous nature of a JMS listener with the advantages that the EJB container does the following for you:

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

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

This chapter discusses the tasks involved in creating an MDB in OC4J and demonstrates simple MDB development with a basic configuration and deployment. Download the `MessageLogger` MDB example (`messagelogger.ear`) from the OC4J sample code page at `http://otn.oracle.com/sample_code/tech/java/oc4j/htdocs/oc4jsamplecode/oc4j-demo-ejb.html` on the OTN site.

## Creating Message Driven Beans

To create an MDB, you perform the following steps:

1. Implement the bean. This includes the following:

   a. The bean class must implement the `javax.ejb.MessageDrivenBean` and `javax.jms.MessageListener` interfaces, which includes the following:

      * the `onMessage` method in the `MessageListener` interface
      * the `setMessageDrivenContext` method in the `MessageDrivenBean` interface

   b. Container callback methods that normally match methods in the EJB home interface. A remote and home interface are not implemented with an MDB. However, some of the callback methods required for these interfaces are implemented in the bean implementation. These include the following:

      * an `ejbCreate` method
      * an `ejbRemove` method

2. Create the MDB deployment descriptors.
3. Configure the JMS `Destination` type (queue or topic) in the OC4J JMS XML file—`jms.xml`.
4. Map the JMS `Destination` type to the MDB in the OC4J-specific deployment descriptor—`orion-ejb-jar.xml`.
5. If you involve a database in your MDB application, configure the DataSource that represents your database in `data-sources.xml`.
6. Create an EJB JAR file containing the bean and the deployment descriptor. Once created, configure the `application.xml` file, create an EAR file, and install the EJB in OC4J.

The following sections demonstrates a simple MDB.

- Bean Class Implementation
- MDB Deployment Descriptor
- JMS XML Configuration
- DataSource XML Configuration
- Deploying the Entity Bean

## Bean Class Implementation

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

The following example is a `MessageLogger` MDB prints the message it receives, and invokes an entity bean, `LogMessage`, to process the message.

As an MDB, it is responsible for the following:

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

```
package com.evermind.logger;

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

public class MessageLogger implements MessageDrivenBean
{
 private MessageDrivenContext messageContext;

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

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

 public void onMessage(Message message)
 {
  /* The whole point for this message logger MDB is to receive and print
     messages. It is not complicated, but it shows how MDBs are set up to
```

```
      receive JMS messages from queues and topics. */

   /*print the message*/
   System.out.println("Received message: " + message);

   try
   {
    /* retrieve the initial context for the lookup */
    Context ic =  new InitialContext();

    /*invoke the LogMessage entity bean to process the message*/
    /* retrieve the home interface of the LogMessage bean*/
    /* making sure to narrow the returned object to LogMessageHome*/
    LogMessageHome home = (LogMessageHome)
       javax.rmi.PortableRemoteObject.narrow(
                  ic.lookup("java:comp/env/logMessages"),
                  LogMessageHome.class);

    /* Retrieve the remote interface and instantiate the bean through
       the home.create method. The LogMessage create method requires
       three parameters: the time of the message, the subject of the
       message, and the actual message */
    home.create(
        new Date(message.getLongProperty("time")),
        message.getStringProperty("subject"),
       message.getStringProperty("message"));
   }
    catch(Exception e)
   {
    throw new EJBException(e);
   }
  }

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

## Configuring XML Files

You need to create both of the MDB deployment descriptors and the JMS `Destination` configuration.

- The JMS Destination configuration defines the queue or topic—The queue or topic itself is configured in the `jms.xml` file.
- The EJB deployment descriptor defines the MDB—The deployment descriptor (`ejb-jar.xml`) specifies whether a queue or a topic is used.
- The OC4J-specific deployment descriptor informs the container which JMS `Destination` to associate with the MDB—The container understands which JMS `Destination` to associate the MDB by the definition in the OC4J-specific deployment descriptor (`orion-ejb-jar.xml`).

## MDB Deployment Descriptor

Within the EJB deployment descriptor, define the MDB name, class, JNDI reference, and JMS `Destination` type (queue or topic) in the `<message-driven>` element. If a queue is specified, you must also define whether it is durable or not.

The following is the deployment descriptor for the entire EJB application. It includes the deployment information for both the `MessageLogger` MDB and the `LogMessage` entity bean. The `MessageLogger` MDB invokes the `LogMessage` entity bean.

The `MessageLogger` MDB is defined in the `<message-driven>` element, as follows:

- MDB name specified in the `<ejb-name>` element
- MDB class defined in the `<ejb-class>` element
- JMS `Destination` type is a `Topic` that is specified in the `<message-driven-destination><jms-destination-type>` element
- JNDI reference information for the entity bean that this MDB invokes is defined in the `<ejb-ref>` element

### ejb-jar.xml

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">

<ejb-jar>
        <enterprise-beans>
             <message-driven>
                  <ejb-name>com.evermind.logger.MessageLogger</ejb-name>
                  <ejb-class>com.evermind.logger.MessageLogger</ejb-class>
                  <message-driven-destination>
```

```
                <jms-destination-type>javax.jms.Topic</jms-destination-type>
               </message-driven-destination>
               <ejb-ref>
                    <ejb-ref-name>logMessages</ejb-ref-name>
                    <ejb-ref-type>Entity</ejb-ref-type>
                    <home>com.evermind.logger.LogMessageHome</home>
                    <remote>com.evermind.logger.LogMessage</remote>
               </ejb-ref>
          </message-driven>
          <entity>
               <ejb-name>com.evermind.logger.LogMessage</ejb-name>
               <home>com.evermind.logger.LogMessageHome</home>
               <remote>com.evermind.logger.LogMessage</remote>
               <ejb-class>com.evermind.logger.LogMessageEJB</ejb-class>
               <persistence-type>Container</persistence-type>
               <prim-key-class>java.lang.Long</prim-key-class>
               <reentrant>False</reentrant>
               <cmp-field><field-name>id</field-name></cmp-field>
               <cmp-field><field-name>time</field-name></cmp-field>
               <cmp-field><field-name>subject</field-name></cmp-field>
               <cmp-field><field-name>message</field-name></cmp-field>
               <cmp-field><field-name>category</field-name></cmp-field>
               <primkey-field>id</primkey-field>
               <ejb-ref>
                    <ejb-ref-name>counter</ejb-ref-name>
                    <ejb-ref-type>Entity</ejb-ref-type>
                    <home>com.evermind.ejb.CounterHome</home>
                    <remote>com.evermind.ejb.Counter</remote>
               </ejb-ref>
          </entity>
     </enterprise-beans>
     <assembly-descriptor>
          <container-transaction>
               <method>
                    <ejb-name>com.evermind.logger.LogMessage</ejb-name>
                    <method-name>*</method-name>
               </method>
               <trans-attribute>Supports</trans-attribute>
          </container-transaction>
     </assembly-descriptor>
</ejb-jar>
```

### OC4J-Specific Deployment Descriptor

Once you have configured the MDB and the JMS Destination type, you must inform the container which JMS Destination to associate with the MDB. You could have several topics and queues defined in your jms.xml file. For information on defining these JMS Destination types in jms.xml, see the JMS chapter in *Oracle9iAS Containers for J2EE Services Guide*.

In order to identify the Destination that is to be associated with the MDB, you map the Destination location and connection factory to the MDB through the <message-driven-deployment> element in the orion-ejb-jar.xml file.

The following is the orion-ejb-jar.xml deployment descriptor for the MessageLogger example. It maps a JMS Topic to the MessageLogger MDB, providing the following:

- MDB name, as defined in the <ejb-name> in the EJB deployment descriptor, is specified in the name attribute.
- JMS Destination, as defined in the jms.xml file, is specified in the destination-location attribute.
- JMS Destination Connection Factory, as defined in the jms.xml file, is specified in the connection-factory-location attribute.

Once all of these are specified in the <message-driven-deployment> element, the container knows how to map the MDB to the correct JMS Destination.

```
<enterprise-beans>
  <message-driven-deployment
      name="com.evermind.logger.MessageLogger"
      destination-location="jms/theTopic"
      connection-factory-location="jms/theTopicConnectionFactory">
    <ejb-ref-mapping name="logMessages" />
  </message-driven-deployment>
  ...
</enterprise-beans>
```

### JMS XML Configuration

Configure the topic or queue that the client sends all messages to that are destined for the MDB. The name, location, and connection factory for either Destination type must be specified.

The following JMS configuration specifies a topic (theTopic) that is used by the MessageLogger example:

```
<?xml version="1.0"?>
```

```
<!DOCTYPE jms-server PUBLIC "Orion JMS server" "http://www.orionserver.com/dtds/
jms-server.dtd">

<jms-server port="9127">
        <topic name="jms/theTopic" location="jms/theTopic">
                <description>Employee topic</description>
        </topic>

        <topic-connection-factory location="jms/theTopicConnectionFactory"
               username="admin" password="welcome" />

        <!-- path to the log-file where JMS-events/errors are stored -->
        <log>
                <file path="../log/jms.log" />
        </log>
</jms-server>
```

### DataSource XML Configuration

If you use a database in your application, you should configure the `DataSource` for that database. Full details for `DataSource` configuration is provided in the *Oracle9iAS Containers for J2EE Services Guide*.

The database that the `MessageLogger` example uses is configured in the `data-sources.xml` file, as follows:

```
<data-source
                class="com.evermind.sql.DriverManagerDataSource"
                name="OracleDS"
                location="jdbc/OracleCoreDS"
                xa-location="jdbc/xa/OracleXADS"
                ejb-location="jdbc/OracleDS"
                connection-driver="oracle.jdbc.driver.OracleDriver"
                username="scott"
                password="tiger"
                url="jdbc:oracle:thin:@mysun:8852:orcl"
                inactivity-timeout="30"
/>
```

## Deploying the Entity Bean

Archive your EJB into a JAR file. You deploy the MDB the same way as the session bean, which is detailed in "Prepare the EJB Application for Assembly" on page 2-11 and "Deploy the Enterprise Application to OC4J" on page 2-13.

## Client Accessing MDB

The client sends a message to the MDB through a JMS `Destination`. The MDB is associated with the JMS `Destination` by the container. The following is a JNDI lookup of the JMS `Destination` for the `MessageLogger` MDB.

```
Context ic - new InitialContext();
Queue msgQueue = (javax.jms.Queue) ic.lookup("java:comp/env/jms/msgQueue");
```

For example, the following JSP sends a message to the MessageLogger MDB Topic:

```
<%@ page import="javax.jms.*, javax.naming.*, java.util.*" %>
<%
   TopicConnectionFactory connectionFactory =
       (TopicConnectionFactory)new InitialContext().lookup
                    ("java:comp/env/jms/theTopicConnectionFactory");
   TopicConnection connection = connectionFactory.createTopicConnection();
   connection.start();
   TopicSession topicSession =
        connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
   topicSession.start();
   Topic topic =
        (Topic)new InitialContext().lookup("java:comp/env/jms/theTopic");
   TopicPublisher publisher = topicSession.createPublisher(topic);
   Message message = topicSession.createMessage();
   message.setJMSType("logMessage");
   message.setLongProperty("time", System.currentTimeMillis());
   message.setStringProperty("subject", request.getParameter("subject"));
   message.setStringProperty("message", request.getParameter("message"));
   publisher.publish(message);
   publisher.close();
   topicSession.close();
   connection.close();
%>
Message sent!
```

# 6 Advanced EJB Subjects

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

## Accessing EJBs

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

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

1. Do you want to set up a logical name for the target bean?
   - Yes: Modify the XML configuration file to set up the `<ejb-ref>` element with the target bean information. The logical name specified in the `<ejb-ref-name>` element is used in the JNDI lookup.
   - No: The actual name of the bean is used in the JNDI lookup. This name has been specified in the target bean's XML deployment descriptors in the `<ejb-name>` element.
2. Where does the client exist relative to the target bean?
   - Within the same application as the target bean? Or is the target bean part of an application that is this client's parent? You do not need to set up any JNDI properties.
   - Otherwise, you must set up JNDI properties. There are two methods for setting up JNDI properties. See "Setting JNDI Properties" on page 6-3 for more information.

### EJB Reference Information

Specify the EJB reference information for the remote EJB in the `<ejb-ref>` element in the `application-client.xml`, `ejb-jar.xml`, or `web.xml` files. A full description or how to set up the `<ejb-ref>` element is given in "Configuring Environment References" on page 6-14.

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

```
<?xml version="1.0"?>
<!DOCTYPE application-client PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE
Application Client 1.2//EN"
"http://java.sun.com/j2ee/dtds/application-client_1_2.dtd">
```

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

OC4J maps the logical name to the actual JNDI name on the client-side. The server-side receives the JNDI name and resolves it within its JNDI tree.

## Setting JNDI Properties

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

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

### No JNDI Properties

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

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

This is also true if the target bean is in an application that has been deployed as this application's parent. To specify parents, use the `-parent` option of the admin.jar command when deploying the originating application.

## JNDI Properties File

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

### Factory

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

### Location

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

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

or

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

### Security

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

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

## JNDI Properties Within Implementation

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

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

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

```
Object homeObject = ic.lookup("java:comp/env/employeeBean");

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

## Using the Initial Context Factory Classes

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

***Example 6–1 Servlet Accessing EJB in Remote OC4J Instance***

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

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

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

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

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

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

### Accessing an EJB in a Remote Server

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

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

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

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

## Reusing or Dedicating Connections

When you execute a JNDI lookup, you retrieve a connection to the server. Each subsequent JNDI lookup for this same server uses the connection returned on the first JNDI lookup. That is, all requests are forwarded over and share the same connection. However, if you want to use a dedicated connection for each connection, specify the "`dedicated.connection`" JNDI property to be true before you retrieve the `InitialContext`, as follows:

```
env.put("dedicated.connection", "true");
```

One of the reasons for using this is if you need to retrieve multiple connections, where each uses a different username/password. If dedicated.connection is false (which is the default), the first username/password is used for all subsequent connections, even if an alternate username/password is supplied. If you want to connect using a different username/password for each connection, you must set dedicated.connection to true. Thus, you will retrieve a separate physical connection, each with its own designated username/password. It opens a new connection instead of reusing a cached connection.

## Location of Commonly-Used Classes Through Parent

If you have classes that can be used by more than one EJB, you can centralize these classes in one of the following ways:

- If two EJBs use the same classes, include these classes in one of the EJBs. Place both EJBs in the same JAR file. After deployment, both EJBs will be able to use the common classes.
- Place the commonly-used class files in a JAR file, which you place in the `$J2EE_HOME/lib` directory. Then, all classes deployed in OC4J can use these supporting classes.
- Since "child" applications can access classes that exist within a "parent" application, place the commonly-used classes in a parent application. All EJBs that use these classes should define the application that contains the common classes as its parent by adding the parent attribute to its `<application>` element in the `server.xml` file as follows:

  ```
  <application ... parent="applicationWithCommonClasses" .../>
  ```

  The `parent` attribute defines an optional 'parent' application, where the default is the global application. The children see the namespace of its parent application. This is used in order to share services such as EJBs among multiple applications.

## Changing XML Files After Deployment

Whenever you deploy an application, OC4J automatically generates the OC4J-specific XML files with the default elements. If you want to change these files or add to the existing XML files, you must copy the XML files to where your original development directory for the application and change it in this location. If you change the XML file within the deployed location, OC4J simply overwrites these changes when the application is deployed again. The changes only stay constant when changed in the development directories.

For all OC4J-specific XML files, you can add these files within the recommended development structure as shown in Figure 6–1.

***Figure 6–1 Development Application Directory Structure***

```
applications/<appname>/
    |
    |-----------META-INF/
    |            |-----------application.xml
    |-----------<ejb_module>/
    |            |-----------EJB classes (my.ejb.class maps to /my/ejb/class)
    |            |-----------META-INF/
    |                         |-----------ejb-jar.xml
    |                         |-----------orion-ejb-jar.xml
    |-----------<web_module>/
    |            |-----------index.html
    |            |-----------JSP pages
    |            |-----------WEB-INF/
    |                         |-----------web.xml
    |                         |-----------orion-web.xml
    |                         |-----------classes/
    |                         |            |-----------Servlet classes
    |                         |                        (my.Servlet to /my/Servlet)
    |                         |-----------lib/
    |                                      |-----------dependent libraries
    |-----------<client_module>/
                 |-----------Client classes
                 |-----------META-INF/
                              |-----------application-client.xml
                              |-----------orion-application-client.xml
```

## Entity Bean Concurrency and Database Isolation Modes

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

- Database Isolation Modes
- Entity Bean Concurrency Modes

### Database Isolation Modes

The `java.sql.Connection` object represents a connection to a specific database. Database isolation modes are provided to define protection against resource contention. When two or more users try to update the same resource, a lost update can occur. That is, one user can overwrite the other user's data without realizing it.

The `java.sql.Connection` standard provides four isolation modes, of which Oracle only supports two of these modes. These are as follows:

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

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

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

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

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

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

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

## Entity Bean Concurrency Modes

OC4J also provides concurrency modes for handling resource contention and parallel execution within container-managed persistence (CMP) entity beans. Bean-managed persistence entity beans manage the resource locking within the bean implementation themselves. The concurrency modes configure when to block to manage resource contention or when to execute in parallel.

The concurrency modes are as follows:

- `PESSIMISTIC`: This manages resource contention and does not allow parallel execution. Only one user at a time is allowed to execute the entity bean at a single time.
- `OPTIMISTIC`: Multiple users can execute the entity bean in parallel. It does not monitor resource contention; thus, the burden of the data consistency is placed on the database isolation modes.
- `READ-ONLY`: Multiple users can execute the entity bean in parallel. The container does not allow any updates to the bean's state.

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

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

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

Parallel execution requires that the pool size for wrapper and bean instances are set correctly. For more information on how to configure the pool sizes, see "Configuring Pool Sizes For Entity Beans" on page 6-12.

## Exclusive Write Access to the Database

The `exclusive-write-access` attribute of the `<entity-deployment>` element states that this is the only bean that accesses its table in the database and that no external methods are used to update the resource. It informs the OC4J instance that any cache maintained for this bean will only be dirtied by this bean. Essentially, if you set this attribute to true, you are assuring the container that this is the only bean that will update the tables used within this bean. Thus, any cache maintained for the bean does not need to constantly update from the back-end database.

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

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

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

## Effects of the Combination of Isolation and Concurrency Modes

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

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

### Differences Between Pessimistic and Optimistic/Serializable

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

### Affects of Concurrency Modes on Clustering

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

## Configuring Pool Sizes For Entity Beans

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

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

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

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

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

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

  Or the minimum number allowed in the pool as follows:

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

- The `max-instances-per-pk` attribute sets the maximum entity bean wrapper instances allowed in its pool for a given primary key. An entity bean's wrapper code can be pooled if it is not used by a client.

  The default maximum value is 50. Set the maximum wrapper instances as follows:

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

  Set the minimum wrapper instances as follows:

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

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

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

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

## Techniques for Updating Persistence

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

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

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

## Configuring Environment References

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

ISVs typically develop EJBs that are independent from the EJB container. In order to distance the bean implementation from the container specifics, you can create environment elements that map to one of the following: defined variables, entity beans, or resource managers. This indirection enables the bean developer to refer to existing variables, EJBs, and a JDBC `DataSource` without specifying the actual name. These names are defined in the deployment descriptor and are linked to the actual names within the OC4J-specific deployment descriptor.

### Environment variables

You can create environment variables that your bean accesses through a lookup on the `InitialContext`. These variables are defined within an `<env-entry>` element and can be of the following types: `String`, `Integer`, `Boolean`, `Double`, `Byte`, `Short`, `Long`, and `Float`. The name of the environment variable is defined within `<env-entry-name>`, the type is defined in `<env-entry-type>`, and its initialized value is defined in `<env-entry-value>`. The `<env-entry-name>` is relative to the "`java:comp/env`" context.

For example, the following two environment variables are declared within the XML deployment descriptor for `java:comp/env/minBalance` and `java:comp/env/maxCreditBalance`.

```
<env-entry>
     <env-entry-name>minBalance</env-entry-name>
     <env-entry-type>java.lang.Integer</env-entry-type>
     <env-entry-value>500</env-entry-value>
</env-entry>
<env-entry>
     <env-entry-name>maxCreditBalance</env-entry-name>
     <env-entry-type>java.lang.Integer</env-entry-type>
     <env-entry-value>10000</env-entry-value>
</env-entry>
```

Within the bean's code, you would access these environment variables through the `InitialContext`, as follows:

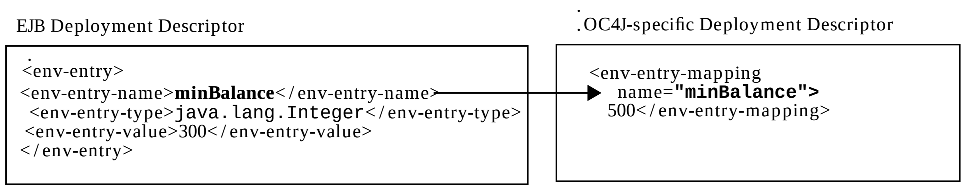
```
InitialContext ic = new InitialContext();
Integer min = (Integer)ic.lookup("java:comp/env/minBalance");
Integer max = (Integer) ic.lookup("java:comp/env/maxCreditBalance"));
```

Notice that to retrieve the values of the environment variables, you prefix each environment element with "`java:comp/env/`", which is the location that the container stored the environment variable.

If you wanted the value of the environment variable to be defined in the OC4J-specific deployment descriptor, you can map the `<env-entry-name>` to the `<env-entry-mapping>` element in the OC4J-specific deployment descriptor. This means that the value specified in the `orion-ejb-jar.xml` file overrides any value that may be specified in the `ejb-jar.xml` file. The type specified in the EJB deployment descriptor stays the same.

Figure 6–2 shows how the `minBalance` environment variable is defined as 500 within the OC4J-specific deployment descriptor.

***Figure 6–2 Environment Variable Mapping***


EJB Deployment Descriptor
<env-entry>
<env-entry-name>minBalance</env-entry-name>
<env-entry-type>java.lang.Integer</env-entry-type>
<env-entry-value>300</env-entry-value>
</env-entry>
OC4J-specific Deployment Descriptor
<env-entry-mapping
name="minBalance">
500</env-entry-mapping>

## Environment References To Other Enterprise JavaBeans

You can define an environment reference to an EJB within the deployment descriptor. If your bean calls out to another bean, you can enable your bean to invoke the second bean using a reference defined within the deployment descriptors. You create a logical name within the EJB deployment descriptor, which is mapped to the concrete name of the bean within the OC4J-specific deployment descriptor.

Declaring the target bean as an environment reference provides a level of indirection: the originating bean can refer to the target bean with a logical name.

To define a reference to another EJB within the JAR or in a bean declared as a parent, you provide the following:

1. Name—provide a name for the target bean. This name is what the bean uses within the JNDI URL for accessing the target bean. The name should begin with "`ejb/`", such as "`ejb/myEmployee`", and will be available within the "`java:comp/env/ejb`" context.

   – This name can be the actual name of the bean; that is, the name defined within the `<ejb-name>` element in the `<session>` or `<entity>` elements.

   – This name can be a logical name that you want to use in your implementation. But it is not the actual name of the bean. If you use a logical name, the actual name must either be specified in the `<ejb-link>` element in this `<ejb-ref>` element or in the `<ejb-ref-mapping>` element in the OC4J-specific deployment descriptor.

   These options are discussed below.

2. Type—define whether the bean is a session or an entity bean. Value should be either "`Session`" or "`Entity`".
3. Home—provide the fully qualified home interface name.
4. Remote—provide the fully qualified remote interface name.
5. Link —provide a name that links this EJB reference with the actual JNDI URL. This is optional.

If you have two beans in the JAR: `BeanA` and `BeanB`. If `BeanB` creates a reference to `BeanA`, you can define this reference in one of three methods:

- Provide the actual name of the bean. `BeanB` would define the following `<ejb-ref>` within its definition:

```
<ejb-ref>
 <ejb-ref-name>myBeans/BeanA</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <home>myBeans.BeanAHome</home>
 <remote>myBeans.BeanA</remote>
```

```
</ejb-ref>
```

No `<ejb-link>` is necessary for this method. However, the `BeanB` implementation must refer to `BeanA` in the JNDI retrieval, which would use `java:comp/env/myBeans/BeanA` for retrieval within an EJB or Java client and use "`myBeans/BeanA`" within a Servlet.

> **Note:** Servlets do not require the prefix of "`java:comp/env`" in the JNDI lookup. Thus, they will always either reference just the actual JNDI name or the logical name of the EJB.

- Provide the actual name of the bean in the `<ejb-link>` element. This method allows you to use any logical name in your bean implementation for the JNDI retrieval:

  ```
  <ejb-ref>
   <ejb-ref-name>ejb/nextVal</ejb-ref-name>
   <ejb-ref-type>Session</ejb-ref-type>
   <home>myBeans.BeanAHome</home>
   <remote>myBeans.BeanA</remote>
   <ejb-link>myBeans/BeanA</ejb-link>
  </ejb-ref>
  ```

  `BeanB` would use `java:comp/env/ejb/nextVal` in the JNDI retrieval of `BeanA`.

- Provide the logical name of the bean in the `<ejb-ref-name>` and the actual name of the bean in the `<ejb-ref-mapping>` element in the OC4J-specific deployment descriptor.

  The reference in the EJB deployment descriptor would be as follows:

  ```
  <ejb-ref>
   <ejb-ref-name>ejb/nextVal</ejb-ref-name>
   <ejb-ref-type>Session</ejb-ref-type>
   <home>myBeans.BeanAHome</home>
   <remote>myBeans.BeanA</remote>
  </ejb-ref>
  ```

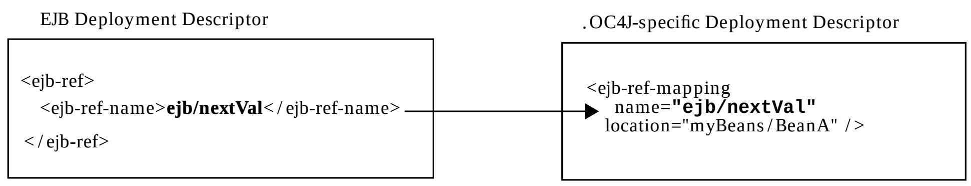
  The "`ejb/nextVal`" logical name is mapped to an actual name in the OC4J-deployment descriptor as follows:

```
<ejb-ref-mapping name="ejb/nextVal" location="myBeans/BeanA"/>
```

`BeanB` would use `java:comp/env/ejb/nextVal` in the JNDI retrieval of `BeanA`.

As shown in Figure 6–3, the logical name for the bean is mapped to the JNDI name by providing the same name, "`ejb/nextVal`", in both the `<ejb-ref-name>` in the EJB deployment descriptor and the `name` attribute within the `<ejb-ref-mapping>` element in the OC4J-specific deployment descriptor.

***Figure 6–3 EJB Reference Mapping***


EJB Deployment Descriptor
<ejb-ref>
<ejb-ref-name>**ejb/nextVal**</ejb-ref-name>
</ejb-ref>
.OC4J-specific Deployment Descriptor
<ejb-ref-mapping
name=**"ejb/nextVal"**
location="myBeans/BeanA" />

***Example 6–2 Defining an EJB Reference Within the Environment***

The following example defines a reference to the `Hello` bean, as follows:

1. The logical name used for the target bean within the originating bean is "`java:comp/env/ejb/HelloWorld`".
2. The target bean is a session bean.
3. Its home interface is `hello.HelloHome`; its remote interface is `hello.Hello`.
4. The link to the JNDI URL for this bean is defined in the OC4J-specific deployment descriptor under the "`HelloWorldBean`" name.

EJB Deployment Descriptor

```
<ejb-ref>
  <description>Hello World Bean</description>
  <ejb-ref-name>ejb/HelloWorld</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>hello.HelloHome</home>
  <remote>hello.Hello</remote>
</ejb-ref>
```

As shown in Figure 6–3, the `<ejb-link>` is mapped to the `name` attribute within the `<ejb-ref-mapping>` element in the OC4J-specific deployment descriptor by providing the same logical name in both elements. The Oracle-specific deployment descriptor would have the following definition to map the logical bean name of "`java:comp/env/ejb/HelloWorld`" to the JNDI URL "`/test/myHello`".

OC 4J-specific Deployment Descriptor

```
<ejb-ref-mapping>
  name="ejb/HelloWorld"
  location="/test/myHello"/>
```

To invoke this bean from within your implementation, you use the `<ejb-ref-name>` defined in the EJB deployment descriptor. In EJB or pure Java clients, you prefix this name with "`java:comp/env/ejb/`", which is where the container places the EJB references defined in the deployment descriptor. Servlets only require the logical name defined in the `<ejb-ref-name>`.

The following is a lookup from an EJB client:

```
InitialContext ic = new InitialContext();
HelloHome hh = (HelloHome)ic.lookup("java:comp/env/ejb/HelloWorld");
```

The following is a lookup from a Servlet, if the Servlet defines the logical name of "`ejb/HelloWorld`" in `<ejb-ref>` in its `web.xml` file and maps it to the actual name of "`/test/myHello`" within the `orion-web.xml` file.

```
InitialContext ic = new InitialContext();
HelloHome hh = (HelloHome)ic.lookup("ejb/HelloWorld");
```

## Environment References To Resource Manager Connection Factory References

The resource manager connection factory references can include resource managers such as JMS, Java mail, URL, and JDBC `DataSource` objects. Similar to the EJB references, you can access these objects from JNDI by creating an environment element for each object reference. However, these references can only be used for retrieving the object within the bean that defines these references. Each is fully described in the following sections:

- JDBC DataSource
- Mail Session
- URL

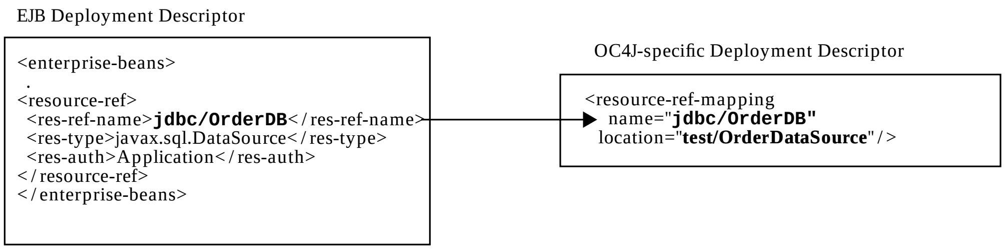
### JDBC DataSource

You can access a database through JDBC either using the traditional method or by creating an environment element for a JDBC `DataSource`. In order to create an environment element for your JDBC `DataSource`, you must do the following:

1. Define the `DataSource` in the `data-sources.xml` file.
2. Create a logical name within the `<res-ref-name>` element in the EJB deployment descriptor. This name should always start with "`jdbc`". In the bean code, the lookup of this reference is always prefaced by "`java:comp/env/jdbc`".
3. Map the logical name within the EJB deployment descriptor to the JNDI name, created in step 1, within the OC4J-specific deployment descriptor.
4. Lookup the object reference within the bean with the "`java:comp/env/jdbc`" preface and the logical name defined in the EJB deployment descriptor.

As shown in Figure 6–4, the JDBC `DataSource` uses the JNDI name "`test/OrderDataSource`". The logical name that the bean knows this resource as is "`jdbc/OrderDB`". These names are mapped together within the OC4J-specific deployment descriptor. Thus, within the bean's implementation, the bean can retrieve the connection to `OrderDataSource` by using the "`java:comp/env/jdbc/OrderDB`" environment element.

***Figure 6–4 JDBC Resource Manager Mapping***


EJB Deployment Descriptor
<enterprise-beans>
.
<resource-ref>
<res-ref-name>jdbc/OrderDB</res-ref-name>
<res-type>javax.sql.DataSource</res-type>
<res-auth>Application</res-auth>
</resource-ref>
</enterprise-beans>
OC4J-specific Deployment Descriptor
<resource-ref-mapping
name="jdbc/OrderDB"
location="test/OrderDataSource"/>

***Example 6–3 Defining an environment element for JDBC Connection***

The environment element is defined within the EJB deployment descriptor by providing the logical name, "`jdbc/OrderDB`", its type of `javax.sql.DataSource`, and the authenticator of "`Application`".

EJB Deployment Descriptor

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

The environment element of "`jdbc/OrderDB`" is mapped to the JNDI bound name for the connection, "`test/OrderDataSource`" within the Oracle-specific deployment descriptor.

OC4J-specific Deployment Descriptor

```
  <resource-ref-mapping
    name="jdbc/OrderDB"
    location="/test/OrderDataSource"/>
```

Once deployed, the bean can retrieve the JDBC `DataSource` as follows:

```
javax.sql.DataSource db;
java.sql.Connection conn;
.
.
.
db = (javax.sql.DataSource)
initCtx.lookup("java:comp/env/jdbc/OrderDB");
conn = db.getConnection();
```

**Note:** This example assumes that a `DataSource` is specified in the `data-sources.xml` file with the JNDI name of "`/test/OrderDataSource`".

### Mail Session

You can create an environment element for a Java mail `Session` object through the following:

1. Bind the `javax.mail.Session` reference within the JNDI name space in the `application.xml` file using the `<mail-session>` element, as follows:

   ```
   <mail-session location="mail/MailSession"
      smtp-host="mysmtp.oraclecorp.com">
      <property name="mail.transport.protocol" value="smtp"/>
   ```

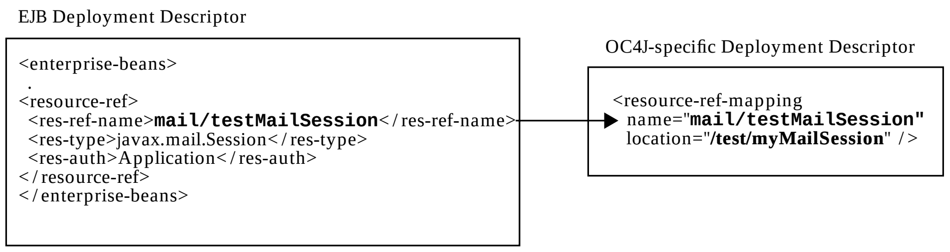
```
   <property name="mail.smtp.from" value="emailaddress@oracle.com"/>
</mail-session>
```

The location attribute contains the JNDI name specified in the location attribute of the `<resource-ref-mapping>` element in the OC4J-specific deployment descriptor.

2. Create a logical name within the `<res-ref-name>` element in the EJB deployment descriptor. This name should always start with "`mail`". In the bean code, the lookup of this reference is always prefaced by "`java:comp/env/mail`".
3. Map the logical name within the EJB deployment descriptor to the JNDI name, created in step 1, within the OC4J-specific deployment descriptor.
4. Lookup the object reference within the bean with the "`java:comp/env/mail`" preface and the logical name defined in the EJB deployment descriptor.

As shown in Figure 6–5, the `Session` object was bound to the JNDI name "`/test/myMailSession`". The logical name that the bean knows this resource as is "`mail/testMailSession`". These names are mapped together within the OC4J-specific deployment descriptor. Thus, within the bean's implementation, the bean can retrieve the connection to the bound `Session` object by using the "`java:comp/env/mail/testMailSession`" environment element.

***Figure 6–5 Session Resource Manager Mapping***


EJB Deployment Descriptor
<enterprise-beans>
.
<resource-ref>
<res-ref-name>mail/testMailSession</res-ref-name>
<res-type>javax.mail.Session</res-type>
<res-auth>Application</res-auth>
</resource-ref>
</enterprise-beans>
OC4J-specific Deployment Descriptor
<resource-ref-mapping
name="mail/testMailSession"
location="/test/myMailSession" />

This environment element is defined with the following information:

| Element | Description |
|---|---|
| `<res-ref-name>` | The logical name of the `Session` object to be used within the originating bean. The name should be prefixed with "`mail/`". In our example, the logical name for our ordering database is "`mail/testMailSession`". |

| Element | Description |
|---|---|
| `<res-type>` | The Java type of the resource. For the Java mail `Session` object, this is `javax.mail.Session`. |
| `<res-auth>` | Define who is responsible for signing on to the database. The value can be "`Application`" or "`Container`" based on who provides the authentication information. |

***Example 6–4 Defining an environment element for Java mail Session***

The environment element is defined within the EJB deployment descriptor by providing the logical name, "`mail/testMailSession`", its type of `javax.mail.Session`, and the authenticator of "`Application`".

EJB Deployment Descriptor

```
<resource-ref>
  <res-ref-name>mail/testMailSession</res-ref-name>
  <res-type>javax.mail.Session</res-type>
  <res-auth>Application</res-auth>
</resource-ref>
```

The environment element of "`mail/testMailSession`" is mapped to the JNDI bound name for the connection, "`test/myMailSession`" within the OC4J-specific deployment descriptor.

OC4J-specific Deployment Descriptor

```
<resource-ref-mapping
  name="mail/testMailSession"
  location="/test/myMailSession" />
```

Once deployed, the bean can retrieve the `Session` object reference as follows:

```
InitialContext ic = new InitialContext();
Session session = (Session)
ic.lookup("java:comp/env/mail/testMailSession");

//The following uses the mail session object
//Create a message object
MimeMessage msg = new MimeMessage(session);

//Construct an address array
String mailTo = "whosit@oracle.com";
InternetAddress addr = new InternetAddress(mailto);
InternetAddress addrs[] = new InternetAddress[1];
addrs[0] = addr;

//set the message parameters
msg.setRecipients(Message.RecipientType.TO, addrs);
msg.setSubject("testSend()" + new Date());
msg.setContent(msgText, "text/plain");

//send the mail message
Transport.send(msg);
```

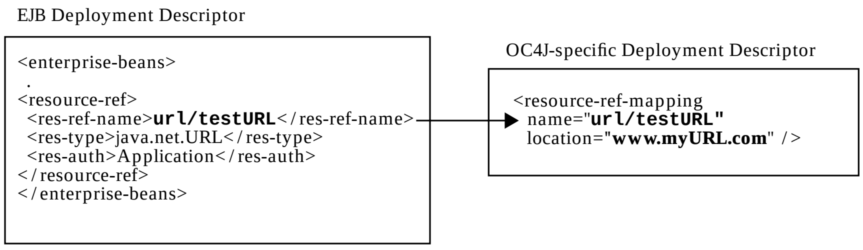
### URL

You can create an environment element for a Java `URL` object through the following:

1. Create a logical name within the `<res-ref-name>` element in the EJB deployment descriptor. This name should always start with "`url`". In the bean code, the lookup of this reference is always prefaced by "`java:comp/env/url`".
2. Map the logical name within the EJB deployment descriptor to the URL within the OC4J-specific deployment descriptor.
3. Lookup the object reference within the bean with the "`java:comp/env/url`" preface and the logical name defined in the EJB deployment descriptor.

As shown in Figure 6–6, the `URL` object was bound to the URL "`www.myURL`.com". The logical name that the bean knows this resource as is "`url/testURL`". These names are mapped together within the OC4J-specific deployment descriptor. Thus, within the bean's implementation, the bean can retrieve the connection to the bound `Session` object by using the "`java:comp/env/url/testURL`" environment element.

***Figure 6–6 URL Resource Manager Mapping***


EJB Deployment Descriptor
<enterprise-beans>
.
<resource-ref>
<res-ref-name>url/testURL</res-ref-name>
<res-type>java.net.URL</res-type>
<res-auth>Application</res-auth>
</resource-ref>
</enterprise-beans>
OC4J-specific Deployment Descriptor
<resource-ref-mapping
name="url/testURL"
location="www.myURL.com" />

This environment element is defined with the following information:

| Element | Description |
|---|---|
| `<res-ref-name>` | The logical name of the `URL` object to be used within the originating bean. The name should be prefixed with "`url/`". In our example, the logical name for our ordering database is "`url/testURL`". |
| `<res-type>` | The Java type of the resource. For the Java URL object, this is `java.net.URL`. |
| `<res-auth>` | Define who is responsible for signing on to the database. At this time, the only value supported is "`Application`". The application provides the authentication information. |

***Example 6–5 Defining an environment element for JDBC Connection***

The environment element is defined within the EJB deployment descriptor by providing the logical name, "`url/testURL`", its type of `java.net.URL`, and the authenticator of "`Application`".

EJB Deployment Descriptor

```
<resource-ref>
  <res-ref-name>url/testURL</res-ref-name>
  <res-type>java.net.URL</res-type>
  <res-auth>Application</res-auth>
</resource-ref>
```

The environment element of "`url/testURL`" is mapped to the URL "`www.myURL.com`" within the OC4J-specific deployment descriptor.

OC4J-specific Deployment Descriptor

```
<resource-ref-mapping
  name="url/testURL"
  location="www.myURL.com" />
```

Once deployed, the bean can retrieve the `URL` object reference as follows:

```
InitialContext ic = new InitialContext();
URL url = (URL) ic.lookup("java:comp/env/url/testURL");

//The following uses the URL object
URLConection conn = url.openConnection();
```

## Configuring Security

Choosing what security to use is a subject that involves more than the authorization details that we discuss in this section. For a full description of security, see the *OC4J Services Guide*.

This book focuses on EJBs and any XML configuration that belongs within the EJB deployment descriptors. Within the security spectrum, EJB authorization is specified within the EJB and OC4J-specific deployment descriptors. It involves assigning roles that are attached to EJBs to users and groups that are defined in the `principals.xml` file. Thus, this section describes assigning roles to EJBs and mapping these roles to specific users or groups.

You can manage the authorization piece of your security within the deployment descriptors, as follows:

- The EJB deployment descriptor describes access rules using logical roles.
- The OC4J-specific deployment descriptor maps the logical roles to concrete users and groups, which are defined in `principals.xml`.

### Users, Groups, and Roles

Users and groups are identities known by the container. Roles are the *logical* identities each application uses to indicate access rights to its different objects. The username/passwords can be digital certificates and, in the case of SSL, private key pairs. The EJB deployment descriptor indicates what roles are needed to access the different parts of the application. The OC4J-specific deployment descriptor provides a mapping between the logical roles and the users/groups known by the container.

Defining users, groups, and roles are discussed in the following sections:

- Specifying Users and Groups
- Specifying Logical Roles in the EJB Deployment Descriptor
- Mapping Logical Roles to Users and Groups

#### Specifying Users and Groups

OC4J supports the definition of users and groups—either shared by all deployed applications or specific to given applications.

- Shared users and groups are defined in the `config/principals.xml` file.

- Application-specific users and groups are listed in the application-specific `principals.xml`, which path is indicated in the `orion-application.xml` file of that application.

The following excerpt from the `principals.xml` file shows how to define a group named `managers` and a user named `guest` with password, `welcome`.

```
<principals>
 <groups>
  <group name="managers">
   <description>Group for all managerial users</description>
   <permission name="rmi:login" />
   <permission name="com.evermind.server.rmi.RMIPermission" />
  </group>
....other groups...
 </groups>
 <users>
  <user username="guest" password="welcome">
   <description>purchase order manager</description>
   <group-membership group="managers" />
  </user>
 </users>
</principals>
```

For a full description of the `principals.xml` file and how to specify users and groups, see the *OC4J Services Guide*.

## Specifying Logical Roles in the EJB Deployment Descriptor

Specify the logical roles that your application uses in the EJB deployment descriptor. The roles are defined within the element named `<security-role>`.

***Example 6–6 EJB JAR Security Role Definition***

This example creates a logical role named POMGR in the `ejb-jar.xml` deployment descriptor.

1. Define the logical security role, POMGR in the `<security-role>` element.

   ```
   <security-role>
    <description>Purchase Order Manager</description>
    <role-name>POMGR</role-name>
   </security-role>
   ```

2. Define the bean and methods that this role can access in the `<method-permission>` element.

   ```
   <method-permission>
    <role-name>POMGR</role-name>
    <method>
     <ejb-name>PurchaseOrderBean</ejb-name>
     <method-name>*</method-name>
    </method>
   </method-permission>
   ```

## Mapping Logical Roles to Users and Groups

Map the logical roles defined in the application deployment descriptors to concrete users or groups through the `<security-role-mapping>` element in the OC4J-specific deployment descriptor.

***Example 6–7 Mapping Logical Role to Actual Role***

This example maps the logical role `POMGR` to the `managers` group in the `orion-ejb-jar.xml` file. Any user that can log in as part of this group is considered to have the `POMGR` role; thus, it can execute the methods of `PurchaseOrderBean`.

```
<security-role-mapping name="POMGR">
 <group name="managers" />
</security-role-mapping>
```

**Note:** You can map a logical role to a single group or to several groups.

To map this role to a specific user, do the following:

```
<security-role-mapping name="POMGR">
 <user name="guest" />
</security-role-mapping>
```

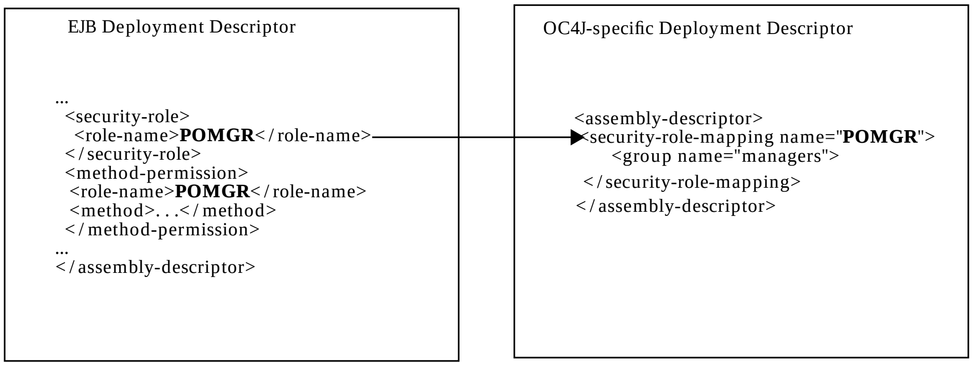
Lastly, you can map a role to a specific user within a specific group, as follows:

```
<security-role-mapping name="POMGR">
 <group name="managers" />
 <user name="guest" />
```

```
</security-role-mapping>
```

As shown in Figure 6–7, the logical role name for `POMGR` defined in the EJB deployment descriptor is mapped to SCOTT within the OC4J-specific deployment descriptor in the `<security-role-mapping>` element.

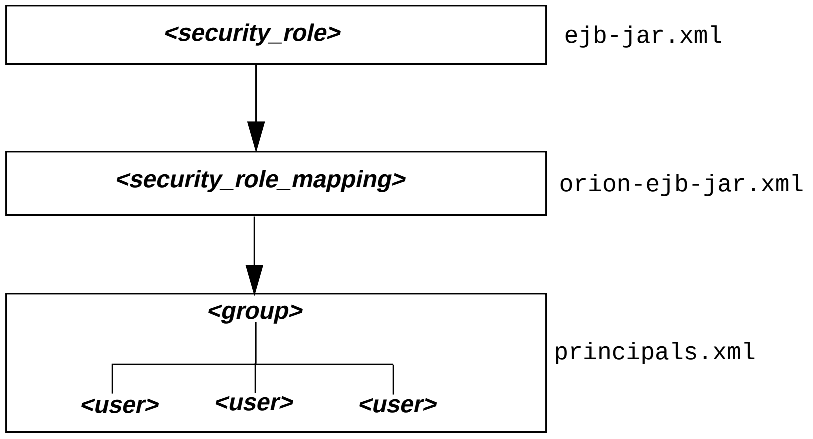
***Figure 6–7 Security Mapping***



Notice that the `<role-name>` in the EJB deployment descriptor is the same as the name attribute in the `<security-role-mapping>` element in the OC4J-specific deployment descriptor. This is what identifies the mapping.

Thus, the definition and mapping of roles is demonstrated in Figure 6–8.

***Figure 6–8 Role Mapping***



## Default Role Mapping

To default all methods in EJBs that have not been associated with a method permission to a security role, use the `<default-method-access>` element.

The following example shows how all methods not associated with a method permission are mapped to the "`managers`" group:

```
<default-method-access>
    <security-role-mapping name="users" impliesAll="true" />
       <group name="managers" />
     </security-role-mapping>
</default-method-access>
```

The `impliesAll` attribute specifies that this includes all users.

## Authenticating EJB Clients

When you access EJBs in a *remote* container, you must pass valid credentials to this container.

- Stand-alone clients define their credentials in the `jndi.properties` file deployed with the EAR file.
- Servlets or JavaBeans running within the container pass their credentials within the `InitialContext`, which is created to look up the remote EJBs.

### Credentials in JNDI Properties

Indicate the username (principal) and password (credentials) to use when looking up remote EJBs in the `jndi.properties` file.

For example, if you want to access remote EJBs as `POMGR/welcome`, define the following properties. The `factory.initial` property indicates that you will use the Oracle JNDI implementation:

```
java.naming.security.principal=POMGR
java.naming.security.credentials=welcome
java.naming.factory.initial=
com.evermind.server.ApplicationClientInitialContextFactory
java.naming.provider.url=ormi://localhost/ejbsamples
```

In your application program, authenticate and access the remote EJBs, as shown below:

```
InitialContext ic = new InitialContext();
CustomerHome =
(CustomerHome)ic.lookup("java:comp/env/purchaseOrderBean");
```

### Credentials in the InitialContext

To access remote EJBs from a servlet or JavaBean, pass the credentials in the `InitialContext` object, as follows:

```
Hashtable env = new Hashtable();
env.put("java.naming.provider.url", "ormi://localhost/ejbsamples");
env.put("java.naming.factory.initial",
        "com.evermind.server.ApplicationClientInitialContextFactory");
env.put(Context.SECURITY_PRINCIPAL, "POMGR");
env.put(Context.SECURITY_CREDENTIALS, "welcome");
Context ic = new InitialContext (env);
CustomerHome =
(CustomerHome)ic.lookup("java:comp/env/purchaseOrderBean")
```

## Common Errors

The following are common errors that may occur when executing EJBs:

- NamingException Thrown
- Deadlock Conditions

### NamingException Thrown

If you are trying to remotely access an EJB and you receive an `javax.naming.NamingException` error, your JNDI properties are probably not initialized properly. See "Accessing EJBs" on page 6-2 for a discussion on setting up JNDI properties when accessing an EJB from a remote object or remote servlet.

### Deadlock Conditions

If the call sequence of several beans cause a deadlock scenario, the OC4J container notices the deadlock condition and throws a Remote exception that details the deadlock condition in one of the offending beans.

# 7 EJB Clustering

EJB clustering offers improved scalability and high-availability through the following circumstances:

- At a certain point, too many incoming client requests can overpower the abilities of your server. You can set up your environment to balance the load of incoming client requests among several servers.
- Servers failing and connections dropping occasionally happens. You can configure several servers in a cluster, so that communication is rerouted to another server in a failover situation.

The methods for providing load balancing and clustering for failover are different for HTTP requests than for EJB communications because Web components use different protocols than EJB components. This chapter discusses EJB clustering; the instructions for setting up the HTTP failover and load balancing environment is detailed in *Oracle9i Application Server Performance Guide*.

The following is discussed in this chapter:

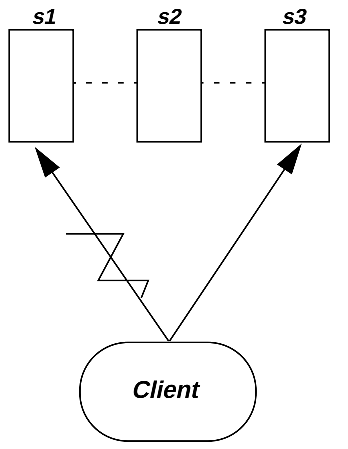
## EJB Clustering Overview

To create an EJB cluster, you specify OC4J nodes that are to be involved in the cluster, configure each of them with the same multicast address, username, and password, and deploy the EJB to be clustered to each of the nodes in the cluster.

Unlike HTTP clustering, OC4J nodes included in an EJB cluster are not currently grouped in an island and do not have a load balancer as a front-end. Instead, the EJB client container stubs discover—either statically or dynamically—all the OC4J nodes in the EJB cluster, shuffle the destination addresses, and choose one from this group for the connection. Thus, the only method for load balancing and failover is a random methodology.

As Figure 7–1 demonstrates, the client container stubs chose server "`s1`" for its EJB connection. However, sometime during the conversation, the connection went down. At this point, the client container stubs shuffle the remaining OC4J node addresses and choose another server to connect to for the failover. In this example, server "`s3`" from the OC4J cluster resumes the conversation.

***Figure 7–1 EJB Clustering Diagram***



The client container stubs discover the OC4J server addresses by one of the following methods:

- static cluster discovery method

  The JNDI addresses of all OC4J nodes that should be contacted for load balancing and failover are provided in the lookup URL, and each address is separated by a comma. For example, the following URL definition provides the client container with three OC4J nodes to use for load balancing and failover.

```
java.naming.provider.url=ormi://s1:23791/ejbsamples,
     ormi://s2:23793/ejbsamples, ormi://s3:23791/ejbsamples;
```

- dynamic cluster discovery method

  The JNDI addresses of all OC4J nodes that can be contacted for load balancing and failover are dynamically discovered during the first JNDI lookup. The client must perform a lookup with a "`lookup:`" prefix, as follows:

  ```
  ic.lookup("lookup:ormi://s1:23971/ejbsamples");
  ```

  During the JNDI lookup, server "`s1`" contacts the other OC4J nodes in the cluster, which are identified as a cluster if they all have the same multicast address (host/port), and retrieves their `ormi` addresses. These addresses are sent back to the client container. From this point forward, the client container shuffles these addresses for any load balancing or failover needs.

  However, the client container never tries to rediscover these addresses. Therefore, if you remove a node from the cluster and add another one during the connection, the client container will be unaware of it until the next time the client re-discovers the cluster nodes through the "`lookup:`" method.

The state of all beans are replicated at the end of every method call to all nodes in the cluster. This option is the most reliable in that the state of the bean is replicated to all nodes in the cluster, using a JMS multicast topic to all nodes in the cluster—which uses the same multicast address. This state stays in the topic until it is needed. Then when a method call comes in on the alternate node, the latest state for the bean is found in the JMS topic, reinstated, and the bean invocation continues.

These methods have different repercussions for each of the EJB types, which are discussed in the following sections:

- Stateless Session Bean Clustering
- Stateful Session Bean Clustering
- Entity Bean Clustering
- Combination of HTTP and EJB Clustering

## Stateless Session Bean Clustering

Stateless session beans do not require any state to be replicated among nodes in a cluster. Thus, the only use of the clustering methods that stateless session beans have is load balancing between nodes. Both the dynamic and state cluster discovery methods can be used for stateless session beans. Failover defaults to the remote invocation handler by redirecting a request.

## Stateful Session Bean Clustering

Stateful session beans require state to be replicated among nodes. In fact, stateful session beans must send all their state between the nodes, which can have a noticeable effect on performance. Thus, the following replication modes are available to you to decide on how to manage the replication performance cost:

- JVM termination replication mode—The state of the stateful session bean is replicated to only one other node in the cluster when the JVM is terminating, which uses JDK 1.3 shutdown hooks. Thus, you must use JVM version 1.3 or later. Within the JVM shutdown process, the state of all stateful session beans within this JVM is replicated to another server on the same multicast address. This is the most performant option, because the state is replicated only once. However, it is not very reliable, for the following reasons:
  - Your state will not be replicated if the power is shut off unexpectedly.
  - The state of the bean exists only on a single node at any time; the depth of failure is equal to one node.
- Stateful session context replication model—This is a finer-grain replication mode. In HTTP clustering, you can manage when and the type of information that is replicated through the `setAttribute` method of the `HTTPSession` object. Oracle offers a similar method through a new OC4J-specific class: `com.evermind.server.ejb.statefulSessionContext`. Although this option is a performant and reliable mechanism, it does not comply with the J2EE specification. Thus, if you provide this within your server code, you cannot port this application to any non-Oracle J2EE server.

## Entity Bean Clustering

The state of the entity bean is saved in a persistent storage, such as a database. Thus, when the client loses the connection to one node in the cluster, it can switch to another node in the cluster without worrying about replication of the entity bean state. However, to ensure that the state is updated from the persistent storage when the load balancing occurs, the entity bean that changes state notifies other nodes that their state is no longer in synch. That is, that their state is "dirty". At this point, nothing is done. If failover occurs and the client accesses another node for this entity bean, then the bean notices that its cache is dirty and resynchronizes its cache to the "`READ_COMMITTED`" state within the database.

### Combination of HTTP and EJB Clustering

If you have a servlet that invokes an EJB, you must include both the HTTP and EJB clustering. For HTTP clustering options, see the HTTP clustering white paper. The type of EJB clustering you choose is based on the EJB type. If you do not configure for both types, you will not have the proper state replication for the type for which you did not configure.

If the HTTP invokes an EJB that is colocated, the `EJBReference` cannot be replicated to another node unless EJB clustering has been enabled. Instead, a null pointer will be copied to the other node. So, you must provide for both types of clustering in order for all of the correct information to be replicated.

## Enabling Clustering For EJBs

To enable the OC4J nodes for EJB clustering, you must perform the following steps:

1. Configure each node in the cluster with the multicast address and a unique node identifier.
2. Configure state replication for any stateful session beans or state synchronization for entity beans.
3. Deploy the EJB to be clustered on all nodes.
4. Modify the client to use either the dynamic or static method for retrieving the cluster node addresses. The dynamic method is recommended.

### Configure Nodes With Multicast Address and Identifier

When you are configuring each OC4J node included in the EJB cluster, you must configure each node with an identical multicast address (host and port number), username, and password. However, each node in the cluster should also include its own unique identifier within the cluster. You can test a network for multicast ability by pinging the following hosts:

- To ping all multicast hosts, execute: `ping 224.0.0.1`.
- To ping all multicast routers, execute: `ping 224.0.0.2`.

Modify the **`rmi.xml`** file and add the `<cluster>` tag to configure the multicast address, username, password, and identifier for the OC4J node, as follows:

```
<cluster host=<multi_host> port=<multi_port>
              username=<multi_user> password=<multi_pwd> />
```

where each variable should be the following:

- `multi_host`: The multicast host used for the EJB cluster that communicates among the nodes in the cluster. The IP addresses that you can use for multicast are between 224.0.0.0 and 239.255.255.255. You must specify this variable.
- `multi_port`: The multicast port used for the EJB cluster for communication among the nodes in the cluster.
- `multi_user/multi_pwd`- The username and password used to authenticate itself to other nodes in the cluster. If the username and password are different for other nodes in the cluster, they will fail to communicate. You can have multiple username and password combinations within a multicast address. Those with the same username/password combinations will be considered a unique cluster.

For example, the following cluster definition identifies a cluster on mulitcast address of host=230.0.0.1, port=9127, username=`mult1`, and password=`hwdr`:

```
<cluster host="230.0.0.1" port="9127"
          username="mult1" password="hwdr"/>
```

You can specify the node identifier number as follows:

- Specify the node identifier in the **`server.xml`** file with the `<cluster>` tag, as follows:

  ```
  <cluster id="123"/>
  ```

- If no identifier is specified, a default identifier consists of the host IP address and port of the node itself.

---

**Note:** The dynamic peer discovery mechanism uses RMI as the mechanism for communication. You must have an RMI listener configured in the `rmi.xml` file with the following syntax:

```
<rmi_server host="<hostname>" port="<port>" />
```

The host name must be the actual name of your node. Do not use the "`localhost`" variable.

---

## EJB Replication Configuration

Modify the **`orion-ejb-jar.xml`** file to add the configuration for stateful session beans and entity beans require for state replication. The following sections offer more details:

- Stateful Session Bean Replication Configuration
- Entity Bean Replication Configuration

### Stateful Session Bean Replication Configuration

You configure the replication type for the stateful session bean within the bean deployment descriptor. Thus, each bean can use a different type of replication.

**VM Termination Replication** Set the `replication` attribute of the `<session-deployment>` tag in the `orion-ejb-jar.xml` file to "`VMTermination`". This is shown below:

```
<session-deployment replication="VMTermination" .../>
```

**End of Call Replication** Set the `replication` attribute of the `<session-deployment>` tag in the `orion-ejb-jar.xml` file to "`endOfCall`". This is shown below:

```
<session-deployment replication="EndOfCall" .../>
```

**Stateful Session Context** No static configuration is necessary when using the stateful session context to replicate information across the clustered nodes. To replicate the desired state, set the information that you want replicated and execute the `setAttribute` method within the `StatefulSessionContext` class in the server code. This enables you to designate what information is replicated and when it is replicated. The state indicated in the parameters of this method is replicated to all nodes in the cluster that share the same multicast address, username, and password.

### Entity Bean Replication Configuration

Configure the clustering for the entity bean within its bean deployment descriptor.

Modify the `orion-ejb-jar.xml` file to add the `clustering-schema` attribute to the `<entity-deployment>` tag, as follows:

```
<entity-deployment ... clustering-schema="asynchronous-cache" .../>
```

## Deploy EJB Application To All Nodes

Deploy the EJB application to all nodes in the cluster. If you do not do so, the client container shuffles through the nodes in the cluster until it finds a node with the EJB deployed on it. This will affect your performance.

You can either deploy the application to each node individually using the `-cluster` option of the `admin.jar` tool or you can use Oracle Enterprise Manager (OEM), which can deploy your application for you to multiple nodes.

Use the following syntax with the `admin.jar` tool:

```
java -jar admin.jar ormi://myhost admin welcome
          -deploy -file bmpapp.ear -deploymentName bmpapp -cluster
```

## Application Client Retrieval Of Clustered Nodes

The client container designates randomly within the nodes in the cluster where to direct the client request. As discussed above, the container discovers the nodes within the cluster through one of the following methods:

- Static Retrieval
- Dynamic Retrieval

### Static Retrieval

The JNDI addresses of all OC4J nodes that should be contacted for load balancing and failover are supplied in the lookup URL, and each address is separated by a comma. For example, the following URL definition provides the client container with three OC4J nodes to use for load balancing and failover.

```
java.naming.provider.url=ormi://s1:23791/ejbsamples,
        ormi://s2:23793/ejbsamples, ormi://s3:23791/ejbsamples;
```

### Dynamic Retrieval

The JNDI addresses of all OC4J nodes that can be contacted for load balancing and failover are dynamically discovered during the first JNDI lookup. The client must perform a lookup with a "`lookup:`" prefix, as follows:

```
ic.lookup("lookup:ormi://s1:23971/ejbsamples");
```

During the JNDI lookup, server "`s1`" contacts the other OC4J nodes in the cluster, which are identified as a cluster if they all have the same multicast address (host/port), and retrieves their `ormi` addresses. These addresses are sent back to the client container. From this point forward, the client container shuffles these addresses for any load balancing or failover needs.

The client container never tries to rediscover these addresses, though. Therefore, if you remove a node from the cluster and add another one during the connection, the client container will be unaware of it until the next JNDI lookup.

## Load Balancing Options

If you configure for load balancing, it balances the load at the connection level. However, if you want load balancing to occur on each JNDI lookup, configure the `LoadBalanceOnLookup` property to true in the JNDI properties before retrieving the InitialContext, as follows:

```
env.put("LoadBalanceOnLookup", "true");
```

# 8

# Active Components For Java

Active Components for Java (AC4J) enables applications to interact as peers in a loosely-coupled manner. Two or more applications participating in a business interaction, exchange information for requesting service and for responding results.

This document describes the architecture needed and the software provided to manage loosely-coupled, interactions between autonomous applications.

## Future Needs of Business Applications

The future of business applications requires the ability to perform loosely-coupled interactions. That is, applications should be able to exchange information with other applications over a long period of time, without limiting resources, and by surviving system crashes. The following lists the requirements for loosely-coupled interactions:

1. Autonomous peer—Each application, when interacting with another application, exists as an autonomous peer. That is, the responding application may choose to ignore the request, or to execute one or more functions on behalf of the requestor (possibly different than the one that the requestor asked for), before responding to the initiating application. As peers, both applications can make requests to each other, but neither can require submission from the other. Neither application can assume control over the resources that its peer application owns.
2. No time constraints—Because the tasks performed sometimes take days, even months, to complete, the time limits imposed must be longer than what can be accomplished in a short period of time.
3. Asynchronous exchange of information—Loosely-coupled interactions require that all exchange of information exists within an asynchronous environment. The inter-peer interaction can be one of the following:
   - Distribute information interaction—Sometimes applications must be able to distribute information asynchronously to its peer where no response is required. However, reliability of the delivery for this message must be ensured.
   - Request / response interaction between components in an asynchronous manner—Applications use a request and response mode of communication, where each entity knows where to respond back to with the results.
4. Reliable, recoverable, and restartable—In order for any application to exist over such a period of time, the application must be reliable—that is, recoverable and restartable—in case of system failures during that time.
5. Scalable—The application must be scalable. That is, the long-running application cannot block execution or lock resources for long periods of time. In order for the application to execute in a reasonable time frame, the framework must provide performance enhancements through concurrently executing computations.

6. Tractable—The application must be able to define and track its business processes and their interaction patterns, as follows:

   a. What business processes have started / completed under what business conditions.

   b. What business processes are pending waiting for what business documents.

   c. What is the pattern of interaction of the business processes, where processes can exchange information and what information they are authorized to push / pull to / from other processes.

   d. What is the sequencing of execution of the defined processes.

## Current Programming Models

The current architectures available are the following:

- Remote Procedure Call Model—Provides a tightly-coupled environment that uses request / response mechanisms in communication.
- Database Transactional Queuing Model—Provides a loosely-coupled environment that uses a one-way mechanism for communication.

The following sections briefly describe these models and show why they do not provide the basis necessary for the five goals presented in "Future Needs of Business Applications" on page 8-2.

### Remote Procedure Call Model

The Remote Procedure Call (RPC) programming model facilitates a tightly-coupled environment that provides for request / response communication. Transactional RPC implementations provide for ACID qualities.

Most RPC implementations currently provide two modes of method invocations: synchronous and deferred synchronous.

#### Transactional RPC Synchronous Invocation

The client program blocks when a remote invocation is made and waits until the results arrive or an exception is thrown. Examples of application types that use transactional RPC implementations are EJB, and most CORBA applications. Web services are also based on the RPC model, but are not transactional.

**Advantage** This model of communication—also called on-line or connected—is based on the request/response paradigm, where the requester and responder of the service are tightly-coupled. Tightly-coupled applications understand how to reply transparently to the requestor.

**Disadvantage** The programs must be available and running for the application to work. In the event of a network or machine failure, or when the application providing the service is busy, the application is not able to continue forward with its processing work. In this case, the state is inconsistent and the application must decide to rollback to a consistent state through JTA. Also, it is not autonomous. One application can control resources of other applications for a long time.

JTA is based on the two-phase commit specification. Two phase commit protocol may cause loss of application autonomy in the case of network disconnection, where the coordinator is unable of making a coherent global decision over the outcome of the global transaction for a long period of time.

**Example** If a purchase order is created and the customer wants to purchase 20 widgets, then the transactional RPC application must do two things:

1. Check inventory for 20 widgets and ask for them to be shipped to the customer.
2. Check the customer's credit to see if the customer has the ability to purchase these widgets.

In this example, an RPC synchronous application would (within a global transaction) do the following:

1. Send a request to the inventory database and block until the answer returns.
2. Send a request to the credit bureau and block until the answer returns.

If all request come back with a satisfactory report, then the transaction is committed and the purchase order is forwarded on to shipping. If one of the two requests fails, the transaction is rolled back. Granted, the application could have the following alternatives that prevents the transaction from being rolled back:

- If the inventory is not available, ask the customer if he/she will wait for a back-order.
- If the credit check failed, ask the customer for an alternate method of payment.

If the transaction is rolled back, the purchase order is voided.

### RPC Deferred Synchronous Invocation

An RPC deferred synchronous invocation is queue-oriented. The client places a request in a queue and is then able to continue processing without blocking for the response. An example of this is a CORBA DII application.

**Advantages** The client does not need to wait for a reply to the request. Instead, it continues processing. Then when the client wants to receive a response, it blocks or polls for the availability of the response. A response can only be delivered to the same process that made the original deferred request. Thus, if multiple deferred requests are pending, only one response is processed at a time.

**Disadvantage** If the client is non-existent, then the response is lost. Thus, for deferred execution to work correctly in the presence of network, machine, and application failures, the requests must be stored persistently and processed exactly once.

**Example** In the purchase order example, the requests to the inventory and credit bureau can be made in parallel. After executing both requests, the client can poll for both responses. The disadvantages would be the same as listed within the RPC synchronous invocation example.

## Database Transactional Queuing Model

The database transactional queuing model supports a loosely-coupled environment where applications use one-way communication. Oracle AQ is an implementation of a database transactional queuing model.

Applications need to process and deliver each message exactly once, even in presence of multiple failures of the sender or the receiver. Mixing the transactional ACID construct with queue processing creates a model that enables applications to reliably process messages with the ACID guarantees.

Applications can be disconnected for long periods of time and occasionally they can reconnect to communicate, using messages. By de-coupling the applications that send messages from the applications that receive messages and process them, queuing facilitates complex scheduling of autonomous applications. Each message can be durably saved until processed exactly once. Processing of the data is performed in a time independent fashion, even in a situation where a message receiver is temporary unavailable.

**Advantages** Delivers and processes messages exactly once, no matter whether the network or receiver application is available or not.

**Disadvantages** This model is based on sending and receiving messages. It is not based on requesting and responding to service requests, which is the foundation of all business protocols for loosely-coupled applications. To satisfy this requirement, the application shoulders the burden of creating and parsing each message. Both sides must know the format, security, and headers required for each message. There is no automatic mechanism for routing messages and executing business methods. The implementation of application logic for these mechanisms is the responsibility of the applications. If a response is called for, the application cannot easily reply, because there is no context that captures the relationship between a requester and a responder application, which is the case for RPC. It is not intended for a request/response environment, so if the client needs a response back from the destination object, it must receive and parse a separate message off of its own queue.

Exception handling describes communication failures and not application exceptions.

There is no guarantee for the consistency of the business transactions. Instead, the program itself must guarantee that the application semantic rollbacks occur appropriately in a failure situation.

**Example** In the purchase order example, the client would enqueue a message to the inventory queue and another to the credit bureau queue. Both must be reliably processed once in order for the transaction to commit. If either the inventory is not available or the client's credit is not good, the business transaction cannot be successfully completed and another message must be created to semantically rollback the one message that was processed positively.

## AC4J Framework

The RPC and transactional database queuing models both have advantages and disadvantages. The disadvantages within J2EE application types are as follows:

- The tightly-coupled, synchronous communication of EJBs does not allow loosely-coupled interactions, nor autonomous peer communication.
- The loosely-coupled, asynchronous communication of JMS provides no correlation of messages nor supports application consistency. JMS only provides a transport with no syntax for one-way messages.
- The loosely-coupled, asynchronous communication of JMS does not enable request/response interaction between entities.

- The need for the JTA Coordinator to control all resources involved in the two-phase commit cannot include autonomous resources in the global transaction.

The disadvantages prevent each model from solving the business goals laid out in "Future Needs of Business Applications" on page 8-2. Thus, a new model is necessary to incorporate the advantages of both models and exclude the disadvantages.

AC4J is a manager of loosely-coupled interactions between autonomous EJB applications. You can partition the application into concurrently executing active units of work—known as Reactions—whose execution is driven by data availability and its purpose is to execute business logic and produce new data. AC4J coordinates the flow of data between Reactions. When data become available on AC4J, the conditions specified by all registered Reactions are checked and if satisfied, then the execution of the methods of all matched Reactions is triggered.

## AC4J Architecture

AC4J allows EJBs to interact in a loosely-coupled fashion. It provides the following features:

- Support for reliable Asynchronous, Disconnected, one way or request/response type of interaction with complexities of JMS programming removed.

  It hides queues/topics and related JMS constructs from applications, provides automatic definition of communication message formats, and packs/unpacks messages, automatic routing of service requests to the appropriate service provider, automatic security context propagation, authorization and identity impersonation, and automatic exception routing and handling, which is integrated in the EJB framework.

- Transactional Data Driven execution of EJB applications.

  Composite matching on available data based on specified rules, which describe under which conditions these data can fire which EJB method. Transparent scheduling and activation of EJBs and execution of their methods.

- Support for Fork/Join operations: parallel invocation of EJB methods and synchronization on their results.
- Automatic Tracking of the work in progress.

### Introduction to AC4J Components

AC4J provides a framework for loosely-coupled interactions, which are included in the following components:

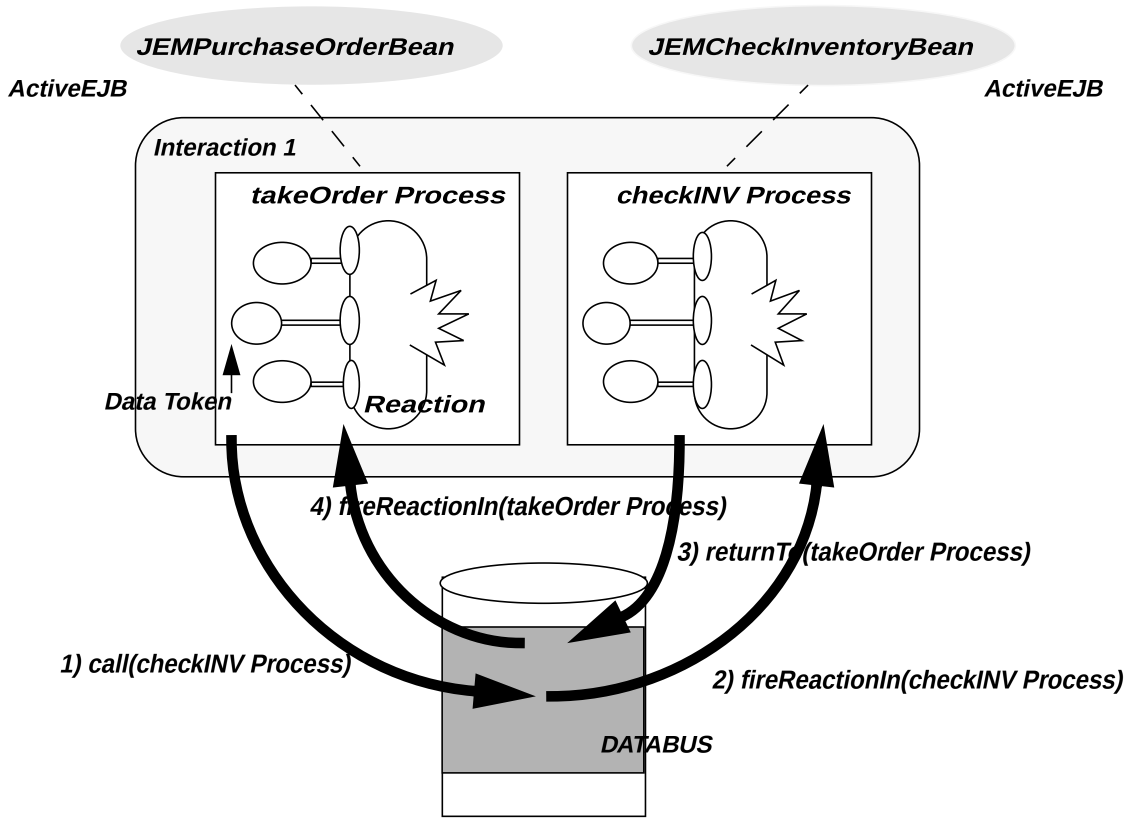
- Active EJBs: An Active EJB contains the business logic. An Active EJB business object (stateless session or entity bean) is instantiated and its method is invoked when a Reaction fires.
- Interactions: An Interaction is a long-lived unit of work that reflects the behavior of a business transaction. It groups a series of data exchanges (with asynchronous, concurrent, and request/response characteristics) between Processes.
- Processes: A Process represents a business task. It encapsulates the units of work—Reactions—which perform the detailed work of a business task.
- Reactions: A Reaction performs the detailed work of a business task. It is used to do the following:
  - push data to and pull data from the Databus

  - process service requests
  - request service from other Active EJBs
  - return results to the caller Active EJB business task or to the application client
  - enforce business constraints that preserve the consistency of a business transaction
  - provide application restartability in case of failures

- Data Tokens: A Data Token describes a request for service or a response from a service request or an exception condition, such as an expiration of a timer.
- Databus: The Databus is the fundamental component in AC4J. Applications, attach to the Databus to exchange data and request services. The Databus is responsible for routing and matching of data tokens with registered Reactions and enables transparent load-balancing of the attached application.

Figure 8–1 demonstrates the relationship of these components to each other. The sections following describe each component.

***Figure 8–1 Relationship of Databus, Interactions, ActiveEJBs, Processes, Data Tokens and Reactions***



## Active EJBs

An EJB provides a natural way for describing business object—such as a customer, a purchase order, or an invoice. The externally visible business tasks of a business object, which is accessible by other applications, is separated from their internal implementation details and are described in the EJB interface.

Traditional EJBs are passive, they must be ready to immediately service a request from a client and return results quickly. Failure to deliver on these promises causes an EJB to be unusable. AC4J allows standard stateless session and entity EJBs to become active. Active EJBs permit requests for service to be de-coupled from the actual service execution. The policies that control when and which EJB method(s) are actually invoked are controlled by the service provider EJB. This de-coupling permits service request and service providers to interact as autonomous peers.

An application can create or lookup a `JEMHandle` and then request service from a business task, which is exposed in the EJB interface.

An Active EJB is uniquely identified by a `JEMHandle` object. A `JEMHandle` object encapsulates the Active EJB name, the J2EE application name, the EJB JAR name, the EJB name, the EJB bean class name, the EJB home interface name, the EJB remote interface name, the instance name (SID) of the database in which the Databus resides on, and the primary key of the EJB that is available only for entity beans.

## Interactions

An Interaction is a long-lived unit of work that reflects the behavior of a business transaction. A business transaction may span multiple applications that reside in different organizations. Contrary to the life of a local or a global transaction, the duration of these business transactions in this disconnected environment, can be long.

The Interaction represents a business goal that you want to complete. For example, if a customer wants to buy something from a business, the entire actions necessary to allow the customer to pay for and receive the item he/she wants is characterized as an Interaction. The Interaction groups a series of business data exchanges by providing the global execution context of the business transaction.

These applications may run in isolation and commit/rollback their own data without knowledge of other applications. However these applications should not be considered as different pieces, because the relationships formed amongst them must be coordinated and their consistency maintained. When a business transaction becomes inconsistent, its participating applications may need to recover. The application recovery can be obtained by registering compensating Reaction(s). For example, once the supplier has confirmed the purchase order request back to the buyer, the buyer needs to register a compensating reaction that monitors additional responses from the supplier that may inform him that the purchase order cannot be fulfilled because the manufacturing department is running late. If the supplier's promise is cancelled, then the buyer's compensating reaction is matched and then fired to allow the buyer application to recover its application consistency. This reaction can pick a new supplier and request the item from him or abandon the purchase order process completely.

An Interaction is uniquely identified by an Interaction identifier (IID). An Interaction can contain multiple Processes.

## Processes

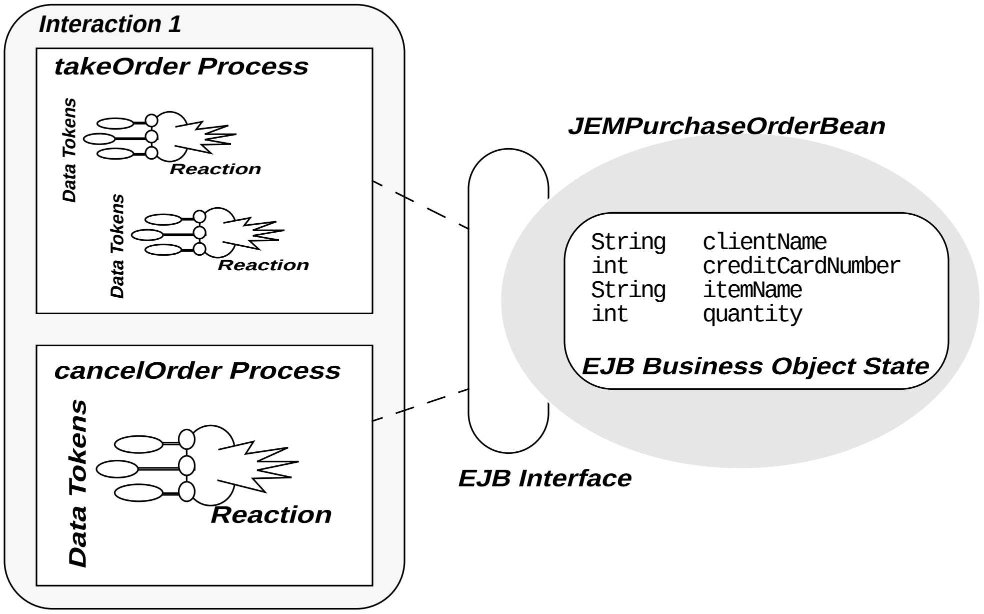
A Process identifies a business task. In our purchase order example, a Process would exist for each of the following business tasks: creating a purchase order, checking inventory, checking customer credit, and shipping the order.

Each Process does the following:

- Encapsulates the Reactions which perform its detailed work.
- Encapsulates data tokens, which contain the business task input parameters and its responses.
- Maintains the data flow context that determines how to return the response to the caller business task.

Figure 8–2 demonstrates an Active EJB, Interaction, and two of its Processes.

***Figure 8–2 Relationship of Active EJB, Interaction, and its Processes***



A Process is uniquely identified by a `JEMPortHandle` object. A `JEMPortHandle` object encapsulates the Process Context and the `JEMHandle` of the Active EJB that the Process belongs to. The Process Context is an union of an Interaction identifier, and the Process Activation identifier. AC4J automatically creates the Interaction and Process Activation identifiers within a `call` operation. Alternatively, the application can supply them in the AC4J `JEMSession::call` operation.

## Reactions

A Reaction performs the detailed work of a Process. Using this construct, an application can specify its persistence interest on the availability of a collection of correlated data tokens which trigger the execution of an Active EJB method. A Reaction is a combination of the following:

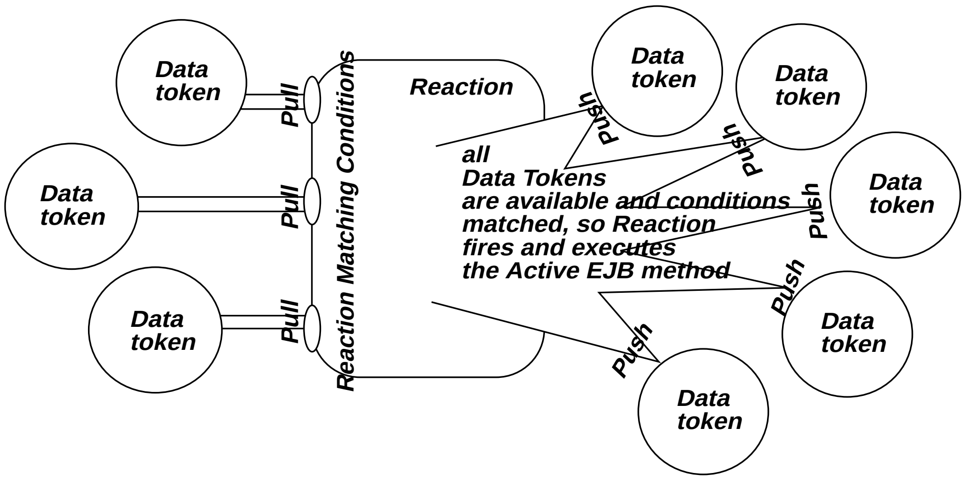
- A Reaction Template: A set of rules designating when the Reaction will match, what data tokens are required to be pulled before firing, and under what conditions the Reaction is allowed to fire.
- An Active EJB method: The method is executed when the Reaction fires.

In every Process, there is a base Reaction, which is implicitly created by AC4J when a Process is created as a result of a AC4J `call` operation. Additionally, an application can explicitly create a Reaction at run time using the `JEMReaction::registerReaction` operation to synchronize on data tokens. The implicit or explicit `registerReaction` operation specifies the Reaction template and the Active EJB method to be executed when matching succeeds.

Reactions (EJB methods) can access / modify shared database objects. These objects can be traditional database objects; thus, facilitating *coarse grain information sharing* in a transactional manner. Similarly, the Reactions exchange *fine grain information*—such as Active EJB method input parameters and return values—using the AC4J Databus.

The Reaction processes incoming requests, can return results based on the request, and enforces business constraints to preserve application consistency. When a Reaction is fired, it can consume one or more input data parameters, process them, and then possibly produce one or more output data tokens for other Reactions. Figure 8–3 demonstrates how when all data tokens are available and the conditions are matched, the Reaction fires, which causes the method to execute. This method may return results which are converted to data tokens by AC4J infrastructure and routed to the caller. This method may request additional services from other Active EJBs to complete the business task. These requests result in the creation of new data tokens that are pushed and routed by the AC4J Databus.

***Figure 8–3 Reaction Pulled Data Tokens, Were Matched, Fired, and had its Active EJB Method Execute***


Data token
Data token
Data token
Pull
Pull
Pull
Reaction Matching Conditions
Reaction
all Data Tokens are available and conditions matched, so Reaction fires and executes the Active EJB method
Push
Push
Push
Push
Push
Data token
Data token
Data token
Data token
Data token

Reactions inside a Process Context instance can push data tokens to the AC4J Databus in the following ways:

- by issuing one or more `JEMReaction::call` operations that request service from other Processes in the same or different Interaction context instance
- by returning or throwing an exception operations to the caller Processes
- by registering a timer, using the `JEMReaction::registerReactionTimer` operation

  When the timer expires, AC4J pushes a time-out exception data token in the current Reaction context instance.

Reactions inside a Process context instance can pull data tokens from the AC4J Databus by registering one or more Reactions in the current Process Context instance using `JEMReaction::registerReaction` method.

One or more Reactions can exist for each business task. A Reaction is used for the request and another for the response to support the asynchronous nature in a request/response environment. The number of Reactions depends on the number of requests and responses necessary.

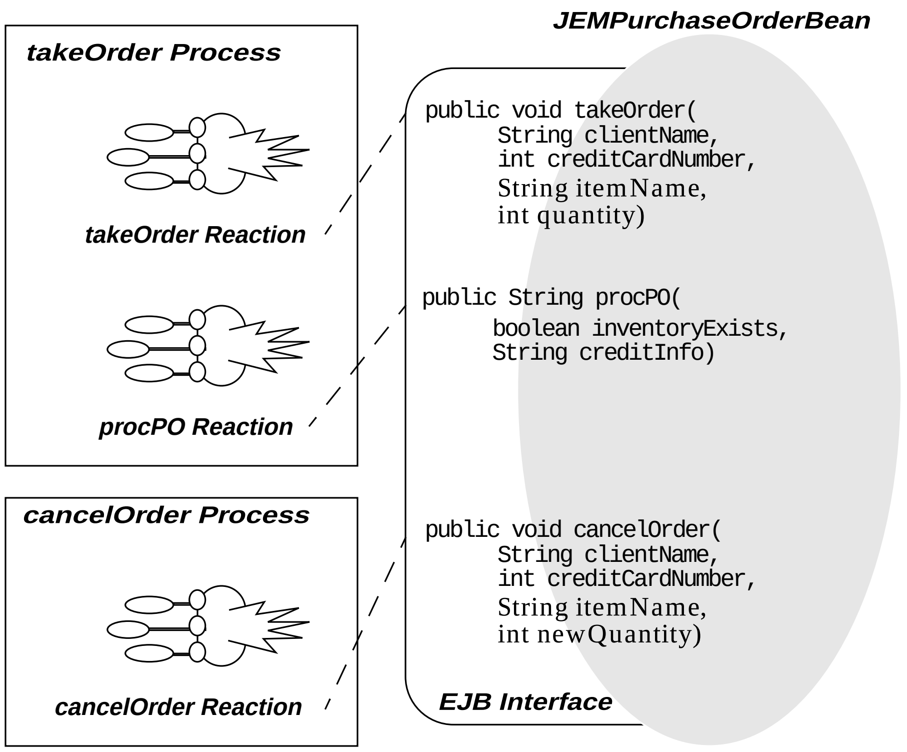
The following example demonstrates how one can receive an asynchronous communication between processes, but still have a request/response environment.

The `takeOrder` Process is the business task for creating the purchase order. In order to create the purchase order, you must check the inventory and the customer's credit. Thus, the `takeOrder` Reaction invokes the following Processes:

- `checkINV`—Under the conditions that the customer asks for a new purchase and provides the data of the items wanted, the `checkINV` Process is activated and its `JEMInventoryBean` Active EJB is instantiated and its base Reaction, `checkINV`, reacts. Later, it returns its results to the `takeOrder` Process and its `JEMPurchaseOrderBean` Active EJB.
- `checkCRED`—This Process is activated and its `JEMCreditBean` Active EJB is instantiated and its base Reaction, `checkCRED`, reacts to check the customer's credit. Later, it returns its results to the `takeOrder` Process and its `JEMPurchaseOrderBean` Active EJB.

After sending the asynchronous requests to the `checkINV` and `checkCRED` Processes, the `takeOrder` Reaction registers another Reaction in the same Process—`procPO`—that waits for the responses back from both the `checkCRED` and `checkINV` Processes. Once all data tokens expected from these Processes are available, the `procPO` Reaction fires and processes the responses. As shown in Figure 8–4, both the `takeOrder` and `procPO` Reactions exist in the same process, as they are components of the same request/response communication.

***Figure 8–4 Relationship Of JEMPurchaseOrderBean Interface Methods, Its takeOrder Process (With Its takeOrder And procPO Reactions) And Its cancelOrder Process (With Its cancelOrder Reaction)***



```
public void takeOrder(
      String clientName,
      int creditCardNumber,
      String itemName,
      int quantity)

public String procPO(
      boolean inventoryExists,
      String creditInfo)

public void cancelOrder(
      String clientName,
      int creditCardNumber,
      String itemName,
      int newQuantity)
```

---

**Note:** In order to satisfy the AC4J requirement of not locking resources, the call should be an asynchronous AC4J call. However, you can still perform synchronous EJB calls to another bean.

---

## Data Tokens

The activation of a Reaction is triggered by the availability of data tokens. Availability is defined by the arrival of one or more data tokens, with the right conditions, and the right access mode.

When an application is requesting a service by using an AC4J call operation the system automatically pushes a request data token, which consists of the following:

- a Process descriptor, which specifies the service that is requested (such as, `takeOrder`)
- a request `JEMPortHandle` object of the service provider to whom the request is destined
- a response `JEMPortHandle` object, which contains the Process Context (Interaction and Process-Activation identifiers) instance and the `JEMHandle` of the requester Process that will later receive the results from the service provider
- business task input arguments, which are used by the service provider to honor the service

Later when a Reaction returns a response data token that is automatically generated by AC4J when an active EJB returns or throws an exception, AC4J fills in the routing information needed for sending the returned information to the caller Process and fills the port handle object of the response data token. In the case where the caller of the returning Process is a client and not another Process, then the Databus stores the response data token to a special Databus area from where the client can retrieve it using the `JEMSession::receiveReactionResponseObjectInstance` operation.

The data types of the objects carried inside an in or out data tokens can be basic data types (such as Integer, String, Float, Boolean) or constructed class types (such as Java serializable objects).

## Databus

Improving the autonomy, scalability, and availability of applications requires components requesting services to be unaware of the identity, location, and the number of components that provide these services. In AC4J, applications are attached to a Databus before starting their operation. The AC4J Databus is responsible for routing and matching of data tokens that are pushed and need to be pulled by registered Reactions. Additionally, it enables scheduling, activation, and execution of the matched Reactions.

### Matching Reactions

The Databus routing subsystem is responsible for making the different types of data tokens available at the specified destination, Process Context instance that is a union of { Interaction identifier and Process Activation identifier }, specified by a `JEMPortHandle` object.

When data tokens are routed and become available in the Databus inside a Process Context instance, AC4J tries to match these data tokens with all registered Reaction(s) available in that context instance. The system tries to match the data token tags specified in a Reaction template, evaluating all constraint conditions against the matched data tokens to filter and discard the inappropriate ones.

Availability of some data token(s) does not mean that a registered Reaction will match immediately. It is only when all data tokens, required by a Reaction, become available that matching succeeds. For example, inside `takeOrder` Process the `takeOrder` base Reaction has registered the `procPO` Reaction that is the waiting for the `checkCRED` and `checkINV` Processes to respond. When `checkINV` Process responds to the `takeOrder` Process, `procPO` Reaction is not matched because it is also waiting for the `checkCRED` Process to respond. It is when the `checkCRED` Process responds to the `takeOrder` Process that the `procPO` Reaction is matched.

Additionally, data token(s) available in the Databus, may be matched with a Reaction that will be registered in the future. This can be used for sequencing Processes, where the completion of one Process can enable another Process. Inside the same Interaction, the `takeOrder` Process must be completed before `cancelOrder` Process can start executing. If `takeOrder` Process has not completed but `cancelOrder` Process is requested from a client, its base Reaction, which is implicitly created by the system, will not be matched because it is waiting for the completion data token of the `takeOrder` Process to be available. If `takeOrder` Process has completed (already pushed its completion data token), then `cancelOrder` Process is requested from a client and it will be immediately matched, because the completion data token of the `takeOrder` Process is already available.

Matching data tokens with Reactions triggers the activation of zero, one, or more Reactions, which are executed in parallel if they don't conflict for shared resources.

## Firing Reactions

Each method of the remote interface of an Active EJB implements the application business logic. When the data tokens become available, and matched with a Reaction, AC4J verifies that the types (primitive or class types) of the data tokens matched on the tags, also match the types of the Reaction Active EJB method types. Then, AC4J verifies that the matched Reaction is authorized to pull the available matched data tokens. If everything passes successfully, then AC4J schedules the activation of the Reaction.

When the matched Reaction is fired, the AC4J container begins a JTA transaction and instantiates the requested Active EJB (stateless session bean or entity EJB) using the primary key inside the `JEMHandle` request object. Then the EJB method, of the fired Reaction, gets executed using the matched data tokens of the Reaction.

AC4J automatically commits the current Reaction at the end of every Active EJB method. A Reaction commit marks the end of a JTA transaction, so that all its changes to shared data tokens and all its service requests / responses that have been sent become visible. The activation of a Reaction has "exactly once" semantics, if the Reaction commits. If a failure occurs after a commit, then the Reaction cannot be rolled back and the changes will persist. If a failure occurs before or during a commit, then the container rolls back the current Reaction. A Reaction rollback reverses all changes to shared data tokens and the service requests / responses are never sent to any recipient component. In case of failures, the firing of a Reaction will be retried by the Databus for a pre-configured number of attempts. The Reaction is marked as completed, with exception completion status, if the maximum retry attempts are reached.

In traditional database machines, where the duration of a transaction is short, abnormal situations cause the whole transaction to be undone, so all performed work is lost and needs to be submitted again for execution. Since Interactions have usually long duration and contain a large number of Reaction(s), AC4J provides additional mechanisms to handling exceptions (such as an Oracle9*i*AS node crash or an Oracle database node crash).

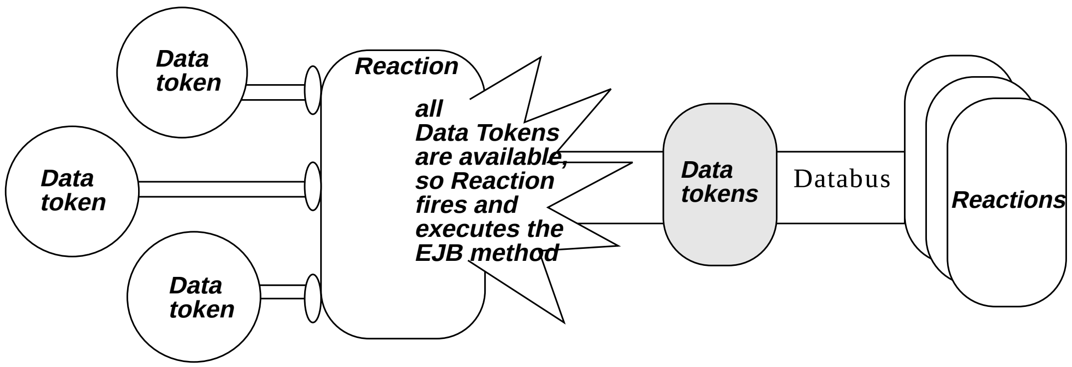
A Reaction is automatically persisted in the Databus by AC4J if it completes successfully. The state that is saved (Process input variable data, Process local variable data, and data flow context information) can be used to continue the application with minimum restart time from the last Reaction. When a node crashes, all Reactions that were running and did not end successfully are rolled back. Then, the interrupted Reactions will be re-executed by another OC4J instance.

AC4J uses a mechanism to capture, propagate, and match the application state and control flow information needed for resuming an application after the crash. Additionally, because Reaction execution is data-driven, there is no need for the system to keep a volatile or persistent copy of the entire program state (such as program execution stack) in order to facilitate the storage of the control flow descriptors or the storage of data variables.

## Relationships of Databus, Data Tokens, and Reactions

Figure 8–5 demonstrates how data tokens cause Reactions to fire, and Reactions send new data tokens to other Reactions over the Databus. The Databus coordinates and matches the data tokens with its Reactions.

***Figure 8–5 Databus, Data Tokens, and Reactions***


Data token
Data token
Data token
Reaction
all Data Tokens are available, so Reaction fires and executes the EJB method
Data tokens
Databus
Reactions

Once the method completes, the Reaction can send information in the form of a data token to another Reaction. All data tokens are sent asynchronously from one Reaction to another over a data channel known as the AC4J Databus. The AC4J Databus routes the data tokens from a producer Reaction to one or more consumer Reactions.

## Set Up Oracle Database For AC4J Support

Before you can execute any Interactions, you must initialize an Oracle9*i* database as a repository for the AC4J Databus. You must configure it to include the following:

- AC4J connection and session capabilities
- AC4J system tablespace
- AC4J super user
- AC4J Databus'
- One or more client users

These can be added to your Oracle9*i* database with scripts that are contained in the `ac4j-sql.jar` file that was downloaded with your Oracle9*i*AS installation. Unzip this JAR file. This JAR file contains a README.TXT that discusses the different SQL command options that are available to you. These are also described below:

In order to create AC4J capabilities, you must execute one of the following SQL scripts as a 'SYS' user on the same machine as the database.

- `createall`: To create all of the defaults including the default Databus, AC4J super user, default client user (`JEMCLIUSER`).
- `createjemtablespace`: To create the table space for AC4J system, execute the `createjemtablespace` SQL script. You must provide the `SYS` username/password, the `TNSENTRY` of this database where the Databus is created.
- `createjem`: To install and create the Databus, execute the `createjem` SQL script. This requires the `SYS` username/password, `TNS_ENTRY`, and an AC4J client username.
- `createclient`: To create another client on an existing Databus, execute the `createclient` SQL script. Provide the SYS username/password, client username/password, and client tablespace.
- `recreatedatabus`: To recreate an existing Databus, which deletes the existing Databus and all its contents and then re-creates it, execute the `recreatedatabus` script. Provide the `SYS` username/password and `TNSENTRY` of the database where the Databus resides.
- `recreateclient`: To recreate an existing client, execute the `recreateclient` SQL script. Provide the SYS username/password and the client username/password.

## AC4J Databus XML Configuration

The Interaction supports JTA global transactions within the database that the Databus exists in. Thus, you need a non-emulated data source for the super user to handle the two-phase commit and a non-emulated data-source for the client to send its asynchronous requests to the Databus. See the DataSource and JTA Chapters in the *Oracle9iAS Containers for J2EE Services Guide* for a full description of this configuration.

For our purchase order example, the following data sources are configured in the `data-sources.xml` file for the two-phase commit.

```
<!--NON-Emulated DataSource for two-phase commit used by super user-->
<data-source
             class="com.evermind.sql.OrionCMTDataSource"
             location="jdbc/jemSuperuserDS"
             username="jemuser"
             password="jempasswd"
             url="jdbc:oracle:thin:@<host>:<port>:<ORCL-SID>"
```

```
                inactivity-timeout="60" >
                <property name="dblink"
                    value="JEMLOOPBACKLINK.REGRESS.RDBMS.DEV.US.ORACLE.COM" />
</data-source>

<!--NON-Emulated DataSource for the client user -->
<data-source
                class="com.evermind.sql.OrionCMTDataSource"
                location="jdbc/jemClientDS"
                username="jemcliuser"
                password="jemclipasswd"
                url="jdbc:oracle:thin:@<host>:<port>:<ORCL-SID>"
                inactivity-timeout="60" >
                <property name="dblink"
                   value="JEMLOOPBACKLINK.REGRESS.RDBMS.DEV.US.ORACLE.COM" />
</data-source>
```

Both of these users were created as default with the SQL scripts listed earlier. The *jemuser* is the super username and the *jemcliuser* is the default client username. The DBLINK is the link to the database that contains the Databus. For the super user data source, this is a loopback link.

## AC4J Example

AC4J is designed for complex applications that interact with each other over long periods of time. This section illustrates the usage of AC4J with a portion of the purchase order example listed in Figure 7-6. The code sample does not show error handling or import statements to simplify the example. Download the full example at `http://otn.oracle.com/sample_code/tech/java/oc4j/htdocs/oc4jsamplecode/oc4j-demo-ejb.html` off the OTN site.

***Example 8–1    Purchase Order Example***

For the purchase order, the `POInteraction` is created. Within the Interaction, several business tasks exist as follows:

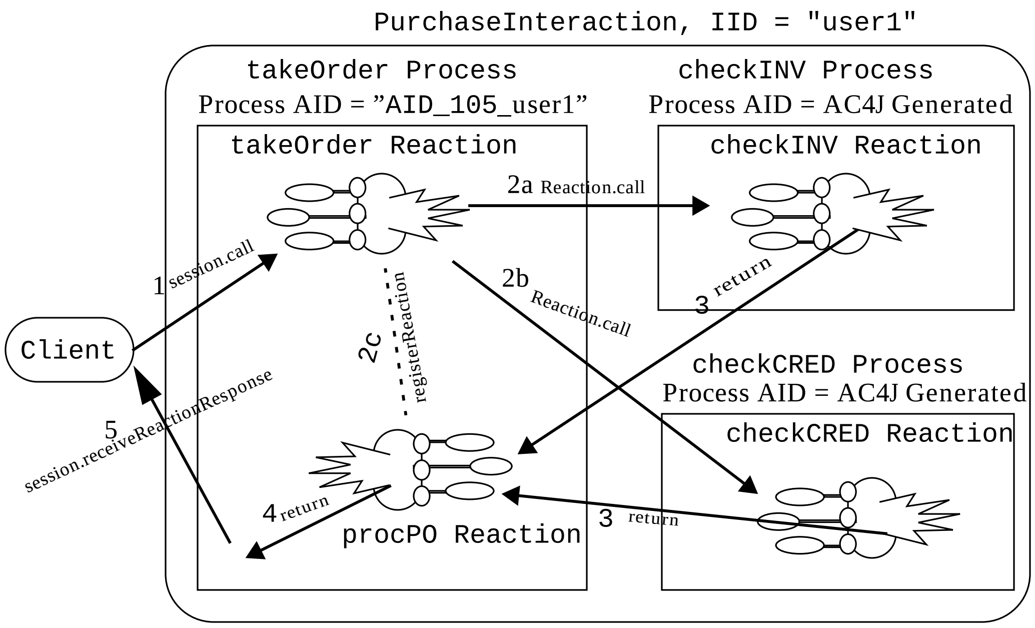
- create purchase order (`takeOrder` Process)
- check inventory (`checkINV` Process)
- check customer credit (`checkCRED` Process)
- process responses from previous checks requests (`procPO` Reaction)

Our example is, as follows:

- `takeOrder` Reaction, which pushes 2 data tokens:
  - A data token that asks the `checkINV` Process if the inventory contains the desired items.
  - A data token that asks the `checkCRED` Process if the credit card given by the customer is able to make the purchase.
- `procPO`, which acts on the responses from the inventory and credit check Processes. If the inventory is available and the credit check goes well, then the `procPO` returns the purchase order confirmation to the client.

Figure 8–6 illustrates the information flow inside an Interaction. Figure 8–6 also demonstrates how all of the Reactions act on data tokens and provide data tokens to other Processes. This assumes that the customer data has already been made available to the `takeOrder` Process. The numbers designate the order in which they fire. That is, the `procPO` is dependent on data tokens from both the `checkINV` and `checkCRED` Processes; thus, it cannot fire until both return their responses back to the `takeOrder` Process.

***Figure 8–6 Information Flow Inside An Interaction***



The steps involved in processing the Purchase Order example of Figure 8–6 can be summarized as follows:

1. *Client sends an asynchronous request to an Active EJB:* The client requests a service from an Active EJB, `JEMPurchaseOrderBean`. The client starts a new purchase order by sending an asynchronous request through the Databus to a `takeOrder` Process.
2. *Active EJB processes the client's request:* The `takeOrder` Process starts a `takeOrder` base Reaction. This base Reaction starts a new purchase order. To complete the purchase order, it must do three things:
   a. Send an asynchronous request to the `checkINV` Process of `JEMInventoryBean` to verify that the items are in inventory.
   b. Send an asynchronous request to the `checkCRED` Process of `JEMCreditBean` to verify that the customer's credit is satisfactory.
   c. Register a `procPO` Reaction in the current Process to receive the results from the above two Processes.

3. *Asynchronous response to the requesting Active-EJB:* Both the `checkINV` and `checkCRED` Processes return responses to the `takeOrder` Process.
4. *Asynchronous response to the client:* The `procPO` Reaction, within `takeOrder` Process, reacts to the information provided by the `checkINV` and `checkCRED` Processes. If satisfactory, it sends the confirmation to the client through the AC4J Databus.
5. *Client receives the response:* Client retrieves the response from the Databus.

## Asynchronous Request to An Active EJB

The following code sample shows steps involved in performing loosely-coupled interactions in AC4J.

***Example 8–2 Client Asynchronously Invoking Active EJB***

```
public static void main(String[] args) throws  ClassNotFoundException, Exception
{
    // 0. create a JNDI context
    Context  context = new  InitialContext();

    // 1. lookup a datasource wher Databus exists
    DataSource clientDS = (DataSource)
                          context.lookup("java:comp/env/jdbc/jemClientDS");
    // 2. Get a JDBC-connection to the database where Databus resides
    Connection conn = clientDS.getConnection("jemcliuser", "jemclipasswd");

    // 3. Create an AC4J connection using the JDBC connection
    JEMConnection AC4JConn = new  JEMConnection(conn);

    // 4. Create an AC4J session over an AC4J connection to the Databus
    JEMSession AC4JSess = new  JEMSession(jemconn);
    // 5. Lookup the Active EJB handle using the jem-name defined
    //    in the orion-ejb-jar.xml
    JEMHandle activeEJBHandle =
    (JEMHandle)context.lookup("JEMPurchaseOrderBean");

    // 6. Gather the base Reaction input parameters. These input parameters are
    //    required by the receiving method, takeOrder.
    Object[] inputParams = new  Object[] { (Object) new  String("user1"),
                                           (Object) new  Integer("1234-119"),
                                           (Object) new  String("pens"),
                                           (Object) new  Integer("3")   };
    // 7. Create the Process Context, Interaction-ID and Activation ID.
    //    NOTE: IID = "user1" = requestor's name
    //          AID = AID = AID_105_user1 = AID_<PO_number>_<cust_name>

    // 8. Make the call over the AC4J session providing the parameters.
    JEMEmitToken req = AC4JSess.call("user1", "AID_105_user1",
                                     activeEJBHandle, "takeOrder",
                                     null, inputParams, null, 0, 0);

    // 9. Commit the changes to the Databus by committing the transaction
    conn.commit();

    // 10. The client must close the AC4J session and connection because it
    //     does not exist within an AC4J container, which would normally
    //     close these.
    conn.close();
    jemconn.close();
    jemsess.close();
}
```

The client exists outside of an AC4J server and is requesting a service from an Active EJB through the AC4J Databus. The AC4J Databus is the conduit and controls the asynchronous communication between the client and all Reactions. Thus, every client residing outside of an AC4J server must first connect to the AC4J Databus and create a new session for interaction to occur.

After you have retrieved a connection to the AC4J Databus and created an AC4J session within it, you can send asynchronous messages to Active EJBs in the same or other AC4J instances. The AC4J Databus coordinates the asynchronous messages and acts as a transactional manager for all AC4J beans involved in the transaction. The steps involved in creating a AC4J-session and completing the client's request are described in the following subsection.

### Connect To the AC4J Databus

The following steps explain the details of a set of steps involved in creating a AC4J session on the AC4J Databus. These steps are sub-set of steps shown in Example 8–2 (Numbered 0-to-4).

1. Retrieve an AC4J connection

   An AC4J connection exists above a JDBC connection. Perform the following:

   a. Retrieve the `DataSource` defined for the database acting as the AC4J conduit. The `DataSource` used should be defined in the `data-sources.xml` as a non-emulated datasource with a `<dblink>` defined to the database where the AC4J Databus resides. See "AC4J Databus XML Configuration" on page 8-21 for more information.

```
Context context = new InitialContext();
DataSource clientDS = (DataSource)
               context.lookup("java:comp/env/jdbc/jemClientDS");
```

b. Retrieve the JDBC connection off of the `DataSource` object.

```
OracleConnection conn = (OracleConnection)clientDS.getConnection();
```

c. Create an AC4J connection off of the JDBC connection object.

```
JEMConnection AC4JConn = new JEMConnection(conn);
```

2. Create an AC4J session in a specified Databus. Using the AC4J connection to the database and providing the name of the Databus you are interested in, create a session within the Databus in the indicated Oracle database.

```
JEMSession AC4JSess = new  JEMSession(AC4Jconn);
```

## Executing an Asynchronous Request

After you have created an AC4J session on the AC4J Databus, the client can send asynchronous messages to Active EJBs. The client must provide the Active EJB handle, the Process handle, and all of the required input parameters to the base Reaction. The following steps explain the details of the `call` that client needs to make in order to complete the AC4J request.

1. *Process Context:* To identify the context where the Process exists, you must provide both the Interaction identifier and the Process Activation identifier. The combination of both of these identifiers is the Processing Context. There are two ways of providing a Process Context:

   - CLIENT PROVIDES—The AC4J Databus uses the identifiers provided by the client to uniquely identify the Processing Context. The client uses the same identifiers to either retrieve the response to the current request or send additional parameters to the Process. In the current example, the client provides the Interaction Identifier (IID) as a customer's name and Process Activation Identifier (P-AID) as an union of purchase order number and the customer's name as shown:

```
String iid = "user1";         // = customer_name
String p_aid = "AID_105_user1"; // = AID_<PO_number>_<customer_name>
JEMEmitToken req = AC4JSess.call(iid, p_aid, ..all other parameters..);
```

- AUTOMATIC CONTEXT—The Interaction and Process Activation identifiers are optional and can be omitted or can be null, in which case the system automatically creates them. If a client fails to provide either of these identifiers then the AC4J Databus will create them to uniquely identify a processing context. However, the client will have to retrieve these identifiers and use them later to pull the response from the AC4J Databus.

  ```
  JEMEmitToken req =
          AC4JSess.call (null, null, ...all other parameters...);
  JEMPortHandle portHandle = req.getPortHandle();
  String iid = portHandle.getIid();
  String p_aid = portHandle.getAid();
  ```

2. *Active EJB handle:* In a synchronous EJB environment, you would use a remote EJB handle for invocation. In an AC4J asynchronous environment, you must provide a similar handle of class type `JEMHandle` that identifies an active EJB. The Active EJB handle can be obtained by looking up the `jem-name` defined in the `orion-ejb-jar.xml` (see Active EJB Deployment pp 7-32).

   ```
   Context context = new InitialContext();
   JMHandle activeEJBHandle =
            (JEMHandle) context.lookup("JEMPurchaseOrderBean");
   JEMEmitToken req = AC4JSess.call(...., activeEJBHandle, ....);
   ```

3. *Reaction Name and Input parameters:* Client provides the base Reaction (Method) name and all or part of it's input parameters that it wishes to call. In the current example, the client provides all the input parameters to complete the AC4J session **call** as follows:

   ```
   // collect input values for the takeOrder method
   Object[] inputParams = new Object[] { (Object) new String("user1"),
                                         (Object) new Integer("1234-123"),
                                         (Object) new String("pens"),
                                         (Object) new Integer("3")
                                       };
   JEMEmitToken req = AC4JSess.call(..., "takeOrder", null, inputParams,
                                   ...);
   ```

   - If client provides only a part of the parameters to this Reaction then it must provide a set of input parameter types and the indexes of the input parameters as well. In the following example we show how a client can complete a call by providing the first two parameters for takeOrder Process:

```
// input parameter types (java-class) for takeOrder method
Class[] takeOrderInputClassTypes =
            new Class[] { String.class, Integer.TYPE,
                          String.class, Integer.TYPE };

// indexes of input parameters you wish to provide for takeOrder method
int[] indexOfInputParams = new int[] {0, 1};

// input values corresponding to the indexes for takeOrder method
Object[] inputParams = new Object[] { (Object) new String("user1"),
                                        (Object) new Integer("1234-123")
                                      };
// remember the interaction and process-activation ids for this call
JEMEmitToken req =
       AC4JSess.call(iid, p_aid, ..., "takeOrder",
                            takeOrderInputClassTypes,
                            indexOfInputParams, inputParams, ...);
```

- When the client decides to provide the remaining two parameters it must use the same Process Context (Interaction and Process-Activation Identifiers) that it used in the first **call** it made to the Process. During the second invocation the steps involved will be

```
// input parameter types (java-class) for takeOrder method
Class[] takeOrderInputClassTypes =
            new Class[] { String.class, Integer.TYPE,
                          String.class, Integer.TYPE };

// indexes of input parameters you wish to provide for takeOrder method
// NOTE: now, client provides the last 2-input parameters
int[] indexOfInputParams = new int[] {2, 3};

// input values corresponding to the indexes for takeOrder method
Object[] inputParams = new Object[] { (Object) new String("pens"),
                                        (Object) new Integer("3")
                                      };
// use the same interaction and process activation ids as those in the
// previous call
JEMEmitToken req =
       AC4JSess.call(iid, p_aid, ..., "takeOrder",
                            takeOrderInputClassTypes,
                            indexOfInputParams, inputParams, ...);
```

4. *AC4J Session call:* Send all asynchronous requests for any Active EJB to the AC4J Session using the `JEMSession::call` method.

When a Reaction wants to provide data to an active EJB method (to the base Reaction of the Process), it executes a **`JEMSession::call`** with this information. The **`JEMSession::call`** contains the Interaction identifier that the EJB is involved in, the Process Activation identifier to identify the Process where the method is instantiated, and the `JEMHandle` of the active EJB. The Interaction and Process Activation identifier are optional and can be omitted or can be null, in which case the system automatically creates them. The Databus identifies the context where the Process can be found and routes the data tokens to the intended Process. Thus, all EJB calls are invoked asynchronously through the mediation of the Databus.

5. *Commit Transaction:* The client must commit the changes to the AC4J Databus. If the client forgets to commit the transaction then the request is lost and is not visible to the AC4J Databus. To make the request visible to the AC4J Databus by doing the JDBC-commit as follows:

   ```
   conn.commit();
   ```

6. Finally, the client must close the JDBC-connection, the AC4J session and connection because it does not exist within an AC4J container. The AC4J container would normally close the AC4J session and connection objects.

   ```
   conn.close(); // client as well an application-code must close
   jemconn.close(); // client must close
   jemsess.close(); // client must close
   ```

## Active EJB processes the Client's Request

Once the client commits the request, the AC4J Databus matches the data-tokens provided by the client with that of the requested Reaction, and internally schedules the instantiation of the `JEMPurchaseOrderBean` Active EJB and activation of the `takeOrder` Process. The `takeOrder` Process starts a `takeOrder` base Reaction which starts a new purchase order. As discussed in Figure 7-6, this reaction, `takeOrder`, processes the client's request by invoking additional services from the other Active EJBs, `JEMInventoryBean` and `JEMCreditBean`, as shown in the following code sample:

***Example 8–3 Active EJB Asynchronously Invoking Another Active EJB***

```
public void takeOrder(String clientName, int creditCardNumber,
                     String itemName, int quantity)
      throws RemoteException, TestException
{
```

```
    // 0. create a JNDI context
    Context context = new InitialContext();

    // 1. Retrieve the current AC4J Reaction.
    JEMReaction currentAC4JReaction = (JEMReaction) JEMReaction.getReaction();

    // 2. Lookup the Active EJB handle using the jem-name defined
    //    in the orion-ejb-jar.xml
    JEMHandle activeInvHandle = (JEMHandle) context.lookup("JEMInventoryBean");

    // 3. Gather all input and return parameters for the checkINV Reaction.
    //    Define input and return parameter types and the parameter values
    Object[] checkINVInputParamValues =
                new Object[] { (Object)itemName,
                               (Object) new Integer (quantity) };
    Class[] checkINVReturnClassType = new Class[] { Boolean.TYPE };

    // 4. Request a service from JEMBeancheckINV through Databus
    JEMEmitToken inventoryRequest=
                   currentAC4JReaction.call(activeInvHandle, "checkINV", null,
                                            checkINVInputParamValues,
                                            checkINVReturnClassType,
                                            null, null, 0,  0);
    // 5. Repeat Steps 2-4 above to request a service from another
    //    Active EJB, JEMCreditBean. The returned JEMEmitToken is
    //    named creditRequest.

    // 6. Register a Reaction, procPO, that will be activated when the
    //    responses from the above two asynchronous calls to the
    //    active-EJBs return
    Class[] procPOInputClassTypes = new  Class[] { Boolean.TYPE, String.class };
    JEMEmitToken[] requests = new JEMEmitToken[] { inventoryRequest,
                                                   creditRequest };
    currentJEMReaction.registerReaction
                 ("procPO", procPOInputClassTypes, requests, 1, null, null, 0);
}
```

The AC4J Databus instantiates the Active EJB, JEMPurchaseOrderBean (corresponding to the JEMHandle provided by the client), in an AC4J server. The takeOrder Process starts a takeOrder base Reaction. The steps involved in the completion of this initiation process are described below:

1. *Process Context:* The current Reaction, takeOrder, is running in an AC4J server. Hence, it already has a Process Context and can be used by the application (or

Active Bean) code. The application code can retrieve the Process Context through the Demarcation as follows:

```
// retrieve the current-Reaction context--a static method
JEMReaction currentAC4JReaction = (JEMReaction) JEMReaction.getReaction();
String iid = currentAC4JReaction.getIid();
String p_aid = currentAC4JReaction.getAid();
```

- The application may use these identifiers to make additional asynchronous **JEMSession::call** by co-relating the business transaction.
- Alternatively, the application code may use the `currentAC4JReaction` to make the additional calls with request/response characteristics. The AC4J Databus then creates a new Process Contexts for the next invocation by using the current Interaction Identifier and a new Process Activation Identifier. The current example uses this approach by using the `currentAC4JReaction`.

2. *AC4J handle:* The base Reaction, `takeOrder,` starts the purchase order initiation process by requesting services from two other Active EJBs, `JEMInventoryBean` and `JEMCreditBean`. The application code needs to retrieve the AC4J handles to these Active EJBs by doing the following:

```
Context context = new InitialContext();
// call to JEMInventoryBean
JEMHandle activeInvHandle = (JEMHandle) context.lookup("JEMInventoryBean");
JEMEmitToken inventoryRequest=
                currentAC4JReaction.call(activeInvHandle, .....);

// call to JEMCreditBean
JEMHandle activeCreditHandle = (JEMHandle) context.lookup("JEMCreditBean");
JEMEmitToken creditRequest=
                currentAC4JReaction.call(activeCreditHandle, .....);
```

3. *Reaction Name, Return parameter type and Input parameters:* Client (now a `takeOrder` Reaction) provides the base Reaction (Method) name, the return parameter's java-class type and all or part of it's input parameters that it wishes to call. In the current example, the client provides all the input parameters needed by the called Reactions (checkINV, CheckCRED) as follows:

```
// collect input values for the checkINV method
Object[] checkINVInputParamValues =
            new Object[] { (Object)itemName,
                           (Object) new Integer (quantity)
                         };
```

```
// state the return Class type of checkINV method
Class[] checkINVReturnClassType = new Class[] { Boolean.TYPE };

// make the call to the checkINV method
JEMEmitToken inventoryRequest=
               currentAC4JReaction.call(..., "checkINV", null,
                                        checkINVInputParamValues,
                                        checkINVReturnClassType, ....);

// collect input values for the checkCRED method
Object[] checkCreditInputParamValues =
            new Object[] { (Object) clientName,
                           (Object) new Integer (creditCardNumber),
                           (Object) new Float (quantity * 1.4) };

// state the return Class type of checkINV method
Class[] checkCreditReturnClassType = new Class[] { String.class };

// make the call ro checkCRED method
JEMEmitToken creditRequest=
               currentAC4JReaction.call(..., "checkCRED", null,
                                        checkCreditInputParamValues,
                                        checkCreditReturnClassType, ....);
```

4. *Register a Return Reaction:* The application code then registers a new Reaction, `procPO`, in the same Process Context of the `currentAC4JReaction`. This registration of the Reaction requires the Reaction name, `procPO`, the input parameter types of the new `procPO` reaction and the `JEMEmitTokens` retrieved from the call to the `currentAC4JReaction.` If the new Reaction has multiple input parameters and is receiving them from different Processes then the Array of `JEMEmitToken` must be constructed in proper order. For example, in the following code the first parameter is waiting for the reply from the `JEMInventoryBean` and the second one is waiting for the reply from `JEMCreditBean`.

```
Class[] procPOInputClassTypes = new  Class[] { Boolean.TYPE, String.class };
JEMEmitToken[] requests = new JEMEmitToken[] { inventoryRequest,
                                               creditRequest };
currentJEMReaction.registerReaction
             ("procPO", procPOInputClassTypes, requests, 1, null, null, 0);
```

## Asynchronous Response to the Requesting Active EJB

The `takeOrder` base Reaction is completed only after the AC4J infrastructure commits the transaction which includes the calls to the other two Active EJBs and a registered Reaction. The “checkINV” and “checkCRED” Processes receive the requests from the AC4J Databus as if it were invoked from any other EJB. The JEMInventoryBean and JEMCreditBean Active EJBs are instantiated. The checkINV and checkCRED base Reactions are fired when they receive the data tokens from the AC4J Databus, which were initiated from the takeOrder Reaction. Both of them receive the request, perform their tasks, and return. The returned values are forwarded by the AC4J Databus to the registered Reaction—procPO.

The following code sample shows the checkINV method. The checkCRED method is similar in its AC4J responsibilities.

***Example 8–4 checkINV Processes Request***

```
public boolean checkINV(String itemName, int quantity)
            throws RemoteException, TestException
{
    boolean inventoryExists = false;
    // The logic in the next step is ommitted
    inventoryExists = query it’s own database for the item and quantity;
    return inventoryExists;
}
```

## Asynchronous Response to the Client

Both checkINV and checkCRED Processes return the responses to the procPO Reaction through the AC4J Databus. The AC4J Databus makes sure that the return data-tokens have valid takeOrder Process Context and matches the input parameter types of the procPO Reaction. When both parameters arrive the procPO Reaction fires and executes the procPO method of the JEMPurchaseOrderBean Active EJB, which reacts to the information provided by the checkINV and checkCRED Processes. It completes the client’s request by posting the result to the AC4J Databus.

***Example 8–5 procPO Reaction Fires***

```
public String procPO(boolean inventoryExists, String creditInfo)
            throws RemoteException, TestException
{
    String poStatus = "Not Shipped";
    if(creditInfo == null)
        return poStatus;
    if (inventoryExists)
    {
        if(creditInfo.equalsIgnoreCase("Credit approved"))
```

```
            poStatus = "Shipped";
        else if (creditInfo.equalsIgnoreCase("Credit failed"))
            poStatus = "Credit failed";
    }
    else
        poStatus = "Items unavailable";

    return poStatus;
}
```

## Receive Response by the Client

The client needs to know the response to it's purchase order request. As stated earlier, each request (or **call**) is identified by a Process Context (Interaction ID and Activation ID). Using the Process Context the client can pull the response from the AC4J Databus.

The received `JEMEmitToken` from the response can then be parsed by the client. If the client existed inside the OC4J container, the container would deconstruct the `JEMEmitToken` to the required type. Instead, the client must parse out the response correctly as shown below:

***Example 8–6 Client Processes Return***

```
public static void main(String[] args) throws  ClassNotFoundException, Exception
{
    // 0. create a JNDI context
    Context  context = new  InitialContext();

    // 1. lookup a datasource wher Databus exists
    DataSource clientDS = (DataSource)
                          context.lookup("java:comp/env/jdbc/jemClientDS");

    // 2. Get a JDBC-connection to the database where Databus resides
    Connection conn = clientDS.getConnection("jemcliuser", "jemclipasswd");

    // 3. Create an AC4J connection using the JDBC connection
    JEMConnection AC4JConn = new  JEMConnection(conn);

    // 4. Create an AC4J session over an AC4J connection to the Databus
    JEMSession AC4JSess = new  JEMSession(jemconn);

    // 5. Lookup the Active EJB handle using the jem-name defined
    //    in the orion-ejb-jar.xml
    JEMHandle activeEJBHandle =
                (JEMHandle) context.lookup("JEMPurchaseOrderBean");

    // 6. Retrieve the Response using the Process context with which
    //    the initial request was made.
    JEMEmitToken  rcvresp = AC4JSess.receiveReactionResponse
              ("user1", "AID_105_user1", activeEJBHandle, "takeOrder", 0);

    // 7. The getReactionResponseObjectInstance method parses the returned
    //    parameter into an java.lang.Object.
    Object obj = rcvresp.getReactionResponseObjectInstance();
    // 8. Print out results
    if (obj  instanceof  java.lang.String)
        String  ret = (String)  obj;

    // 9. The client must commit the transaction
    conn.commit();

    // 10. The client must close the AC4J session and connection because it
    //     does not exist within an AC4J container, which would normally
    //     close these.
    conn.close();
    jemsess.close();
    jemconn.close();
}
```

As seen earlier, procPO reaction reacts to the information provided by the `checkINV` and `checkCRED` Processes. It completes the client's request by posting the result to the AC4J Databus. The client must connect to the AC4J Databus to retrieve it's response by providing a proper Process Context. The steps involved in connecting to the AC4J Databus were described earlier. After receiving the response the client can retrieve a `java.lang.Object` instance which must be processed further.

## Retrieving an Asynchronous Response

After creating a AC4J session on the AC4J Databus, the client can retrieve the response by doing the following steps:

1. *Process Context:* The client must provide a proper Process Context that identifies where the request was made. The client must provide both the Interaction identifier and the Process Activation identifier. In the current example the client provides the Interaction Identifier (IID) as a customer's name and Process

Activation Identifier (P-AID) as an union of purchase order number and the customer's name as shown:

```
String iid = "user1";         // = customer_name
String p_aid = "AID_105_user1"; // = AID_<PO_number>_<customer_name>
JEMEmitToken  rcvresp = AC4JSess.receiveReactionResponse
          (iid, p_aid, ...);
```

2. *Active EJB handle:* The client must provide the Active EJB handle to which the initial request was made. The Active EJB handle can be obtained by looking up the `jem-name` defined in the `orion-ejb-jar.xml` (see "AC4J Active EJB Deployment" on page 8-37).

```
Context context = new InitialContext();
JMHandle activeEJBHandle =
          (JEMHandle) context.lookup("JEMPurchaseOrderBean");
JEMEmitToken  rcvresp = AC4JSess.receiveReactionResponse
          (..., activeEJBHandle, ...);
```

3. *Reaction Name:* Client may need to provide the process name to which it initiated the call, which, in this case, is the `takeOrder` Process.

```
JEMEmitToken  rcvresp = AC4JSess.receiveReactionResponse
          (..., "takeOrder", ...);
```

4. *Retrieve Object:* The `JEMEmitToken` received from the `receiveReactionResponse` can be used to retrieve the Object instance as follows:

```
Object obj = rcvresp.getReactionResponseObjectInstance();
```

5. *Commit Transaction:* The client must commit the changes to the AC4J Databus. If the client forgets to commit the transaction then the client can pull the response multiple times. However, it is not a recommended mode of operation. To let the AC4J Databus know that the response was properly retrieved do the following:

```
conn.commit();
```

## AC4J Active EJB Deployment

The active EJB is developed as any other EJB. The changes that enable the EJB to be used in an AC4J Interaction is in the OC4J-specific deployment descriptor. These are discussed below:

Deploy the EJB with AC4J element specifications in the OC4J-specific deployment descriptor. The following example defines the `takeOrder` EJB as an active EJB.

- The `<jem-server-extension>` element defines the database with the Databus that the active EJBs in this JAR file use for their AC4J communication.

  ```
  <jem-server-extension data-source-location="jdbc/jemSuperuserDS">
    <description>AC4J datasource location</description>
  </jem-server-extension>
  ```

- The `<jem-deployment>` element in the `orion-ejb-jar.xml` file identifies the EJB defined in the `ejb-jar.xml` file as an active EJB. It provides an AC4J name (`jem-name`) that is used to identify the bean within the AC4J calls. For example, this bean is defined as `JEMPurchaseOrderBean`, which was used in the `JEMHandle` creation. The identity of the caller, who is allowed to request services and retrieve responses from the Active EJB, can be declared in the `called-by` tag. This `caller` tag identifies the user in the Databus. For example, `JEMCLIUSER` is the user name that was used to create a `jem-session`,

  ```
  <jem-deployment jem-name="JEMPurchaseOrderBean"
                  ejb-name="PurchaseOrderBean">
    <description>AC4J EJB</description>
    <called-by>
      <caller caller-identity="JEMCLIUSER"/>
    </called-by>
  </jem-deployment>
  ```

The following is the entire `orion-ejb-jar.xml` file for the three Active EJBs.

```
<?xml version="1.0"?>
<!DOCTYPE orion-ejb-jar PUBLIC "-//Evermind//DTD Enterprise JavaBeans 1.1
runtime//EN" "http://www.orionserver.com/dtds/orion-ejb-jar.dtd">

<orion-ejb-jar deployment-version="1.4.5" deployment-time="e60dffcea9">
<enterprise-beans>
    <jem-server-extension
        data-source-location="jdbc/nonEmulatedDS"
        scheduling-threads="1">
        <description>AC4J deployment</description>
    </jem-server-extension>

    <jem-deployment jem-name="JEMPurchaseOrderBean"
                     ejb-name="PurchaseOrderBean">
        <description>Active Purchase Order bean</description>

        <called-by>
            <caller  caller-identity="JEMCLIUSER"/>
```

```
            </called-by>

            <security-identity>
                <description>using the caller identity </description>
                <use-caller-identity>true</use-caller-identity>
            </security-identity>
        </jem-deployment>

        <jem-deployment jem-name="JEMInventoryBean"
                        ejb-name="InventoryBean">
            <description>Active Inventory bean</description>

            <called-by>
                <caller  caller-identity="JEMCLIUSER"/>
            </called-by>

            <security-identity>
                <description>using the caller identity </description>
                <use-caller-identity>true</use-caller-identity>
            </security-identity>
        </jem-deployment>

        <jem-deployment jem-name="JEMCreditBean"
                        ejb-name="CreditBean">
            <description>Active Credit bean</description>

            <called-by>
                <caller  caller-identity="JEMCLIUSER"/>
            </called-by>

            <security-identity>
                <description>using the caller identity </description>
                <use-caller-identity>true</use-caller-identity>
            </security-identity>
        </jem-deployment>

    </enterprise-beans>
```

# A
# OC4J-Specific DTD Reference

This appendix describes the elements contained within the OC4J-specific EJB deployment descriptor: `orion-ejb-jar.dtd`. Some of the elements within this deployment descriptor are too complex for this appendix, so references to another source may be mentioned.

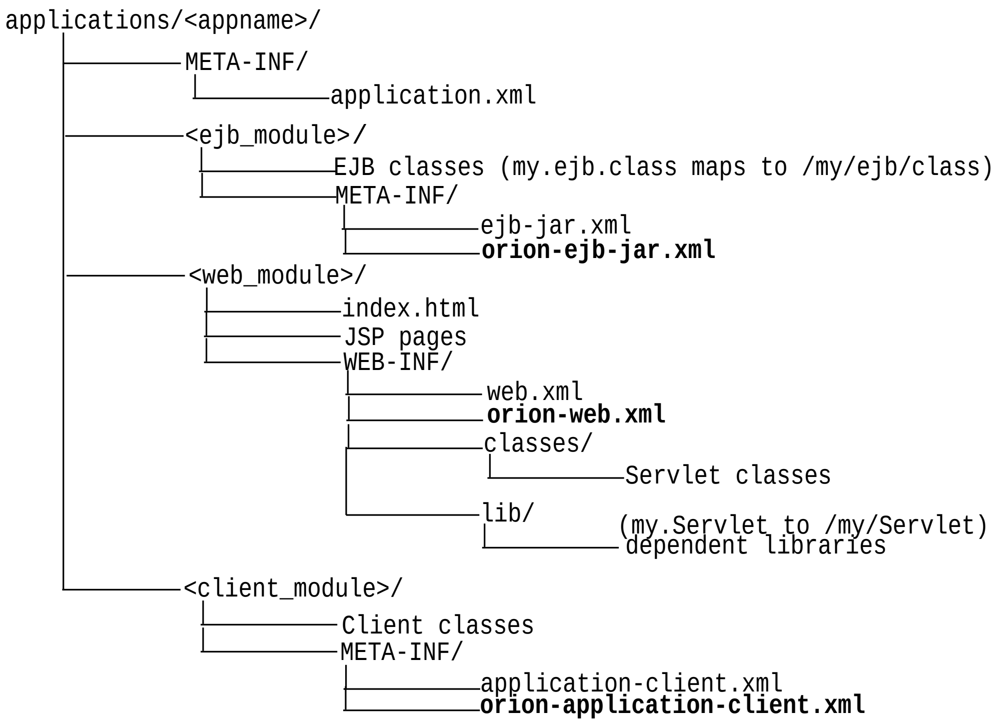
The description of this deployment descriptor has been divided into the following sections:

- Overall description of section order—The required ordering of this XML file is described in "OC4J-Specific Deployment Descriptor for EJBs" on page A-2.
- DTD listing—The official DTD, without comments, is shown in "DTD Listing" on page A-9. This is printed as a quick reference on the number of elements required within another element.
- Element description—An alphabetical listing and description for each element is discussed in "Element Description" on page A-12.

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

Oracle recommends that you add your OC4J-specific XML files within the recommended development structure as shown in Figure A–1.

***Figure A–1 Development Application Directory Structure***



## OC4J-Specific Deployment Descriptor for EJBs

The OC4J-specific deployment descriptor contains extended deployment information for session beans, entity beans, message driven beans, and security for these EJBs. The major element structure within this deployment descriptor has the following structure:

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

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

- Enterprise Beans Section
- Assembly Descriptor Section

## Enterprise Beans Section

The `<enterprise-beans>` section defines additional deployment information for all EJBs: session beans, entity beans, and message driven beans. There is a section for each type of EJB.

The following sections describe the elements within `<enterprise-beans>` element;

- Session Bean Section
- Entity Bean Section
- Message Driven Bean Section
- CMP Field Mapping Section
- Method Definition

### Session Bean Section

The `<session-bean>` section provides additional deployment information for a session bean deployed within this JAR file. The `<session-bean>` section contains the following structure:

```
<session-deployment call-timeout=... copy-by-value=... location=...
      max-tx-retries=... name=... persistence-filename=... timeout=...
      wrapper=...
  <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>
</session-deployment>
```

### Entity Bean Section

The `<entity-bean>` section provides additional deployment information for an entity bean deployed within this JAR file. The `<entity-bean>` section contains the following structure:

```
<entity-deployment call-timeout=... clustering-schema=... copy-by-value=...
      data-source=... exclusive-write-access=... isolation=... location=...
      max-tx-retries=... name=... table=... validity-timeout=... wrapper=...>
  <primkey-mapping>
   <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
      persistence-type=...></cmp-field-mapping>
  </primkey-mapping>
  <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
      persistence-type=...> </cmp-field-mapping>
  <finder-method partial=... query=... >
   <method></method>
  </finder-method>
  <env-entry-mapping name=...></env-entry-mapping>
  <ejb-ref-mapping location=... name=... />
  <resource-ref-mapping location=... name=... >
    <lookup-context location=...>
       <context-attribute name=... value=... />
    </lookup-context>
  </resource-ref-mapping>
</entity-deployment>
```

### Message Driven Bean Section

The `<message-driven-bean>` section provides additional deployment information for a message driven bean deployed within this JAR file. The `<message-driven-bean>` section contains the following structure:

```
<message-driven-deployment connection-factory-location=...
      destination-location=... name=...>
  <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>
</message-driven-deployment>
```

## 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=...>
  <description></description>
</jem-server-extension>
```

The `<jem-deployment>` section provides additional deployment information for an active EJB deployed within this JAR file. The `<jem-deployment>` section contains the following structure:

```
 <jem-deployment jem-name=... ejb-name=...>
   <description></description>
   <called-by>
     <caller  caller-identity=.../>
   </called-by>
   <security-identity>
    <description></description>
    <use-caller-identity></use-caller-identity>
   </security-identity>
 </jem-deployment>
```

## CMP Field Mapping Section

The mapping of logical names to actual names can be a complex process. See "Object-Relational Mapping of Persistent Fields" on page 3-13 for a discussion on mapping CMP data fields.

The following are the XML elements used for CMP persistent data field mapping within the `orion-ejb-jar.xml` file:

```
<cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
      persistence-type=...>
   <fields>
      <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
          persistence-type=...></cmp-field-mapping>
   </fields>
   <properties>
      <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
            persistence-type=...></cmp-field-mapping>
   </properties>
   <entity-ref home=...>
      <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
```

```
       persistence-type=...></cmp-field-mapping>
</entity-ref>
<list-mapping table=...>
   <primkey-mapping>
      <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
          persistence-type=...></cmp-field-mapping>
   </primkey-mapping>
   <value-mapping immutable="true|false" type=...>
      <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
          persistence-type=...></cmp-field-mapping>
   </value-mapping>
</list-mapping>
<collection-mapping table=...>
   <primkey-mapping>
      <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
          persistence-type=...></cmp-field-mapping>
   </primkey-mapping>
   <value-mapping immutable="true|false" type=...>
      <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
          persistence-type=...></cmp-field-mapping>
   </value-mapping>
</collection-mapping>
<set-mapping table=...>
   <primkey-mapping>
      <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
          persistence-type=...></cmp-field-mapping>
   </primkey-mapping>
   <value-mapping immutable="true|false" type=...>
      <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
          persistence-type=...></cmp-field-mapping>
   </value-mapping>
</set-mapping>
<map-mapping table=...>
   <primkey-mapping>
      <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
          persistence-type=...></cmp-field-mapping>
   </primkey-mapping>
   <map-key-mapping type=...>
      <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
          persistence-type=...></cmp-field-mapping>
   </map-key-mapping>
   <value-mapping immutable="true|false" type=...>
      <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
          persistence-type=...></cmp-field-mapping>
   </value-mapping>
   </map-mapping>
</cmp-field-mapping>
```

## Method Definition

The following structure is used to specify the methods (and possibly parameters of that method) of the bean.

```
<method>
   <description></description>
   <ejb-name></ejb-name>
   <method-intf></method-intf>
   <method-name></method-name>
   <method-params>
     <method-param></method-param>
   </method-params>
</method>
```

The style used can be one of the following:

1. When referring to all the methods of the specified enterprise bean's home and remote interfaces, specify the methods as follows:

   ```
   <method>
        <ejb-name>EJBNAME</ejb-name>
             <method-name>*</method-name>
    </method>
   ```

2. When referring to multiple methods with the same overloaded name, specify the methods as follows:

```
<method>
 <ejb-name>EJBNAME</ejb-name>
        <method-name>METHOD</method-name>
 </method>>
```

3. When referring to a single method within a set of methods with an overloaded name, you can specify each parameter within the method as follows:

   ```
   <method>
   <ejb-name>EJBNAME</ejb-name>
          <method-name>METHOD</method-name>
              <method-params>
                     <method-param>PARAM-1</method-param>
                     <method-param>PARAM-2</method-param>
   ```

```
                        ...
                    <method-param>PARAM-n</method-param>
              </method-params>
    <method>
```

## Assembly Descriptor Section

In addition to specifying deployment information for individual beans, you can also specify addition deployment 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>
```

## DTD Listing

The following lists the `orion-ejb-jar.xml` DTD to show the ordering required, and optional parameters for each element. The definitions for each element is described in "Element Description" on page A-12.

```
<!ELEMENT properties (cmp-field-mapping*)>
<!ELEMENT fields (cmp-field-mapping*)>
<!ELEMENT session-deployment (env-entry-mapping*, ejb-ref-mapping*,
  resource-ref-mapping*)>
<!ATTLIST session-deployment call-timeout CDATA #IMPLIED
  copy-by-value CDATA #IMPLIED
  location CDATA #IMPLIED
  max-tx-retries CDATA #IMPLIED
  name CDATA #IMPLIED
  persistence-filename CDATA #IMPLIED
  timeout CDATA #IMPLIED
  wrapper CDATA #IMPLIED
  replication CDATA #IMPLIED>
<!ELEMENT collection-mapping (primkey-mapping, value-mapping)>
<!ATTLIST collection-mapping table CDATA #IMPLIED>
<!ELEMENT resource-ref-mapping (lookup-context?)>
<!ATTLIST resource-ref-mapping location CDATA #IMPLIED
  name CDATA #REQUIRED>
<!ELEMENT method-intf (#PCDATA)>
<!ELEMENT entity-ref (cmp-field-mapping)>
<!ATTLIST entity-ref home CDATA #IMPLIED>
<!ELEMENT enterprise-beans ((session-deployment | entity-deployment |
  message-driven-deployment | jem-deployment)+, jem-server-extension?)>
<!ELEMENT ejb-ref-mapping (#PCDATA)>
<!ATTLIST ejb-ref-mapping location CDATA #IMPLIED
  name CDATA #REQUIRED>
<!ELEMENT primkey-mapping (cmp-field-mapping)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT env-entry-mapping (#PCDATA)>
<!ATTLIST env-entry-mapping name CDATA #IMPLIED>
<!ELEMENT security-role-mapping (group*, user*)>
<!ATTLIST security-role-mapping impliesAll CDATA #IMPLIED
  name CDATA #IMPLIED>
<!ELEMENT method-params (method-param*)>
<!ELEMENT cmp-field-mapping
  (fields|properties|entity-ref|list-mapping|collection-mapping|set-mapping|
  map-mapping|field-persistence-manager)?>
<!ATTLIST cmp-field-mapping ejb-reference-home CDATA #IMPLIED
  name CDATA #IMPLIED
  persistence-name CDATA #IMPLIED
  persistence-type CDATA #IMPLIED>
<!ELEMENT list-mapping (primkey-mapping, value-mapping)>
<!ATTLIST list-mapping table CDATA #IMPLIED>
<!ELEMENT group (#PCDATA)>
<!ATTLIST group name CDATA #IMPLIED>
<!ELEMENT default-method-access (security-role-mapping)>
<!ELEMENT map-key-mapping (cmp-field-mapping)>
<!ATTLIST map-key-mapping type CDATA #IMPLIED>
<!ELEMENT map-mapping (primkey-mapping, map-key-mapping, value-mapping)>
<!ATTLIST map-mapping table CDATA #IMPLIED>
<!ELEMENT value-mapping (cmp-field-mapping)>
<!ATTLIST value-mapping immutable CDATA #IMPLIED
  type CDATA #IMPLIED>
<!ELEMENT method-param (#PCDATA)>
<!ELEMENT user (#PCDATA)>
<!ATTLIST user name CDATA #IMPLIED>
<!ELEMENT lookup-context (context-attribute+)>
<!ATTLIST lookup-context location CDATA #IMPLIED>
<!ELEMENT context-attribute (#PCDATA)>
<!ATTLIST context-attribute name CDATA #IMPLIED
  value CDATA #IMPLIED>
<!ELEMENT set-mapping (primkey-mapping, value-mapping)>
<!ATTLIST set-mapping table CDATA #IMPLIED>
<!ELEMENT message-driven-deployment (env-entry-mapping*, ejb-ref-mapping*,
  resource-ref-mapping*)>
<!ATTLIST message-driven-deployment cache-timeout CDATA #IMPLIED
  connection-factory-location CDATA #IMPLIED
  destination-location CDATA #IMPLIED
  max-instances CDATA #IMPLIED
  min-instances CDATA #IMPLIED
  name CDATA #IMPLIED>
<!ELEMENT jem-server-extension (description?, data-bus?)>
<!ATTLIST jem-server-extension
  data-source-location CDATA #REQUIRED
  scheduling-threads CDATA #IMPLIED>
<!ELEMENT data-bus EMPTY>
<!ATTLIST data-bus
  data-bus-name CDATA #REQUIRED
  url CDATA #IMPLIED>
<!ELEMENT jem-deployment (description?, data-bus?, called-by,
  security-identity)>
<!ATTLIST jem-deployment
  jem-name CDATA #REQUIRED
  ejb-name CDATA #REQUIRED>
```

```
<!ELEMENT called-by (caller+)>
<!ELEMENT caller EMPTY>
<!ATTLIST caller
  caller-identity CDATA #REQUIRED>
<!ELEMENT security-identity
  (description?,(use-caller-identity|run-as-specified-identity))>
<!ELEMENT use-caller-identity EMPTY>
<!ELEMENT run-as-specified-identity (description?, role-name)>
<!ELEMENT role-name (#PCDATA)>
<!ELEMENT ejb-name (#PCDATA)>
<!ELEMENT field-persistence-manager (property)>
<!ATTLIST field-persistence-manager class CDATA #IMPLIED>
<!ELEMENT property (#PCDATA)>
<!ATTLIST property name CDATA #IMPLIED
  value CDATA #IMPLIED>
<!ELEMENT finder-method (method)>
<!ATTLIST finder-method partial CDATA #IMPLIED
  query CDATA #IMPLIED>
<!ELEMENT method (description?, ejb-name, method-intf?, method-name,
  method-params?)>
<!ELEMENT entity-deployment (primkey-mapping?, cmp-field-mapping*,
  finder-method*, env-entry-mapping*, ejb-ref-mapping*, resource-ref-mapping*)>
<!ATTLIST entity-deployment call-timeout CDATA #IMPLIED
  clustering-schema CDATA #IMPLIED
  copy-by-value CDATA #IMPLIED
  data-source CDATA #IMPLIED
  exclusive-write-access CDATA #IMPLIED
  instance-cache-timeout CDATA #IMPLIED
  location CDATA #IMPLIED
  isolation (commited | serializable | uncommited | repeatable_reads)
    CDATA #IMPLIED
  locking-mode (pessimistic | optimistic | read-only | old_pessimistic)
  max-instances CDATA #IMPLIED
  min-instances CDATA #IMPLIED
  max-instances-per-pk CDATA #IMPLIED
  min-instances-per-pk CDATA #IMPLIED
  max-tx-retries CDATA #IMPLIED
  update-changed-fields-only (true | false) "true"
  name CDATA #IMPLIED
  pool-cache-timeout CDATA #IMPLIED
  table CDATA #IMPLIED
  validity-timeout CDATA #IMPLIED
  wrapper CDATA #IMPLIED>
<!ELEMENT orion-ejb-jar (enterprise-beans, assembly-descriptor)>
<!ATTLIST orion-ejb-jar deployment-time CDATA #IMPLIED
```

```
  deployment-version CDATA #IMPLIED>
<!ELEMENT assembly-descriptor (security-role-mapping*, default-method-access?)>
<!ELEMENT method-name (#PCDATA)>
```

## Element Description

**<assembly-descriptor>**

The mapping of the assembly descriptor elements.

**<cmp-field-mapping>**

Deployment information for a container-managed persistence field. If no subtags are used to define different behavior, the field is persisted through serialization or native handling of "recognized" primitive types.

Attributes:

- ejb-reference-home - The JNDI-location of the fields remote EJB-home if the field is an entity EJBObject or an EJBHome.
- name - The name of the field.
- persistence-name - The name of the field in the database table.
- persistence-type - The database type (valid values varies from database to database) of the field.

**<collection-mapping>**

Specifies a relational mapping of a Collection type. A Collection consists of n unordered items (order isnt specified and not relevant). The field containing the mapping must be of type java.util.Collection.

Attiributes:

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

**<context-attribute>**

An attribute sent to the context. The only mandatory attribute in JNDI is the ’`java.naming.factory.initial`’ which is the classname of the context factory implementation.

Attributes:

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

**<default-method-access>**

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

**<description>**

A short description.

**<ejb-name>**

The ejb-name element specifies an enterprise bean's name. This name is assigned by the ejb-jar file producer to name the enterprise bean in the ejb-jar file's deployment descriptor. The name must be unique among the names of the enterprise beans in the same ejb-jar file. The enterprise bean code does not depend on the name; therefore the name can be changed during the application-assembly process without breaking the enterprise bean's function. There is no architected relationship between the ejb-name in the deployment descriptor and the JNDI name that the Deployer will assign to the enterprise bean's home. The name must conform to the lexical rules for an NMTOKEN.

**<ejb-ref-mapping>**

The `ejb-ref` element that is used for the declaration of a reference to another enterprise bean's home. The `ejb-ref-mapping` element ties this to a JNDI-location when deploying.

Attributes:

- location - The JNDI location to look up the EJB home from.
- name - The ejb-ref's name. Matches the name of an ejb-ref in ejb-jar.xml.

**<enterprise-beans>**

The beans contained in this EJB JAR file.

**<entity-deployment>**

Deployment information for an entity bean.

Attributes:

- call-timeout - The time (long milliseconds in decimal) to wait for an EJB if it is busy (before throwing a `RemoteException`, treating it as a deadlock). This is also used as a SQL query timeout. If the timeout occurs before the SQL query finishes, a SQL exception is thrown. If zero, the timeout is disabled. The default is 90 seconds.
- clustering-schema - The name of the data-source used if using container-managed persistence.

- copy-by-value - Whether or not to copy all the incoming/outgoing parameters for all incoming and outgoing EJB calls. Set to 'false' if your application does not assume copy-by-value semantics for these parameters. The default is 'true'.
- data-source - The name of the data source used if using container-managed persistence.
- disable-wrapper-cache - If true, a pool of wrapper instances is not maintained. The default is false.
- exclusive-write-access - Whether or not the EJB-server has exclusive write (update) access to the database backend. This can be used only for entity beans that use a "`read_only`" locking mode. In this case, it increases the performance for common bean operations and enables better caching. The default is false.
- instance-cache-timeout - The amount of time in seconds that entity wrapper instances are assigned to an identity. If you specify 'never', you retain the wrapper instances until they are garbage collected. The default is 60 seconds.
- isolation - Specifies the isolation-level for database actions. The valid values for Oracle databases are 'serializable' and 'committed'. The default is 'committed'. Non-Oracle databases can be the following: 'none', 'committed', 'serializable', 'uncommitted', and 'repeatable_read'.
- location - The JNDI-name this bean will be bound to.
- locking-mode - The concurrency modes configure when to block to manage resource contention or when to execute in parallel. The concurrency modes are as follows:
  - `PESSIMISTIC`: This manages resource contention and does not allow parallel execution. Only one user at a time is allowed to execute the entity bean at a single time.
  - `OPTIMISTIC`: Multiple users can execute the entity bean in parallel. It does not monitor resource contention; thus, the burden of the data consistency is placed on the database isolation modes. This is the default.
  - `READ-ONLY`: Multiple users can execute the entity bean in parallel. The container does not allow any updates to the bean's state.
- max-instances - The number of maximum bean implementation instances to be kept instantiated or pooled. The default is 10.
- min-instances - The number of minimum bean implementation instances to be kept instantiated or pooled. The default is 0.

- max-instances-per-pk - The number of maximum wrapper instances to be kept instantiated or pooled. The default is 50.
- min-instances-per-pk - The number of minimum wrapper instances to be kept instantiated or pooled. The default is 0.
- max-tx-retries - The number of times to retry a transaction that was rolled back due to system-level failures. The default is 3. Consider setting to zero if using the serializable isolation level. Within a transaction, the container uses the max-tx-retries value of the first invoked bean within the transaction
- name - The name of the bean, this matches the name of a bean in the assembly descriptor (`ejb-jar.xml`).
- pool-cache-timeout - The amount of time in seconds that the bean implementation instances are to be kept in the "pooled" (unassigned) state, specifying 'never' retains the instances until they are garbage collected. The default is 60.
- table - The name of the table in the database if using container-managed persistence.
- validity-timeout - The maximum amount of time (in milliseconds) that an entity is valid in the cache (before being reloaded). Useful for loosely coupled environments where rare updates from legacy systems occur. This attribute is only valid for entity beans with locking mode of `read_only` and when exclusive-write-access="true" (the default).
- update-changed-fields-only - Specifies whether the container updates only modified fields or all fields to persistence storage for CMP entity beans when `ejbStore` is invoked. The default is true, which specifies to only update modified fields.
- wrapper - Name of the OC4J wrapper class for this bean. (internal server attribute, do not edit)

### <entity-ref>

Specified the configuration for persisting an entity reference via it's primary key. The child-tag of this tag is the specification of how to persist the primary key.

Attributes:

- home - JNDI location of the EJBHome to get lookup the beans at.

### <env-entry-mapping>

Overrides the value of an `env-entry` in the assembly descriptor. It is used to keep the EAR clean from deployment-specific values. The body is the value.

Attribute:

- name - The name of the context parameter.

**<fields>**

Specifies the configuration of a field-based (java class field) mapping persistence for this field. The fields that are to be persisted have to be public, non-static, non-final and the type of the containing object has to have an empty constructor.

**<finder-method>**

The definition of a container-managed finder method. This defines the selection criteria in a `findByXXX`() method in the bean's home.

Attributes:

- partial - Whether or not the specified query is a partial one. A partial query is the 'where' clause or the 'order' (if it starts with order) clause of the SQL query. Queries are partial by default. If partial="false" is specified then the full query is to be entered as value for the query attribute and you need to make sure that the query produces a result-set containing all of the CMP fields. This is useful when doing advances queries involving table joins and similar.
- query - The query part of an SQL statement. This is the section following the WHERE keyword in the statement. Special tokens are $number which denotes an method argument number and $name which denotes a cmp-field name. For instance the query for "findByAge(int age)" would be (assuming the cmp-field is named 'age'): "$1 = $age".

**<group>**

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

Attributes:

- name - The name of the group.

**<jem-deployment>**

Specifies an active EJB for deployment into the AC4J container.

Attributes:

- called-by - Provides the user identity of who will call this bean through the Databus.
- security-identity - describes if the Databus should use the caller or run-as identity.

**<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.

**<list-mapping>**

Specifies a relational mapping of a List type. A List is a sequential (where order/index is important) Collection of items. The field containing the mapping must be of type java.util.List or the legacy types java.util.Vector or Type[].

Attributes:

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

**<lookup-context>**

The specification of an optional `javax.naming.Context` implementation used for retrieving the resource. This is useful when using third party modules, such as a third party JMS server. Either use the context implementation supplied by the resource vendor or, if none exists, write an implementation that negotiates with the vendor software.

Attribute:

- location - The name looked for in the foreign context when retrieving the resource.

**<map-key-mapping>**

Specifies a mapping of the map key. Map keys are always immutable.

Attributes:

- type - The fully qualified class name of the type of the value. Examples are com.acme.Product, java.lang.String etc.

**<map-mapping>**

Specifies a relational mapping of a Map type. A Map consists of n unique keys and their mapping to values. The field containing the mapping must be of type java.util.Map or the legacy types java.util.Hashtable or java.util.Properties.

Attributes:

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

**<message-driven-deployment>**

Deployment information for a MDB.

Attributes:

- connection-factory-location - The JNDI location of the connection factory to use.
- destination-location - The JNDI location of the destination (queue/topic) to use.
- max-instances - The maximum number of bean instances to instantiate. The default is -1, which implies an infinite number.
- min-instances - The minimum number of bean instances to instantiate.
- name - The name of the bean, this matches the name of a bean in the assembly descriptor (`ejb-jar.xml`).

**<method>**

Specify the methods (and possibly parameters of that method) of the bean.

**<method-intf>**

The method-intf element allows a method element to differentiate between the methods with the same name and signature that are defined in both the remote and home interfaces. The method-intf element must be one of the following: Home or Remote.

**<method-name>**

The method-name element contains a name of an enterprise bean method, or the asterisk (*) character. The asterisk is used when the element denotes all the methods of an enterprise bean's remote and home interfaces.

**<method-param>**

The method-param element contains the fully-qualified Java type name of a method parameter.

**<method-params>**

The method-params element contains a list of the fully-qualified Java type names of the method parameters.

**<orion-ejb-jar>**

An `orion-ejb-jar.xml` file contains the OC4J-specific deployment information for an EJB. It is used to specify initial deployment properties. After each deployment the deployment file is reformatted and altered by the server for additional information.

Attributes:

- deployment-time - The time (long milliseconds in decimal) of the last deployment, if not matching the last editing date the jar will be redeployed. (internal server value, do not edit)
- deployment-version - The version of OC4J this jar was deployed with, if it's not matching the current version then it will be redeployed. (internal server value, do not edit)

**<primkey-mapping>**

Designates how the primary key is mapped.

**<properties>**

Specifies the configuration of a property-based (bean properties) mapping persistence for this field. The properties have to adhere to the usual JavaBeans specification and the type of the containing object has to have an empty constructor This is also designated within the EJB specification.

**<resource-ref-mapping>**

The `resource-ref` element is used for the declaration of a reference to an external resource such as a data source, JMS queue, or mail session. The `resource-ref-mapping` ties this to a JNDI-location when deploying.

Attributes:

- location - The JNDI location to look up the resource factory from.
- name - The resource-ref name. Matches the name of an resource-ref in `ejb-jar.xml`.

**<security-role-mapping>**

The runtime mapping (to groups and users) of a role. Maps to a security-role of the same name in the assembly descriptor.

Attributes:

- impliesAll - Whether or not this mapping implies all users. The default is false.
- name - The name of the role

**<session-deployment>**

Deployment information for a session bean.

Attributes:

- cache-timeout—How long to keep stateless sessions cached in the pool. Only applies to stateless session beans. The default is 60. Legal values are positive integer values or '`never`'.

- call-timeout—The time (long milliseconds in decimal) to wait for an EJB if it is busy. After this times out, a `RemoteException` is thrown and the EJB is treated as involved in a deadlock. If value is set to 0, OC4J waits for the EJB "forever". This is the default.
- copy-by-value—Whether or not to copy (clone) all the incoming and outgoing parameters in EJB calls. Set to 'false' if you are certain that your application does not assume copy-by-value semantics for a speed-up. The default is 'true'.
- location—The JNDI-name that this bean will be bound to.
- max-tx-retries—The number of times to retry a transaction that was rolled back due to system-level failures. The default is 3. Within a transaction, the container uses the max-tx-retries value of the first invoked bean within the transaction.
- name—The name of the bean, which matches the name of a bean in the assembly section of the EJB deployment descriptor (`ejb-jar.xml`).
- persistence-filename—Path to the file where sessions are stored across restarts.
- timeout—Inactivity timeout in seconds. If the value is zero or negative, then all timeouts are disabled. The default is 30 minutes.
- wrapper—Name of the OC4J wrapper class for this bean. This is an internal server value and should not be edited.

**<set-mapping>**

Specifies a relational mapping of a Set type. A Set consists of n unique unordered items (order is not specified and not relevant). The field containing the mapping must be of type java.util.Set.

Attributes:

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

**<user>**

A user that this security-role-mapping implies.

Attributes:

- name - The name of the user.

**<value-mapping>**

Specified a mapping of the primary key part of a set of fields.

Attributes:

- immutable - Whether or not the value can be trusted to be immutable once added to the `Collection/Map`. Setting this to true will optimize database operations extensively. The default value is "true" for set-mapping and map-mappings and "false" for collection-mapping and list-mapping.

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

# B
# Third Party Licenses

This appendix includes the Third Party License for all the third party products included with Oracle9*i* Application Server. Topics include:

- Apache HTTP Server
- Apache JServ

## Apache HTTP Server

Under the terms of the Apache license, Oracle is required to provide the following notices. However, the Oracle program license that accompanied this product determines your right to use the Oracle program, including the Apache software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the Apache software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or Apache.

### The Apache Software License

## Apache JServ

### Apache JServ Public License

# Index