
# EJB 2.1 Programmer's Guide

JOnAS Team ()

- Feb 2008 -

# Table of Contents

# List of Tables

# List of Examples

This guide explains how to program with EJB 2.1. To use the new EJB 3.0 beans, you should refer to EJB 3.0 Programmer's Guide[1]

[1] ejb3_programmer_guide.html

# Chapter 1. Developing EJB 2.1 Session Beans

## 1.1. EJB 2.1 Session Beans Description

### 1.1.1. Introduction

A Session Bean is composed of the following parts, which are developed by the Enterprise Bean Provider:

- The **Component Interface** is the client view of the bean. It contains all the "business methods" of the bean.
- The **Home Interface** contains all the methods for the bean life cycle (creation, suppression) used by the client application.
- The **bean implementation class** implements the business methods and all the methods (described in the EJB specification), allowing the bean to be managed in the container.
- The **deployment descriptor** contains the bean properties that can be edited at assembly or deployment time.

Note that, according to the EJB 2.1 specification, the couple "Component Interface and Home Interface" may be either local or remote. **Local Interfaces** (Home and Component) are to be used by a client running in the same JVM as the EJB component. Create and finder methods of a local or remote home interface return local or remote component interfaces respectively. An EJB component can have both remote and local interfaces, even if typically only one type of interface is provided.

The description of these elements is provided in the following sections.

**Note**

in this documentation, the term "Bean" always means "Enterprise Bean."

A session bean object is a short-lived object that executes on behalf of a single client.There are **stateless** and **stateful session beans**. Stateless beans do not maintain state across method calls. Any instance of stateless beans can be used by any client at any time. Stateful session beans maintain state within and between transactions. Each stateful session bean object is associated with a specific client. A stateful session bean with container-managed transaction demarcation can optionally implement the **SessionSynchronization** interface. In this case, the bean objects will be informed of transaction boundaries. A rollback could result in a session bean object's state being inconsistent; in this case, implementing the SessionSynchronization interface may enable the bean object to update its state according to the transaction completion status.

### 1.1.2. The Home Interface

A Session bean's home interface defines one or more `create(...)` methods. Each `create` method must be named create and must match one of the `ejbCreate` methods defined in the enterprise Bean class. The return type of a `create` method must be the enterprise Bean's remote interface type. The home interface of a stateless session bean must have one `create` method that takes no arguments. All the exceptions defined in the throws clause of an `ejbCreate` method must be defined in the throws clause of the matching `create` method of the home interface. A remote home interface extends the `javax.ejb.EJBHome` interface, while a local home interface extends the `javax.ejb.EJBLocalHome` interface.

### 1.1.2.1. Example

The following examples use a Session Bean named Op.

```
public interface OpHome extends EJBHome {
     Op create(String user) throws CreateException, RemoteException;
 }
```

A local home interface could be defined as follows (LocalOp being the local component interface of the bean):

```
public interface LocalOpHome extends EJBLocalHome {
    LocalOp create(String user) throws CreateException;
}
```

## 1.1.3. The Component Interface

The Component Interface is the client's view of an instance of the session bean. This interface contains the business methods of the enterprise bean. The interface must extend the `javax.ejb.EJBObject` interface if it is remote, or the `javax.ejb.EJBLocalObject` if it is local. The methods defined in a remote component interface must follow the rules for Java RMI (this means that their arguments and return value must be valid types for java RMI, and their throws clause must include the `java.rmi.RemoteException`). For each method defined in the component interface, there must be a matching method in the enterprise Bean's class (same name, same arguments number and types, same return type, and same exception list, except for RemoteException).

### 1.1.3.1. Example

```
public interface Op extends EJBObject {
    public void buy (int Shares)  throws RemoteException;
    public int  read ()           throws RemoteException;
}
```

The same type of component interface could be defined as a local interface (even if it is not considered good design to define the same interface as both local and remote):

```
public interface LocalOp extends EJBLocalObject {
      public void buy (int Shares);
      public int  read ();
  }
```

## 1.1.4. The Enterprise Bean Class

This class implements the Bean's business methods of the component interface and the methods of the `SessionBean` interface, which are those dedicated to the EJB environment. The class must be defined as public and may not be abstract. The Session Bean interface methods that the EJB provider must develop are the following:

- public void `setSessionContext`(SessionContext ic);

  This method is used by the container to pass a reference to the SessionContext to the bean instance. The container invokes this method on an instance after the instance has been created. Generally, this method stores this reference in an instance variable.

- public void `ejbRemove`();

  This method is invoked by the container when the instance is in the process of being removed by the container. Since most session Beans do not have any resource state to clean up, the implementation of this method is typically left empty.

- public void `ejbPassivate`();

  This method is invoked by the container when it wants to passivate the instance. After this method completes, the instance must be in a state that allows the container to use the Java Serialization protocol to externalize and store the instance's state.

- public void `ejbActivate`();

  This method is invoked by the container when the instance has just been reactivated. The instance should acquire any resource that it has released earlier in the ejbPassivate() method.

A stateful session Bean with container-managed transaction demarcation can optionally implement the `javax.ejb.SessionSynchronization` interface. This interface can provide the Bean with transaction synchronization notifications. The Session Synchronization interface methods that the EJB provider must develop are the following:

- public void afterBegin();

  This method notifies a session Bean instance that a new transaction has started. At this point the instance is already in the transaction and can do any work it requires within the scope of the transaction.

- public void afterCompletion(boolean committed);

  This method notifies a session Bean instance that a transaction commit protocol has completed and tells the instance whether the transaction has been committed or rolled back.

- public void beforeCompletion();

  This method notifies a session Bean instance that a transaction is about to be committed.

## 1.1.4.1. Example

```
package sb;

import java.rmi.RemoteException;
import javax.ejb.EJBException;
import javax.ejb.EJBObject;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.ejb.SessionSynchronization;
import javax.naming.InitialContext;
import javax.naming.NamingException;

// This is an example of Session Bean, stateful, and synchronized.

public class OpBean implements SessionBean, SessionSynchronization {

    protected int total = 0;        // actual state of the bean
    protected int newtotal = 0;        // value inside Tx, not yet committed.
    protected String clientUser = null;
    protected SessionContext sessionContext = null;

    public void  ejbCreate(String user) {
        total = 0;
        newtotal = total;
        clientUser = user;
    }

    public void ejbActivate() {
        // Nothing to do for this simple example
    }

    public void ejbPassivate() {
        // Nothing to do for this simple example
    }

    public void ejbRemove() {
        // Nothing to do for this simple example
    }
```

```
    public void setSessionContext(SessionContext sessionContext) {
        this.sessionContext = sessionContext;
    }

    public void afterBegin() {
        newtotal = total;
    }

    public void beforeCompletion() {
        // Nothing to do for this simple example

        // We can access the bean environment everywhere in the bean,
        // for example here!
        try {
            InitialContext ictx = new InitialContext();
            String value = (String) ictx.lookup("java:comp/env/prop1");
            // value should be the one defined in ejb-jar.xml
        } catch (NamingException e) {
            throw new EJBException(e);
        }
    }

    public void afterCompletion(boolean committed) {
        if (committed) {
            total = newtotal;
        } else {
            newtotal = total;
        }
    }

    public void buy(int s) {
        newtotal = newtotal + s;
        return;
    }

    public int read() {
        return newtotal;
    }
}
```

# 1.2. Tuning EJB 2.1 session beans

JOnAS handles a pool for each stateless session bean. The pool can be configured in the JOnAS-specific deployment descriptor with the following tags:

- min-pool-size

  This optional integer value represents the minimum instances that will be created in the pool when the bean is loaded. This will improve bean instance creation time, at least for the first beans. The default value is 0.

- max-cache-size

  This optional integer value represents the maximum of instances in memory. The purpose of this value is to keep JOnAS scalable. The policy is the following: At bean creation time, an instance is taken from the pool of free instances. If the pool is empty, a new instance is always created. When the instance must be released (at the end of a business method), it is pushed into the pool, except if the current number of instances created exceeds the max-cache-size, in which case this instance is dropped. The default value is no limit.

### Example 1.1. Example

```
<jonas-ejb-jar>
   <jonas-session>
     <ejb-name>SessSLR</ejb-name>
     <jndi-name>EJB/SessHome</jndi-name>
     <max-cache-size>20</max-cache-size>
     <min-pool-size>10</min-pool-size>
   </jonas-session>
 </jonas-ejb-jar>
```

# Chapter 2. Developing Entity Beans

## 2.1. EJB 2.1 Entity Beans Description

### 2.1.1. Introduction

An Entity Bean is composed of the following parts, which are developed by the Enterprise Bean Provider:

- The *Component Interface* is the client view of the bean. It contains all the "business methods" of the bean.
- The *Home Interface* contains all the methods for the bean life cycle (creation, suppression) and for instance retrieval (finding one or several bean objects) used by the client application. It can also contain methods called "home methods," supplied by the bean provider, for business logic which are not specific to a bean instance.
- The *Primary Key* class (for entity beans only) contains a subset of the bean's fields that identifies a particular instance of an entity bean. This class is optional since the bean programmer can alternatively choose a standard class (for example, java.lang.String)
- The *bean implementation* class implements the business methods and all the methods (described in the EJB specification) allowing the bean to be managed in the container.
- The *deployment descriptor*, containing the bean properties that can be edited at assembly or deployment time.

Note that, according to the EJB 2.1 specification, the couple *Component Interface* and *Home Interface* may be either local or remote. *Local Interfaces* (Home and Component) are to be used by a client running in the same JVM as the EJB component. Create and finder methods of a local or remote home interface return local or remote component interfaces respectively. An EJB component can have both remote and local interfaces, even if typically only one type of interface is provided.

The description of these elements is provided in the following sections.

**Note**

in this documentation, the term "Bean" always means "Enterprise Bean."

An entity bean represents persistent data. It is an object view of an entity stored in a relational database. The persistence of an entity bean can be handled in two ways:

- *Container-Managed Persistence*: the persistence is implicitly managed by the container, no code for data access is supplied by the bean provider. The bean's state will be stored in a relational database according to a mapping description delivered within the deployment descriptor (CMP 1) or according to an implicit mapping (CMP 2).
- *Bean-Managed Persistence*: the bean provider writes the database access operations (JDBC code) in the methods of the enterprise bean that are specified for data creation, load, store, retrieval, and remove operations (ejbCreate, ejbLoad, ejbStore, ejbFind..., ejbRemove).

Currently, the platform handles persistence in relational storage systems through the JDBC interface. For both container-managed and bean-managed persistence, JDBC connections are obtained from an object provided at the EJB server level, the DataSource. The DataSource interface is defined in the JDBC 2.0 standard extensions. A DataSource object identifies a database and a means to access it via

JDBC (a JDBC driver). An EJB server may propose access to several databases and thus provides the corresponding DataSource objects. DataSources are described in more detail in the sectionSection 2.6, “Configuring JDBC DataSources with 'dbm' service”

# 2.1.2. The Home Interface

In addition to home business methods, the `Home interface` is used by any client application to create, remove, and retrieve instances of the entity bean. The bean provider only needs to provide the desired interface; the container will automatically provide the implementation. The interface must extend the `javax.ejb.EJBHome` interface if it is remote, or the `javax.ejb.EJBLocalHome` interface if it is local. The methods of a remote home interface must follow the rules for java RMI. The signatures of the `create` and `finder methods` should match the signatures of the `ejbCreate` and `ejbFinder` methods that will be provided later in the enterprise bean implementation class (same number and types of arguments, but different return types).

## 2.1.2.1. create methods

- The return type is the enterprise bean's component interface.
- The exceptions defined in the throws clause must include the exceptions defined for the ejbCreate and ejbPostCreate methods, and must include javax.ejb.CreateException and java.rmi.RemoteException (the latter, only for a remote interface).

## 2.1.2.2. remove methods

- The interfaces for these methods must not be defined, they are inherited from EJBHome or EJBLocalHome.
- The method is void remove taking as argument the primary key object or the handle (for a remote interface).
- The exceptions defined in the throws clause should be javax.ejb.RemoveException and java.ejb.EJBException for a local interface.
- The exceptions defined in the throws clause should be javax.ejb.RemoveException and java.rmi.RemoteException for a remote interface.

## 2.1.2.3. finder methods

Finder methods are used to search for an EJB object or a collection of EJB objects. The arguments of the method are used by the entity bean implementation to locate the requested entity objects. For bean-managed persistence, the bean provider is responsible for developing the corresponding ejbFinder methods in the bean implementation. For container-managed persistence, the bean provider does not write these methods; they are generated at deployment time by the platform tools; the description of the method is provided in the deployment descriptor, as defined in the section Section 2.3, “Configuring Database Access for Container Managed Persistence” In the Home interface, the finder methods must adhere to the following rules:

- They must be named " find<method> " (e.g. `findLargeAccounts` ).
- The return type must be the enterprise bean's component interface, or a collection thereof.
- The exceptions defined in the throws clause must include the exceptions defined for the matching ejbFind method, and must include `javax.ejb.FinderException` and `java.rmi.RemoteException` (the latter, only for a remote interface).

At least one of these methods is mandatory: findByPrimaryKey , which takes as argument a primary key value and returns the corresponding EJB object.

home methods:

- Home methods are methods that the bean provider supplies for business logic that is not specific to an entity bean instance.
- The throws clause of every home method on the remote home interface includes the java.rmi.RemoteException.
- Home methods implementation is provided by the bean developer in the bean implementation class as public static methods named `ejbHome<METHOD_NAME>(...)`, where <METHOD_NAME> is the name of the method in the home interface.

### 2.1.2.4. Example

The Account bean example, provided with the platform examples, is used to illustrate these concepts. The state of an entity bean instance is stored in a relational database, where the following table should exist, if CMP 1.1 is used:

```
create table ACCOUNT (ACCNO integer primary key, CUSTOMER varchar(30), BALANCE
 number(15,4));
```

```
public interface AccountHome extends EJBHome {
    public Account create(int accno, String customer, double balance) throws
 RemoteException, CreateException;
    public Account findByPrimaryKey(Integer pk) throws RemoteException, FinderException;
    public Account findByNumber(int accno) throws RemoteException, FinderException;
    public Enumeration findLargeAccounts(double val) throws RemoteException,
 FinderException;
}
```

## 2.1.3. The Component Interface

Business methods:

The Component Interface is the client's view of an instance of the entity bean. It is what is returned to the client by the Home interface after creating or finding an entity bean instance. This interface contains the business methods of the enterprise bean. The interface must extend the `javax.ejb.EJBObject` interface if it is remote, or the `javax.ejb.EJBLocalObject` if it is local. The methods of a remote component interface must follow the rules for java RMI. For each method defined in this component interface, there must be a matching method of the bean implementation class (same arguments number and types, same return type, same exceptions except for RemoteException).

### 2.1.3.1. Example

```
public interface Account extends EJBObject {
    public double getBalance() throws RemoteException;
    public void setBalance(double d) throws RemoteException;
    public String getCustomer() throws RemoteException;
    public void setCustomer(String c) throws RemoteException;
    public int getNumber() throws RemoteException;
}
```

## 2.1.4. The Primary Key Class

The Primary Key class is necessary for entity beans only. It encapsulates the fields representing the primary key of an entity bean in a single object. If the primary key in the database table is composed of a single column with a basic data type, the simplest way to define the primary key in the bean is to

use a standard java class (for example, `java.lang.Integer` or `java.lang.String` ). This must have the same type as a field in the bean class. It is not possible to define it as a primitive field (for example, int, float or boolean). Then, it is only necessary to specify the type of the primary key in the deployment descriptor:

```
<prim-key-class>java.lang.Integer</prim-key-class>
```

And, for container-managed persistence, the field which represents the primary key:

```
<primkey-field>accno</primkey-field>
```

The alternative way is to define its own Primary Key class, described as follows:

The class must be serializable and must provide suitable implementation of the `hashcode()` and `equals(Object)` methods.

For container-managed persistence , the following rules must be followed:

- The fields of the primary key class must be declared as public.
- The primary key class must have a public default constructor.
- The names of the fields in the primary key class must be a subset of the names of the container-managed fields of the enterprise bean.

## 2.1.4.1. Example

```
public class  AccountBeanPK implements java.io.Serializable {
    public int accno;
    public AccountBeanPK(int accno) { this.accno = accno; }
    public AccountBeanPK() { }
    public int hashcode() { return accno; }
    public boolean equals(Object other) {
        ...
    }
}
```

## 2.1.4.2. Special case: Automatic generation of primary key fields

There are two ways to manage the automatic primary key with JOnAS. The first method is closer to what is described in the EJB specification, i.e. an automatic PK is a hidden field, the type of which is not known even by the application. The second method is to declare a usual PK CMP field of type java.lang.Integer as automatic. The two cases are described below.

1. Standard automatic primary keys

   In this case, an automatic PK is a hidden field, the type of which is not known even by the application. All that is necessary is to stipulate in the standard deployment descriptor that this EJB has an automatic PK, by specifying java.lang.Object as primkey-class. The primary key will be completely hidden to the application (no CMP field, no getter/setter method). This is valid for both CMP 2.x and CMP1 entity beans. The container will create an internal CMP field and generate its value when the entity bean is created.

   Example:

   Standard deployment descriptor:

```
  <entity>
    ...
    <ejb-name>AddressEJB</ejb-name>
    <local-home>com.titan.address.AddressHomeLocal</local-home>
    <local>com.titan.address.AddressLocal</local>
    <ejb-class>com.titan.address.AddressBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.Object</prim-key-class>
    <reentrant>False</reentrant>
    <cmp-version>2.x</cmp-version>
    <abstract-schema-name>Cmp2_Address</abstract-schema-name>
    <cmp-field><field-name>street</field-name></cmp-field>
    <cmp-field><field-name>city</field-name></cmp-field>
    <cmp-field><field-name>state</field-name></cmp-field>
    <cmp-field><field-name>zip</field-name></cmp-field>
```

Address Bean Class extract:

```
   // Primary key is not explicitly initialized during ejbCreate method
   // No cmp field corresponds to the primary key
   public Integer ejbCreateAddress(String street, String city,
        String state, String zip ) throws javax.ejb.CreateException {
     setStreet(street);
     setCity(city);
     setState(state);
     setZip(zip);
     return null;
   }
```

If nothing else is specified, and the JOnAS default CMP 2 database mapping is used, the JOnAS container will generate a database column with name JPK_ to handle this PK. However, it is possible to specify in the JOnAS-specific Deployment Descriptor the name of the column that will be used to store the PK value in the table, using the specific <automatic-pk-field-name> element, as follows (this is necessary for CMP2 legacy and for CMP1):

JOnAS-specific deployment descriptor:

```
  <jonas-ejb-jar xmlns="http://www.objectweb.org/jonas/ns"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.objectweb.org/jonas/ns http://www.objectweb.org/jonas/
ns/jonas-ejb-jar_4_0.xsd" >
  <jonas-entity>
     <ejb-name>AddressEJB</ejb-name>
     <jdbc-mapping>
        <jndi-name>jdbc_1</jndi-name>
        <automatic-pk-field-name>FieldPkAuto</automatic-pk-field-name>
     </jdbc-mapping>
  </jonas-entity>
```

2. CMP field as automatic primary key

   The idea is to declare a usual PK CMP field of type java.lang.Integer as automatic, then it no longer appears in create methods and its value is automatically generated by the container at EJB instance creation time. But it is still a cmp field, with getter/setter methods, accessible from the application. Example:

   In the standard DD, there is a usual primary key definition,

```
  <entity>
     ...
     <prim-key-class>java.lang.Integer</prim-key-class>
     <cmp-field><field-name>id</field-name></cmp-field>
     <primkey-field>id</primkey-field>
```

   and in the JOnAS-specific Deployment Descriptor, it should be specified that this PK is automatic,

```
  <jonas-entity>
     ...
```

```
    <jdbc-mapping>

    <automatic-pk>true</automatic-pk>
```

Note: The automatic primary key is given a unique ID by an algorithm that is based on the system time; therefore, IDs may be not sequential.

Important restriction: This algorithm will not work if used inside a cluster with the same entity bean that is being managed in several jonas servers.

# 2.1.5. The Enterprise Bean Class

The EJB implementation class implements the bean's business methods of the component interface and the methods dedicated to the EJB environment, the interface of which is explicitly defined in the EJB specification. The class must implement the `javax.ejb.EntityBean` interface, must be defined as public, cannot be abstract for CMP 1.1, and must be abstract for CMP 2.0 (in this case, the abstract methods are the get and set accessor methods of the bean cmp and cmr fields). Following is a list of the EJB environment dedicated methods that the EJB provider must develop.

The first set of methods are those corresponding to the create and find methods of the Home interface:

- `public PrimaryKeyClass ejbCreate(...);`

  This method is invoked by the container when a client invokes the corresponding create operation on the enterprise Bean's home interface. The method should initialize instance's variables from the input arguments. The returned object should be the primary key of the created instance. For bean-managed persistence, the bean provider should develop here the JDBC code to create the corresponding data in the database. For container-managed persistence, the container will perform the database insert after the ejbCreate method completes and the return value should be null .

- `public void ejbPostCreate(...);`

  There is a matching ejbPostCreate method (same input parameters) for each ejbCreate method. The container invokes this method after the execution of the matching ejbCreate(...) method. During the ejbPostCreate method, the object identity is available.

- `public <PrimaryKeyClass or Collection> ejbFind<method> (...);` // bean-managed persistence only

  The container invokes this method on a bean instance that is not associated with any particular object identity (some kind of class method ...) when the client invokes the corresponding method on the Home interface. The implementation uses the arguments to locate the requested object(s) in the database and returns a primary key (or a collection thereof). Currently, collections will be represented as `java.util.Enumeration` or `java.util.Collection` . The mandatory `FindByPrimaryKey` method takes as argument a primary key type value and returns a primary key object (it verifies that the corresponding entity exists in the database). For container-managed persistence , the bean provider does not have to write these finder methods; they are generated at deployment time by the EJB platform tools. The information needed by the EJB platform for automatically generating these finder methods should be provided by the bean programmer. The EJB 1.1 specification does not specify the format of this finder method description; for JOnAS , the CMP 1.1 finder methods description should be provided in the JOnAS-specific deployment descriptor of the Entity Bean (as an SQL query). Refer to the section Section 2.3, “Configuring Database Access for Container Managed Persistence”. The EJB 2.0 specification defines a standard way to describe these finder methods, i.e. in the standard deployment descriptor, as an EJB-QL query. Also refer to the section Section 2.3, “Configuring Database Access for Container Managed Persistence”. Then, the methods of the `javax.ejb.EntityBean` interface must be implemented:

  - `public void setEntityContext (EntityContext ic);`

Used by the container to pass a reference to the EntityContext to the bean instance. The container invokes this method on an instance after the instance has been created. Generally, this method is used to store this reference in an instance variable.

- `public void unSetEntityContext ();`

  Unset the associated entity context. The container calls this method before removing the instance. This is the last method the container invokes on the instance.

- `public void ejbActivate ();`

  The container invokes this method when the instance is taken out of the pool of available instances to become associated with a specific EJB object. This method transitions the instance to the ready state.

- `public void ejbPassivate ();`

  The container invokes this method on an instance before the instance becomes dissociated with a specific EJB object. After this method completes, the container will place the instance into the pool of available instances.

- `public void ejbRemove ();`

  This method is invoked by the container when a client invokes a remove operation on the enterprise bean. For entity beans with bean-managed persistence , this method should contain the JDBC code to remove the corresponding data in the database. For container-managed persistence , this method is called before the container removes the entity representation in the database.

- `public void ejbLoad ();`

  The container invokes this method to instruct the instance to synchronize its state by loading it from the underlying database. For bean-managed persistence , the EJB provider should code at this location the JDBC statements for reading the data in the database. For container-managed persistence , loading the data from the database will be done automatically by the container just before ejbLoad is called, and the ejbLoad method should only contain some "after loading calculation statements."

- `public void ejbStore ();`

  The container invokes this method to instruct the instance to synchronize its state by storing it to the underlying database. For bean-managed persistence , the EJB provider should code at this location the JDBC statements for writing the data in the database. For entity beans with container-managed persistence , this method should only contain some "pre-store statements," since the container will extract the container-managed fields and write them to the database just after the ejbStore method call.

## 2.1.5.1. Example

The following examples are for container-managed persistence with EJB 1.1 and EJB 2.0. For bean-managed persistence, refer to the examples delivered with your specific platform.

### 2.1.5.1.1. CMP 1.1

```
package eb;

import java.rmi.RemoteException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.ejb.ObjectNotFoundException;
import javax.ejb.RemoveException;
import javax.ejb.EJBException;
```

```
public class AccountImplBean implements EntityBean {

  // Keep the reference on the EntityContext
  protected EntityContext entityContext;

  // Object state
  public Integer accno;
  public String customer;
  public double balance;

  public Integer ejbCreate(int val_accno, String val_customer, double val_balance) {

    // Init object state
    accno = new Integer(val_accno);
    customer = val_customer;
    balance = val_balance;
    return null;
  }

  public void ejbPostCreate(int val_accno, String val_customer, double val_balance) {
    // Nothing to be done for this simple example.
  }

  public void ejbActivate() {
    // Nothing to be done for this simple example.
  }

  public void ejbLoad() {
    // Nothing to be done for this simple example, in implicit persistence.
  }

  public void ejbPassivate() {
    // Nothing to be done for this simple example.
  }


  public void ejbRemove() {
    // Nothing to be done for this simple example, in implicit persistence.
  }

  public void ejbStore() {
    // Nothing to be done for this simple example, in implicit persistence.
  }

  public void setEntityContext(EntityContext ctx) {
    // Keep the entity context in object
    entityContext = ctx;
  }

  public void unsetEntityContext() {
    entityContext = null;
  }

  public double getBalance() {
    return balance;
  }

  public void setBalance(double d) {
    balance = balance + d;
  }

  public String  getCustomer() {
    return customer;
  }

  public void setCustomer(String c) {
    customer = c;
  }

  public int getNumber()  {
    return accno.intValue();
  }
}
```

### 2.1.5.1.2. CMP 2.0

```
import java.rmi.RemoteException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
```

```
import javax.ejb.ObjectNotFoundException;
import javax.ejb.RemoveException;
import javax.ejb.CreateException;
import javax.ejb.EJBException;

public abstract class AccountImpl2Bean implements EntityBean {

  // Keep the reference on the EntityContext
  protected EntityContext entityContext;


  /*======================== Abstract set and get accessors for cmp fields ==============*/

  public abstract String getCustomer();
  public abstract void setCustomer(String customer);

  public abstract double getBalance();
  public abstract void setBalance(double balance);

  public abstract int getAccno();
  public abstract void setAccno(int accno);

  /*======================== ejbCreate methods ============================*/


  public Integer ejbCreate(int val_accno, String val_customer, double val_balance)
  throws CreateException {

    // Init object state
    setAccno(val_accno);
    setCustomer(val_customer);
    setBalance(val_balance);
    return null;
  }

  public void ejbPostCreate(int val_accno, String val_customer, double val_balance) {
    // Nothing to be done for this simple example.
  }


  /*====================== javax.ejb.EntityBean implementation =================*/

  public void ejbActivate() {
    // Nothing to be done for this simple example.
  }

  public void ejbLoad() {
    // Nothing to be done for this simple example, in implicit persistence.
  }

  public void ejbPassivate() {
    // Nothing to be done for this simple example.
  }

  public void ejbRemove() throws RemoveException {
    // Nothing to be done for this simple example, in implicit persistence.
  }

  public void ejbStore() {
    // Nothing to be done for this simple example, in implicit persistence.
  }

  public void setEntityContext(EntityContext ctx) {

    // Keep the entity context in object
    entityContext = ctx;
  }

  public void unsetEntityContext()  {
    entityContext = null;
  }

  /**
  * Business method to get the Account number
  */
  public int getNumber()  {
    return getAccno();
  }
}
```

# 2.2. Writing Database Access Methods for Bean Managed Persistence

For bean-managed persistence , data access operations are developed by the bean provider using the JDBC interface. However, getting database connections must be obtained through the javax.sql.DataSource interface on a datasource object provided by the EJB platform. This is mandatory since the EJB platform is responsible for managing the connection pool and for transaction management. Thus, to get a JDBC connection, in each method performing database operations, the bean provider must:

- call the getConnection(...) method of the DataSource object, to obtain a connection to perform the JDBC operations in the current transactional context (if there are JDBC operations),
- call the close() method on this connection after the database access operations, so that the connection can be returned to the connection pool (and be dissociated from the potential current transaction).

A method that performs database access must always contain the getConnection and close statements, as follows:

```
public void doSomethingInDB (...) {
  conn = dataSource.getConnection();
  ... // Database access operations
  conn.close();
}
```

A DataSource object associates a JDBC driver with a database (as an ODBC datasource). It is created and registered in JNDI by the EJB server at launch time (refer also to the section " JDBC DataSources configuration [1] ").

A DataSource object is a resource manager connection factory for `java.sql.Connection` objects, which implements connections to a database management system. The enterprise bean code refers to resource factories using logical names called "Resource manager connection factory references." The resource manager connection factory references are special entries in the enterprise bean environment. The bean provider must use resource manager connection factory references to obtain the datasource object as follow:

- Declare the resource reference in the standard deployment descriptor using a `resource-ref` element.
- Lookup the datasource in the enterprise bean environment using the JNDI interface (refer to the section Enterprise Bean's Environment [2] ).

The deployer binds the resource manager connection factory references to the actual resource factories that are configured in the server. This binding is done in the JOnAS-specific deployment descriptor using the jonas-resource element.

## 2.2.1. Example

The declaration of the resource reference in the standard deployment descriptor looks like the following:

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

[1] configuration_guide.html#config.jdbc

The <res-auth> element indicates which of the two resource manager authentication approaches is used:

- Container : the deployer sets up the sign-on information.
- Bean : the bean programmer should use the getConnection method with user and password parameters.

The JOnAS-specific deployment descriptor must map the environment JNDI name of the resource to the actual JNDI name of the resource object managed by the EJB server. This is done in the <jonas-resource> element.

```
 <jonas-entity>
  <ejb-name>AccountExpl</ejb-name>
  <jndi-name>AccountExplHome</jndi-name>
  <jonas-resource>
  <res-ref-name>jdbc/AccountExplDs</res-ref-name>
  <jndi-name>jdbc_1</jndi-name>
  </jonas-resource>
  </jonas-entity>
```

The ejbStore method of the same Account example with bean-managed persistence is shown in the following example. It performs JDBC operations to update the database record representing the state of the entity bean instance. The JDBC connection is obtained from the datasource associated with the bean. This datasource has been instantiated by the EJB server and is available for the bean through its resource reference name, which is defined in the standard deployment descriptor.

In the bean, a reference to a datasource object of the EJB server is initialized:

```
it = new InitialContext();

ds = (DataSource)it.lookup("java:comp/env/jdbc/AccountExplDs");
```

Then, this datasource object is used in the implementation of the methods performing JDBC operations, such as ejbStore, as illustrated in the following:

```
public void ejbStore
 Connection conn = null;
  PreparedStatement stmt = null;
  try { // get a connection
  conn = ds.getConnection();
  // store Object state in DB
  stmt = conn.prepareStatement("update account set customer=?,balance=? where accno=?");
  stmt.setString(1, customer);
  stmt.setDouble(2, balance);
  Integer pk = (Integer)entityContext.getPrimaryKey();
  stmt.setInt(3, pk.accno);
  stmt.executeUpdate();
  } catch (SQLException e) {
  throw new javax.ejb.EJBException("Failed to store bean to database", e);
  } finally {
  try {
  if (stmt != null) stmt.close();  // close statement
  if (conn != null) conn.close();  // release connection
  } catch (Exception ignore) {}
  }
}
```

Note that the close statement instruction may be important if the server is intensively accessed by many clients performing entity bean access. If the statement is not closed in the finally block, since stmt is in the scope of the method, it will be deleted at the end of the method (and the close will be implicitly done). However, it may be some time before the Java garbage collector deletes the statement object. Therefore, if the number of clients performing entity bean access is important, the DBMS may raise a "too many opened cursors" exception (a JDBC statement corresponds to a DBMS cursor). Since connection pooling is performed by the platform, closing the connection will not result in a physical connection close, therefore opened cursors will not be closed. Thus, it is preferable to explicitly close the statement in the method.

It is a good programming practice to put the JDBC connection and JDBC statement close operations in a finally bloc of the try statement.

# 2.3. Configuring Database Access for Container Managed Persistence

The standard way to indicate to an EJB platform that an entity bean has container-managed persistence is to fill the <persistence-type> tag of the deployment descriptor with the value "container," and to fill the <cmp-field> tag of the deployment descriptor with the list of container-managed fields (the fields that the container will have in charge to make persistent). The CMP version (1.x or 2.x) should also be specified in the <cmp-version> tag. In the textual format of the deployment descriptor, this is represented by the following lines:

```
<persistence-type>container</persistence-type>
<cmp-version>2.x</cmp-version>
<cmp-field>
  <field-name>fieldOne</field-name>
</cmp-field>
<cmp-field>
  <field-name>fieldTwo</field-name>
</cmp-field>
```

With container-managed persistence the programmer need not develop the code for accessing the data in the relational database; this code is included in the container itself (generated by the platform tools). However, for the EJB platform to know how to access the database and which data to read and write in the database, two types of information must be provided with the bean:

- First, the container must know which database to access and how to access it. To do this, the only required information is the name of the DataSource that will be used to get the JDBC connection. For container-managed persistence, only one DataSource per bean should be used.

- Then, it is necessary to know the mapping of the bean fields to the underlying database (which table, which column). For CMP 1.1 or CMP 2.0, this mapping is specified by the deployer in the JOnAS-specific deployment descriptor. Note that for CMP 2.0, this mapping may be entirely generated by JOnAS.

The EJB specification does not specify how this information should be provided to the EJB platform by the bean deployer. Therefore, what is described in the remainder of this section is specific to JOnAS.

## 2.3.1. CMP 1.x specifics

For CMP 1.1, the bean deployer is responsible for defining the mapping of the bean fields to the database table columns. The name of the DataSource can be set at deployment time, since it depends on the EJB platform configuration. This database configuration information is defined in the JOnAS-specific deployment descriptor via the jdbc-mapping element. The following example defines the mapping for a CMP 1.1 entity bean:

```
<jdbc-mapping>
  <jndi-name>jdbc_1</jndi-name>
  <jdbc-table-name>accountsample</jdbc-table-name>
  <cmp-field-jdbc-mapping>
  <field-name>mAccno</field-name>
  <jdbc-field-name>accno</jdbc-field-name>
  </cmp-field-jdbc-mapping>
  <cmp-field-jdbc-mapping>
  <field-name>mCustomer</field-name>
  <jdbc-field-name>customer</jdbc-field-name>
  </cmp-field-jdbc-mapping>
  <cmp-field-jdbc-mapping>
  <field-name>mBalance</field-name>
  <jdbc-field-name>balance</jdbc-field-name>
</jdbc-mapping>
```

jdbc_1 is the JNDI name of the DataSource object identifying the database. accountsample is the name of the table used to store the bean instances in the database. mAccno, mCustomer, and mBalance are the names of the container-managed fields of the bean to be stored in the accno, customer, and balance columns of the accountsample table. This example applies to container-managed persistence. For bean-managed persistence, the database mapping does not exist.

For CMP 1.1, the jdbc-mapping element can also contain information defining the behaviour of the implementation of a find<method> method (i.e. the ejbFind<method> method, that will be generated by the platform tools). This information is represented by the finder-method-jdbc-mapping element.

For each finder method, this element provides a way to define an SQL WHERE clause that will be used in the generated finder method implementation to query the relational table storing the bean entities. Note that the table column names should be used, not the bean field names. Example:

```
<finder-method-jdbc-mapping>
  <jonas-method>
    <method-name>findLargeAccounts</method-name>
  </jonas-method>
  <jdbc-where-clause>where balance &gt; ?</jdbc-where-clause>
</finder-method-jdbc-mapping>
```

The previous finder method description will cause the platform tools to generate an implementation of ejbFindLargeAccount(double arg) that returns the primary keys of the entity bean objects corresponding to the tuples returned by the "select ... from Account where balance > ?", where '?' will be replaced by the value of the first argument of the findLargeAccount method. If several '?' characters appear in the provided WHERE clause, this means that the finder method has several arguments and the '?' characters will correspond to these arguments, adhering to the order of the method signature.

In the WHERE clause, the parameters can be followed by a number, which specifies the method parameter number that will be used by the query in this position. Example: The WHERE clause of the following finder method can be:

```
Enumeration findByTextAndDateCondition(String text, java.sql.Date date)

WHERE (description like ?1 OR summary like ?1) AND (?2 &gt; date)
```

Note that a <finder-method-jdbc-mapping> element for the findByPrimaryKey method is not necessary, since the meaning of this method is known.

The datatypes supported for container-managed fields in CMP 1.1 are the following:

### Table 2.1. Supported datatypes in CMP 1.1

| | | |
|---|---|---|
| boolean | BIT | getBoolean(), setBoolean() |
| byte | TINYINT | getByte(), setByte() |
| int | INTEGER | getInt(), setInt() |
| long | BIGINT | getLong(), setLong() |
| float | FLOAT | getFloat(), setFloat() |
| double | DOUBLE | getDouble(), setDouble |
| byte[] | VARBINARY or LONGVARBINARY (1) | getBytes(), setBytes() |
| java.lang.String | VARCHAR or LONGVARCHAR (1) | getString(), setString() |
| java.lang.Boolean | BIT | getBoolean(), setObject() |
| java.lang.Integer | INTEGER | getInt(), setObject() |
| java.lang.Short | SMALLINT | getShort(), setObject() |
| java.lang.Long | BIGINT | getLong(), setObject() |

| | | |
|---|---|---|
| java.lang.Float | REAL | getFloat(), setObject() |
| java.lang.Double | DOUBLE | getDouble(), setObject() |
| java.math.BigDecimal | NUMERIC | getBigDecimal(), setObject() |
| java.math.BigInteger | NUMERIC | getBigDecimal(), setObject() |
| java.sql.Date | DATE | getDate(), setDate() |
| java.sql.Time | TIME | getTime(), setTime() |
| java.sql.Timestamp | TIMESTAMP | getTimestamp(), setTimestamp() |
| any serializable class | VARBINARY or LONGVARBINARY (1) | getBytes(), setBytes() |

(1) The mapping for String will normally be VARCHAR, but will turn into LONGVARCHAR if the given value exceeds the driver's limit on VARCHAR values. The case is similar for byte[] and VARBINARY and LONGVARBINARY values.

### 2.3.2. CMP 2.x specifics

For a CMP 2.0 entity bean, only the jndi-name element of the jdbc-mapping is mandatory, since the mapping may be generated automatically (for an explicit mapping definition, refer to the "JOnAS Database Mapping" section of the Using CMP2.0 persistence chapter):

```
    <jdbc-mapping>
      <jndi-name>jdbc_1</jndi-name>
    </jdbc-mapping>
    <cleanup>create</cleanup>
```

For a CMP 2.0 entity bean, the JOnAS-specific deployment descriptor contains an additional element, cleanup, at the same level as the jdbc-mapping element, which can have one of the following values:

removedata — at bean loading time, the content of the tables storing the bean data is deleted

removeall — at bean loading time, the tables storing the bean data are dropped (if they exist) and created

none — do nothing

create — default value (if the element is not specified), at bean loading time, the tables for storing the bean data are created if they do not exist

the information defining the behaviour of the implementation of a find<method> method is located in the standard deployment descriptor, as an EJB-QL query (i.e. this is not JOnAS-specific information). The same finder method example in CMP 2.0:

```
      <query>
        <query-method>
          <method-name>findLargeAccounts</method-name>
          <method-params>
              <method-param>double</method-param>
          </method-params>
        </query-method>
        <ejb-ql>SELECT OBJECT(o) FROM accountsample o WHERE o.balance &gt; ?1</ejb-ql>
      </query>
```

For CMP 2.0, the supported datatypes depend on the JORM mapper used.

## 2.4. Tuning EJB 2.1 entity beans

JOnAS must make a compromise between scalability and performance. For that reason, several tags in the JOnAS-specific deployment descriptor have been introduced. For most applications, there is

no need to change the default values for all these tags. See `$JONAS_ROOT/xml/jonas-ejb-jar_4_7.xsd` for a complete description of the JOnAS-specific deployment descriptor.

Note that if several of these elements are used, they should appear in the following order within the <jonas-entity> element:

1. is-modified-method-name
2. passivation-timeout
3. read-timeout
4. max-wait-time
5. inactivity-timeout
6. deadlock-timeout
7. shared
8. prefetch
9. hard-limit
10. max-cache-size
11. min-pool-size
12. cleanup
13. lock-policy

## 2.4.1. lock-policy

The JOnAS ejb container is able to manage seven different lock-policies:

| | |
|---|---|
| container-serialized (default) | The container ensures the transaction serialization. This policy is suitable for entity beans having non transacted methods that can modify the bean state. It works only if the bean is accessed from 1 jonas server (shared = false). |
| container-serialized-transacted | The container ensures the transaction serialization. This policy is suitable for most entity beans. |
| container-read-committed | This policy is similar to container-serialized-transacted, except that accesses outside transaction do not interfere with transactional accesses. This can help to avoid deadlocks in certain cases, when accessing a bean concurrently with and without a transactional context. The only drawback of this policy is that it consumes more memory (2 instances instead of 1). |
| container-read-uncommitted (deprecated) | All methods share the same instance (like container-serialized), but there is no synchronization. For example, this policy is of interest for read-only entity beans, or if the bean instances are very rarely modified. It will fail if two or more threads try to modify the same instance concurrently. This policy is deprecated because it can be replaced by `container-read-write` . |
| database | Allow the database to handle the transaction isolation. With this policy, it is possible to choose the transaction isolation in a |

| | |
|---|---|
| | database. This may be of interest for applications that heavily use transactional read-only operations, or when the flag shared is needed. It does not work with all databases and is not memory efficient. |
| read-only | The bean state is never written to the database. If the bean is `shared`, the bean state is read from the database regularly. Use the `read-timeout` tag to specify the timeout value. |
| container-read-write | All methods share the same instance (like container-serialized). A synchronization is done only when the instance is modified. No lock is taken while the instance is read only. This policy does not work if the bean is `shared`, nor does it work for CMP 1.x beans. |

Important: If CMP1 beans are deployed, only the following policies should be used:

- container-serialized
- container-serialized-transacted
- read-only
- container-read-committed
- database, but only with shared=true.

## 2.4.2. shared

This flag will be defined as `true` if the bean persistent state can be accessed outside the JOnAS Server. When this flag is `false`, the JOnAS Server can do some optimization, such as not re-reading the bean state before starting a new transaction. The default value depends on the lock-policy:

- `false` for `container-serialized`, `container-read-uncommitted` or `container-read-write`. For these 3 policies, shared=false is mandatory.
- `true` in the other cases.

Lock policy `database` works only if shared=true.

## 2.4.3. prefetch

This is a CMP2-specific option. The default is false. This can be set to true if it is desirable to have a cache managed after finder methods execution, in order to optimize further accesses inside the same transaction.

Important note:

- The prefetch will not be used for methods that have no transactional context.
- It is impossible to set the prefetch option if the lock policy is container-read-uncommitted.

## 2.4.4. max-cache-size / hard-limit / max-wait-time

This optional integer value represents the maximum number of instances in memory. The purpose of this value is to keep JOnAS scalable. The default value is "no limit". To save memory, this value should be set very low if it is known that instances will not be reused. Depending on whether `hard-limit` value is `true` or `false`, this `max-cache-size` value will be overtaken or not: In the case of `hard-limit = true`, the container will never allocate more instances than

the `max-cache-size` value. When the limit is reached, the thread will be set to waiting until instances are released. It is possible to specify the maximum time to wait for an instance with the tag `max-wait-time` . The default is 0, which means "no wait".

## 2.4.5. min-pool-size

This optional integer value represents the number of instances that will be created in the pool when the bean is loaded. This will improve bean instance create time, at least for the first instances. The default value is 0. When passivation occurs, for example if there are too many instances in memory, instances are released and placed in the pool only if min-pool-size is not reached. The intent is to try to keep at least min-pool-size instances in the pool of available instances.

## 2.4.6. is-modified-method-name

To improve performance of CMP 1.1 entity beans, JOnAS implements the isModified extension. Before performing an update, the container calls a method of the bean whose name is identified in the `is-modified-method-name` element of the JOnAS-specific deployment descriptor. This method is responsible for determining if the state of the bean has been changed. By doing this, the container determines if it must store data in the database or not.

**Note**

This is useless with CMP2 entity beans, since this will be done automatically by the container.

### 2.4.6.1. Example

The bean implementation manages a boolean `isDirty` and implements a method that returns the value of this boolean: `isModified`

```
private transient boolean isDirty;
public boolean isModified() {
    return isDirty;
}
```

The JOnAS-specific deployment descriptor directs the bean to implement an `isModified` method:

```
<jonas-entity>
  <ejb-name>Item</ejb-name>
  <is-modified-method-name>isModified</is-modified-method-name>
  .....
</jonas-entity>
```

Methods that modify the value of the bean must set the flag `isDirty` to `true` . Methods that restore the value of the bean from the database must reset the flag `isDirty` to `false` . Therefore, the flag must be set to `false` in the `ejbLoad()` and `ejbStore()` methods.

## 2.4.7. passivation-timeout

This flag is used only when lock-policy = container-serialized. When instances are accessed outside of any transaction, their state is kept in memory to improve performance. However, a passivation will occur in three situations:

1. When the bean is unloaded from the server, at least when the server is stopped.

2. When a transaction is started on this instance.

3. After a configurable timeout: passivation-timeout. If the bean is always accessed with no transaction, it may be prudent to periodically store the bean state on disk.

This passivation timeout can be configured in the JOnAS-specific deployment descriptor, with the non-mandatory tag <passivation-timeout>.

Example:

```
<jonas-entity>
  <ejb-name>Item</ejb-name>
  <passivation-timeout>5</passivation-timeout>
  .....
</jonas-entity>
```

This entity bean will be passivated every five second, if not accessed within transactions.

## 2.4.8. read-timeout

This flag is used only when lock-policy = read-only. In case shared=true has been set, it is important to synchronize the bean state by reading it periodically from the storage. This is configurable with the read-timeout flag. Value is in seconds.

## 2.4.9. inactivity-timeout

Bean passivation sends the state of the bean to persistent storage and removes from memory only the bean instance objects that are holding this state. All container objects handling bean access (remote object, interceptors, ...) are kept in memory so that future access will work, requiring only a reload operation (getting the state). It may be advantageous to conserve more memory and completely remove the bean instance from memory; this can be achieved through the <inactivity-timeout> element. This element is used to save memory when a bean has not been used for a long period of time. If the bean has not been used after the specified time (in seconds), all its container objects are removed from the server. If a client has kept a reference on a remote object and tries to use it, then the client will receive an exception.

# 2.5. Using the CMP 2 Persistence in Entity Beans

This section highlights the main differences between CMP as defined in EJB 2.0 specification (called CMP2.0) and CMP as defined in EJB 1.1 specification (called CMP1.1). Major new features in the standard development and deployment of CMP2.0 entity beans are listed (comparing them to CMP1.1), along with JOnAS-specific information. Mapping CMP2.0 entity beans to the database is described in detail. Note that the database mapping can be created entirely by JOnAS, in which case the JOnAS-specific deployment descriptor for an entity bean should contain only the datasource and the element indicating how the database should be initialized.

## 2.5.1. Standard CMP2.0 Aspects

This section briefly describes the new features available in CMP2.0 as compared to CMP 1.1, and how these features change the development of entity beans.

### 2.5.1.1. Entity Bean Implementation Class

The EJB implementation class 1) implements the bean's business methods of the component interface, 2) implements the methods dedicated to the EJB environment (the interface of which is explicitly defined in the EJB specification), and 3) defines the abstract methods representing both the

persistent fields (cmp-fields) and the relationship fields (cmr-fields). The class must implement the `javax.ejb.EntityBean` interface, be defined as public, and be abstract (which is not the case for CMP1.1, where it must not be abstract). The abstract methods are the get and set accessor methods of the bean cmp and cmr fields. Refer to the examples and details in the section " Developing Entity Beans [3] " of the JOnAS documentation.

### 2.5.1.2. Standard Deployment Descriptor

The standard way to indicate to an EJB platform that an entity bean has container-managed persistence is to fill the `<persistence-type>` tag of the deployment descriptor with the value "container," and to fill the `<cmp-field>` tags of the deployment descriptor with the list of container-managed fields (the fields that the container will have in charge to make persistent) and the `<cmr-field>` tags identifying the relationships. The CMP version (1.x or 2.x) should also be specified in the <cmp-version> tag. This is represented by the following lines in the deployment descriptor:

```
<persistence-type>container</persistence-type>
<cmp-version>1.x</cmp-version>
<cmp-field>
  <field-name>fieldOne</field-name>
</cmp-field>
<cmp-field>
  <field-name>fieldTwo</field-name>
</cmp-field>
```

Note that for running CMP1.1-defined entity beans on an EJB2.0 platform, such as JOnAS 3.x, you must introduce this <cmp-version> element in your deployment descriptors, since the default cmp-version value (if not specified) is 2.x .

Note that for CMP 2.0, the information defining the behaviour of the implementation of a find<method> method is located in the standard deployment descriptor as an EJB-QL query (this is not a JOnAS-specific information). For CMP 1.1, this information is located in the JOnAS-specific deployment descriptor as an SQL WHERE clause specified in a <finder-method-jdbc-mapping> element.

Finder method example in CMP 2.0: for a `findLargeAccounts(double val)` method defined on the Account entity bean of the JOnAS eb example.

```
<query>
  <query-method>
    <method-name>findLargeAccounts</method-name>
    <method-params>
        <method-param>double</method-param>
    </method-params>
  </query-method>
  <ejb-ql>SELECT OBJECT(o) FROM accountsample o WHERE o.balance &gt; ?1</ejb-ql>
</query>
```

## 2.5.2. JOnAS EJBQL extension

### 2.5.2.1. LIMIT clause

The LIMIT feature has been added to the standard EJBQL query language. This feature enables you to retrieve just a portion of the results generated by the rest of a query.

The syntax of the LIMIT clause is:

```
limit_clause ::= LIMIT limit_expression (, limit_expression )?
limit_expression ::= integer_literal | input_parameter
```

[3] ejb2_programmer_guide.html#ejb2.entity.desc

```
```

The first limit_expression corresponds to the start_at range and the second one to the size range.

The limit_clause is the last clause of the query:

```
ejbql ::= select_clause from_clause [where_clause] [orderby_clause] [limit_clause]
```

Example:

```
SELECT OBJECT(c) FROM jt2_Customer AS c ORDER BY c.id LIMIT ?1, 20
```

Note that this feature is currently not implemented on all the database types supported by JORM/MEDOR.

## 2.5.3. JOnAS Database mappers

For implementing the EJB 2.0 persistence (CMP2.0), JOnAS relies on the JORM [4] framework. JORM itself relies on JOnAS DataSources (specified in DataSource properties files) for connecting to the actual database. JORM must adapt its object-relational mapping to the underlying database, for which it makes use of adapters called "mappers." Thus, for each type of database (and more precisely for each JDBC driver), the corresponding mapper must be specified in the DataSource. This is the purpose of the datasource.mapper property of the DataSource properties file. Note that all JOnAS-provided DataSource properties files (in JOnAS_ROOT/conf) already contain this property with the correct mapper.

| property name | description | possible values | |
|---|---|---|---|
| datasource.mapper | JORM database mapper | rdb | generic mapper (JDBC standard driver ...) |
| | | rdb.cloudscape | Cloudscape |
| | | rdb.db2 | DB2 |
| | | rdb.firebird | Firebird |
| | | rdb.hsql | HSQL |
| | | rdb.mckoi | McKoi Db |
| | | rdb.mysql | MySQL |
| | | rdb.oracle8 | Oracle 8 and lesser versions |
| | | rdb.oracle | Oracle 9 |

[4] http://www.objectweb.org/jorm/index.html

| property name | description | possible values | |
|---|---|---|---|
| | | rdb.postgres | PostgreSQL (>= 7.2) |
| | | rdb.sapdb | Sap DB |
| | | rdb.sqlserver | MS Sql Server |
| | | rdb.sybase | Sybase |

Contact the JOnAS team to obtain a mapper for other databases.

The container code generated at deployment is now independent of the JORM mappers. Until JOnAS 4.1.4, the container code generated at deployment (GenIC or ejbjar ant task) was dependent on this mapper. It was possible to deploy (generate container code) a bean for several mappers in order to change the database (i.e. the DataSource file) without redeploying the bean. These mappers were specified as the mappernames argument of the GenIC command or as the mappernames attribute of the JOnAS ANT ejbjar task. The value was a comma-separated list of mapper names for which the container classes were generated. These mappernames options are now deprecated.

# 2.5.4. JOnAS Database Mapping (Specific Deployment Descriptor)

The mapping to the database of entity beans and their relationships may be specified in the JOnAS-specific deployment descriptor, in `jonas-entity` elements, and in `jonas-ejb-relation` elements. Since JOnAS is able to generate the database mapping, all the elements of the JOnAS-specific deployment descriptor defined in this section (which are sub-elements of `jonas-entity` or `jonas-ejb-relation` ) are optional, except those for specifying the datasource and the initialization mode (i.e. the `jndi-name` of `jdbc-mapping` and `cleanup` ). The default values of these mapping elements, provided in this section, define the JOnAS-generated database mapping.

## 2.5.4.1. Specifying and Initializing the Database

For specifying the database within which a CMP 2.0 entity bean is stored, the `jndi-name` element of the `jdbc-mapping` is necessary. This is the JNDI name of the DataSource representing the database storing the entity bean.

```
<jdbc-mapping>
  <jndi-name>jdbc_1</jndi-name>
</jdbc-mapping>
```

For a CMP 2.0 entity bean, the JOnAS-specific deployment descriptor contains an additional element, `cleanup` , to be specified before the `jdbc-mapping` element, which can have one of the following values:

| | |
|---|---|
| removedata | at bean loading time, the content of the tables storing the bean data is deleted |
| removeall | at bean loading time, the tables storing the bean data are dropped (if they exist) and created |
| none | do nothing |

| | |
|---|---|
| create | default value (if the element is not specified), at bean loading time, the tables for storing the bean data are created if they do not exist. |

It may be useful for testing purposes to delete the database data each time a bean is loaded. For this purpose, the part of the JOnAS-specific deployment descriptor related to the entity bean may look like the following:

```
<cleanup>removedata</cleanup>
<jdbc-mapping>
  <jndi-name>jdbc_1</jndi-name>
</jdbc-mapping>
```

## 2.5.4.2. CMP2 fields Mapping

Mapping CMP fields in CMP2.0 is similar to that of CMP 1.1, but in CMP2.0 it is also possible to specify the SQL type of a column. Usually this SQL type is used if JOnAS creates the table ( `create` value of the `cleanup` element), and if the JORM default chosen SQL type is not appropriate.

Standard Deployment Descriptor

```
.....
<entity>
  <ejb-name>A</ejb-name>
  .....
  <cmp-field>
    <field-name>idA</field-name>
  </cmp-field>
  <cmp-field>
    <field-name>f</field-name>
  </cmp-field>
.....
</entity>
.....
```

### Table 2.2. Database Mapping for CMP2 Fields

| | |
|---|---|
| t_A | |
| c_idA | c_f |
| ... | ... |

JOnAS Deployment Descriptor

```
.....
<jonas-entity>
  <ejb-name>A</ejb-name>
  .....
  <jdbc-mapping>
    <jndi-name>jdbc_1</jndi-name>
    <jdbc-table-name>t_A</jdbc-table-name>
    <cmp-field-jdbc-mapping>
      <field-name>idA</field-name>
      <jdbc-field-name>c_idA</jdbc-field-name>
    </cmp-field-jdbc-mapping>
    <cmp-field-jdbc-mapping>
      <field-name>f</field-name>
      <jdbc-field-name>c_f</jdbc-field-name>
      <sql-type>varchar(40)</sql-type>
    </cmp-field-jdbc-mapping>
  </jdbc-mapping>
  .....
</jonas-entity>
.....
```

**Table 2.3. Defaults values for JOnAS deployment descriptor**

| | |
|---|---|
| jndi-name | Mandatory |
| jdbc-table-name | Optional. Default value is the upper-case CMP2 abstract-schema-name, or the CMP1 ejb-name, suffixed by _ . |
| cmp-field-jdbc-mapping | Optional. |
| jdbc-field-name | Optional. Default value is the field-name suffixed by _ . idA_ and f_ in the example. |
| sql-type | Optional. Default value defined by JORM. |

# 2.5.4.3. CMR fields Mapping to primary-key-fields (simple pk)

## 2.5.4.3.1. 1-1 unidirectional relationships

### 2.5.4.3.1.1. Standard Deployment Descriptor

```
.....
<entity>
  <ejb-name>A</ejb-name>
  .....
  <cmp-field>
    <field-name>idA</field-name>
  </cmp-field>
  <primkey-field>idA</primkey-field>
  .....
</entity>
.....
<entity>
  <ejb-name>B</ejb-name>
  .....
  <cmp-field>
    <field-name>idB</field-name>
  </cmp-field>
  <primkey-field>idB</primkey-field>
  .....
</entity>
.....
<relationships>
  <ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
    <ejb-relationship-role>
      <!-- A => B -->
            <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>A</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>b</cmr-field-name>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <!-- B => A -->
      <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>B</ejb-name>
      </relationship-role-source>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
.....
```

### 2.5.4.3.1.2. Database Mapping for 1-1 relationship

| t_A | | t_B |
|---|---|---|
| c_idA | cfk_idB | c_idB |

| | | |
|---|---|---|
| ... | ... | ... |

There is a foreign key in the table of the bean that owns the CMR field.

#### 2.5.4.3.1.3. JOnAS Deployment Descriptor

```
.....
<jonas-entity>
  <ejb-name>A</ejb-name>
  .....
  <jdbc-mapping>
    <jndi-name>jdbc_1</jndi-name>
    <jdbc-table-name>t_A/jdbc-table-name>
    <cmp-field-jdbc-mapping>
      <field-name>idA</field-name>
      <jdbc-field-name>c_idA</jdbc-field-name>
    </cmp-field-jdbc-mapping>
  </jdbc-mapping>
  .....
</jonas-entity>
.....
<jonas-entity>
  <ejb-name>B</ejb-name>
  .....
  <jdbc-mapping>
    <jndi-name>jdbc_1</jndi-name>
    <jdbc-table-name>t_B/jdbc-table-name>
    <cmp-field-jdbc-mapping>
      <field-name>idB</field-name>
      <jdbc-field-name>c_idB</jdbc-field-name>
    </cmp-field-jdbc-mapping>
  </jdbc-mapping>
  .....
</jonas-entity>
.....
<jonas-ejb-relation>
  <ejb-relation-name>a-b</ejb-relation-name>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_idb</foreign-key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
</jonas-ejb-relation>
.....
```

foreign-key-jdbc-name is the column name of the foreign key in the table of the source bean of the relationship-role. In this example, where the destination bean has a primary-key-field, it is possible to deduce that this foreign-key-jdbc-name column is to be associated with the column of this primary-key-field in the table of the destination bean.

Default values:

| | |
|---|---|
| jonas-ejb-relation | Optional |
| foreign-key-jdbc-name | Optional. Default value is the abstract-schema-name of the destination bean, suffixed by _ , and by its primary-key-field. B_idb in the example. |

### 2.5.4.3.2. 1-1 bidirectional relationships

Compared to 1-1 unidirectional relationships, there is a CMR field in both of the beans, thus making two types of mapping possible.

#### 2.5.4.3.2.1. Standard Deployment Descriptor

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

```
      <ejb-relation-name>a-b</ejb-relation-name>
      <ejb-relationship-role>
        <!-- A => B -->
              <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
          <ejb-name>A</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>b</cmr-field-name>
        </cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <!-- B => A -->
        <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
          <ejb-name>B</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>a</cmr-field-name>
        </cmr-field>
      </ejb-relationship-role>
    </ejb-relation>
  </relationships>
  .....
```

#### 2.5.4.3.2.2. Database Mapping

Two mappings are possible. One of the tables may hold a foreign key.

Case 1:

| t_A | | t_B |
|---|---|---|
| c_idA | cfk_idB | c_idB |
| ... | ... | ... |

Case 2:

| t_A | t_B | |
|---|---|---|
| c_idA | c_idB | cfk_idA |
| ... | ... | ... |

#### 2.5.4.3.2.3. JOnAS Deployment Descriptor

Case 1:

```
  .....
  <jonas-ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
    <jonas-ejb-relationship-role>
      <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
      <foreign-key-jdbc-mapping>
        <foreign-key-jdbc-name>cfk_idb</foreign-key-jdbc-name>
      </foreign-key-jdbc-mapping>
    </jonas-ejb-relationship-role>
  </jonas-ejb-relation>
  .....
```

Case 2:

```
  .....
  <jonas-ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
    <jonas-ejb-relationship-role>
      <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
```

```
      <foreign-key-jdbc-mapping>
        <foreign-key-jdbc-name>cfk_ida</foreign-key-jdbc-name>
      </foreign-key-jdbc-mapping>
    </jonas-ejb-relationship-role>
  </jonas-ejb-relation>
  .....
```

For the default mapping, the foreign key is in the table of the source bean of the first ejb-relationship-role of the ejb-relation. In the example, the default mapping corresponds to case 1, since the ejb-relationship-role a2b is the first defined in the ejb-relation a-b . Then, the default values are similar to those of the 1-1 unidirectional relationship.

## 2.5.4.3.3. 1-N unidirectional relationships

### 2.5.4.3.3.1. Standard Deployment Descriptor

```
  .....
  <relationships>
    <ejb-relation>
      <ejb-relation-name>a-b</ejb-relation-name>
      <ejb-relationship-role>
        <!-- A => B -->
              <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
          <ejb-name>A</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>b</cmr-field-name>
          <cmr-field-type>java.util.Collection</cmr-field-type>
        </cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <!-- B => A -->
        <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
        <relationship-role-source>
          <ejb-name>B</ejb-name>
        </relationship-role-source>
      </ejb-relationship-role>
    </ejb-relation>
  </relationships>
  .....
```

### 2.5.4.3.3.2. Database Mapping

| t_A | t_B | |
|---|---|---|
| c_idA | c_idB | cfk_idA |
| ... | ... | ... |

In this case, the foreign key must be in the table of the bean which is on the "many" side of the relationship (i.e. in the table of the source bean of the relationship role with multiplicity many), t_B.

### 2.5.4.3.3.3. JOnAS Deployment Descriptor

```
  .....
  <jonas-ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
    <jonas-ejb-relationship-role>
      <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
      <foreign-key-jdbc-mapping>
        <foreign-key-jdbc-name>cfk_ida</foreign-key-jdbc-name>
      </foreign-key-jdbc-mapping>
    </jonas-ejb-relationship-role>
  </jonas-ejb-relation>
  .....
```

Default values:

| | |
|---|---|
| jonas-ejb-relation | Optional |
| foreign-key-jdbc-name | Optional. Default value is the abstract-schema-name of the destination bean of the "one" side of the relationship (i.e. the source bean of the relationship role with multiplicity one), suffixed by _ , and by its primary-key-field. A_ida in the example. |

## 2.5.4.3.4. 1-N bidirectional relationships

Similar to 1-N unidirectional relationships, but with a CMR field in each bean.

### 2.5.4.3.4.1. Standard Deployment Descriptor

```
.....
<relationships>
  <ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
    <ejb-relationship-role>
      <!-- A => B -->
            <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>A</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>b</cmr-field-name>
        <cmr-field-type>java.util.Collection</cmr-field-type>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <!-- B => A -->
      <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>B</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>a</cmr-field-name>
      </cmr-field>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
.....
```

### 2.5.4.3.4.2. Database mapping

| | | |
|---|---|---|
| t_A | t_B | |
| c_idA | c_idB | cfk_idA |
| ... | ... | ... |

In this case, the foreign key must be in the table of the bean which is on the "many" side of the relationship (i.e. in the table of the source bean of the relationship role with multiplicity many), t_B.

### 2.5.4.3.4.3. JOnAS Deployment Descriptor

```
.....
<jonas-ejb-relation>
  <ejb-relation-name>a-b</ejb-relation-name>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_ida</foreign-key-jdbc-name>
    </foreign-key-jdbc-mapping>
```

```
      </jonas-ejb-relationship-role>
    </jonas-ejb-relation>
    .....
```

Default values:

| | |
|---|---|
| jonas-ejb-relation | Optional |
| foreign-key-jdbc-name | Optional. Default value is the abstract-schema-name of the destination bean of the "one" side of the relationship (i.e. the source bean of the relationship role with multiplicity one), suffixed by _ , and by its primary-key-field. A_ida in the example. |

## 2.5.4.3.5. N-1 unidirectional relationships

Similar to 1-N unidirectional relationships, but the CMR field is defined on the "many" side of the relationship, i.e. on the (source bean of the) relationship role with multiplicity "many."

### 2.5.4.3.5.1. Standard Deployment Descriptor

```
    .....
    <relationships>
      <ejb-relation>
        <ejb-relation-name>a-b</ejb-relation-name>
        <ejb-relationship-role>
          <!-- A => B -->
                <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
          <multiplicity>Many</multiplicity>
          <relationship-role-source>
            <ejb-name>A</ejb-name>
          </relationship-role-source>
          <cmr-field>
            <cmr-field-name>b</cmr-field-name>
          </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
          <!-- B => A -->
          <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
          <multiplicity>One</multiplicity>
          <relationship-role-source>
            <ejb-name>B</ejb-name>
          </relationship-role-source>
        </ejb-relationship-role>
      </ejb-relation>
    </relationships>
    .....
```

### 2.5.4.3.5.2. Database mapping

| t_A | | t_B |
|---|---|---|
| c_idA | cfk_idB | c_idB |
| ... | ... | ... |

In this case, the foreign key must be in the table of the bean which is on the "many" side of the relationship (i.e. in table of the source bean of the relationship role with multiplicity many), t_A.

### 2.5.4.3.5.3. JOnAS Deployment Descriptor

```
    .....
    <jonas-ejb-relation>
      <ejb-relation-name>a-b</ejb-relation-name>
      <jonas-ejb-relationship-role>
```

```
      <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
      <foreign-key-jdbc-mapping>
        <foreign-key-jdbc-name>cfk_idb</foreign-key-jdbc-name>
      </foreign-key-jdbc-mapping>
    </jonas-ejb-relationship-role>
  </jonas-ejb-relation>
  .....
```

Default values:

| | |
|---|---|
| jonas-ejb-relation | Optional |
| foreign-key-jdbc-name | Optional. Default value is the abstract-schema-name of the destination bean of the "one" side of the relationship (i.e. the source bean of the relationship role with multiplicity one), suffixed by _ , and by its primary-key-field. B_idb in the example. |

## 2.5.4.3.6. N-M unidirectional relationships

### 2.5.4.3.6.1. Standard Deployment Descriptor

```
  .....
  <relationships>
    <ejb-relation>
      <ejb-relation-name>a-b</ejb-relation-name>
      <ejb-relationship-role>
        <!-- A => B -->
              <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
        <relationship-role-source>
          <ejb-name>A</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>b</cmr-field-name>
          <cmr-field-type>java.util.Collection</cmr-field-type>
        </cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <!-- B => A -->
        <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
        <relationship-role-source>
          <ejb-name>B</ejb-name>
        </relationship-role-source>
      </ejb-relationship-role>
    </ejb-relation>
  </relationships>
  .....
```

### 2.5.4.3.6.2. Database mapping

| | | | |
|---|---|---|---|
| t_A | t_B | tJoin_AB | |
| c_idA | c_idB | cfk_idA | cfk_idB |
| ... | ... | ... | ... |

In this case, there is a join table composed of the foreign keys of each entity bean table.

### 2.5.4.3.6.3. JOnAS Deployment Descriptor

```
  .....
  <jonas-ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
    <jdbc-table-name>tJoin_AB</jdbc-table-name>
    <jonas-ejb-relationship-role>
```

```
      <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
      <foreign-key-jdbc-mapping>
        <foreign-key-jdbc-name>cfk_idb</foreign-key-jdbc-name>
      </foreign-key-jdbc-mapping>
    </jonas-ejb-relationship-role>
    <jonas-ejb-relationship-role>
      <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
      <foreign-key-jdbc-mapping>
        <foreign-key-jdbc-name>cfk_ida</foreign-key-jdbc-name>
      </foreign-key-jdbc-mapping>
    </jonas-ejb-relationship-role>
  </jonas-ejb-relation>
  .....
```

Default values

| | |
|---|---|
| jonas-ejb-relation | Optional |
| jdbc-table-name | Optional. Default value is built from the abstract-schema-names of the beans, separated by _ . A_B in the example. |
| foreign-key-jdbc-name | Optional. Default value is the abstract-schema-name of the destination bean, suffixed by _ , and by its primary-key-field. B_idb and A_ida in the example. |

## 2.5.4.3.7. N-M bidirectional relationships

Similar to N-M unidirectional relationships, but a CMR field is defined for each bean.

### 2.5.4.3.7.1. Standard deployment Descriptor

```
  .....
  <relationships>
    <ejb-relation>
      <ejb-relation-name>a-b</ejb-relation-name>
      <ejb-relationship-role>
        <!-- A => B -->
              <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
        <relationship-role-source>
          <ejb-name>A</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>b</cmr-field-name>
          <cmr-field-type>java.util.Collection</cmr-field-type>
        </cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <!-- B => A -->
        <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
        <relationship-role-source>
          <ejb-name>B</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>a</cmr-field-name>
          <cmr-field-type>java.util.Collection</cmr-field-type>
        </cmr-field>
      </ejb-relationship-role>
    </ejb-relation>
  </relationships>
  .....
```

### 2.5.4.3.7.2. Database mapping

| | | | |
|---|---|---|---|
| t_A | t_B | tJoin_AB | |
| c_idA | c_idB | cfk_idA | cfk_idB |

| ... | ... | ... | ... |
|---|---|---|---|

In this case, there is a join table composed of the foreign keys of each entity bean table.

#### 2.5.4.3.7.3. JOnAS Deployment Descriptor

```
.....
<jonas-ejb-relation>
  <ejb-relation-name>a-b</ejb-relation-name>
  <jdbc-table-name>tJoin_AB</jdbc-table-name>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_idb</foreign-key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_ida</foreign-key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
</jonas-ejb-relation>
.....
```

Default values:

| | |
|---|---|
| jonas-ejb-relation | Optional |
| jdbc-table-name | Optional. Default value is built from the abstract-schema-names of the beans, separated by _ . A_B in the example. |
| foreign-key-jdbc-name | Optional. Default value is the abstract-schema-name of the destination bean, suffixed by _ , and by its primary-key-field. B_idb and A_ida in the example. |

## 2.5.4.4. CMR fields Mapping to composite primary-keys

In the case of composite primary keys, the database mapping should provide the capability to specify which column of a foreign key corresponds to which column of the primary key. This is the only difference between relationships based on simple primary keys. For this reason, not all types of relationship are illustrated below.

### 2.5.4.4.1. 1-1 bidirectional relationships

#### 2.5.4.4.1.1. Standard Deployment Descriptor

```
.....
<entity>
  <ejb-name>A</ejb-name>
  .....
  <prim-key-class>p.PkA</prim-key-class>
  .....
  <cmp-field>
    <field-name>id1A</field-name>
  </cmp-field>
  <cmp-field>
    <field-name>id2A</field-name>
  </cmp-field>
  .....
</entity>
.....
<entity>
  <ejb-name>B</ejb-name>
  .....
```

```
    <prim-key-class>p.PkB</prim-key-class>
    .....
    <cmp-field>
      <field-name>id1B</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>id2B</field-name>
    </cmp-field>
    .....
  </entity>
  .....
  <relationships>
    <ejb-relation>
      <ejb-relation-name>a-b</ejb-relation-name>
      <ejb-relationship-role>
        <!-- A => B -->
              <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
          <ejb-name>A</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>b</cmr-field-name>
        </cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <!-- B => A -->
        <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
          <ejb-name>B</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>a</cmr-field-name>
        </cmr-field>
      </ejb-relationship-role>
    </ejb-relation>
  </relationships>
  .....
```

#### 2.5.4.4.1.2. Database mapping

Two mappings are possible, one or another of the tables may hold the foreign key.

Case 1:

| t_A | | | | t_B | |
|---|---|---|---|---|---|
| c_id1A | c_id2A | cfk_id1B | cfk_id2B | c_id1B | c_id2B |
| ... | ... | ... | ... | ... | ... |

Case 2:

| t_A | | t_B | | | |
|---|---|---|---|---|---|
| c_id1A | c_id2A | c_id1B | c_id2B | cfk_id1A | cfk_id2A |
| ... | ... | ... | ... | ... | ... |

#### 2.5.4.4.1.3. JOnAS Deployment Descriptor

Case 1:

```
  .....
  <jonas-ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
    <jonas-ejb-relationship-role>
      <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
      <foreign-key-jdbc-mapping>
        <foreign-key-jdbc-name>cfk_id1b</foreign-key-jdbc-name>
        <key-jdbc-name>c_id1b</key-jdbc-name>
      </foreign-key-jdbc-mapping>
```

```
      <foreign-key-jdbc-mapping>
        <foreign-key-jdbc-name>cfk_id2b</foreign-key-jdbc-name>
        <key-jdbc-name>c_id2b</key-jdbc-name>
      </foreign-key-jdbc-mapping>
    </jonas-ejb-relationship-role>
  </jonas-ejb-relation>
  .....
```

Case 2:

```
  .....
  <jonas-ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
    <jonas-ejb-relationship-role>
      <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
      <foreign-key-jdbc-mapping>
        <foreign-key-jdbc-name>cfk_id1a</foreign-key-jdbc-name>
        <key-jdbc-name>c_id1a</key-jdbc-name>
      </foreign-key-jdbc-mapping>
      <foreign-key-jdbc-mapping>
        <foreign-key-jdbc-name>cfk_id2a</foreign-key-jdbc-name>
        <key-jdbc-name>c_id2a</key-jdbc-name>
      </foreign-key-jdbc-mapping>
    </jonas-ejb-relationship-role>
  </jonas-ejb-relation>
  .....
```

For the default mapping (values), the foreign key is in the table of the source bean of the first ejb-relationship-role of the ejb-relation. In the example, the default mapping corresponds to case 1, since the ejb-relationship-role a2b is the first defined in the ejb-relation a-b.

## 2.5.4.4.2. N-M unidirectional relationships

### 2.5.4.4.2.1. Standard Deployment Descriptor

```
  .....
  <entity>
    <ejb-name>A</ejb-name>
    .....
    <cmp-field>
      <field-name>id1A</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>id2A</field-name>
    </cmp-field>
    .....
  </entity>
  .....
  <entity>
    <ejb-name>B</ejb-name>
    .....
    <cmp-field>
      <field-name>id1B</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>id2B</field-name>
    </cmp-field>
    .....
  </entity>
  .....
  <relationships>
    <ejb-relation>
      <ejb-relation-name>a-b</ejb-relation-name>
      <ejb-relationship-role>
        <!-- A => B -->
              <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
        <relationship-role-source>
          <ejb-name>A</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>b</cmr-field-name>
          <cmr-field-type>java.util.Collection</cmr-field-type>
        </cmr-field>
```

```
      </ejb-relationship-role>
      <ejb-relationship-role>
        <!-- B => A -->
        <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
        <relationship-role-source>
          <ejb-name>B</ejb-name>
        </relationship-role-source>
      </ejb-relationship-role>
    </ejb-relation>
  </relationships>
  .....
```

#### 2.5.4.4.2.2. Database mapping

| t_A | | t_B | | tJoin_AB | | | |
|---|---|---|---|---|---|---|---|
| c_id1A | c_id2A | c_id1B | c_id2B | cfk_id1A | cfk_id2A | cfk_id1B | cfk_id2B |
| ... | ... | ... | ... | ... | ... | ... | ... |

In this case, there is a join table composed of the foreign keys of each entity bean table.

#### 2.5.4.4.2.3. JOnAS Deployment Descriptor

```
  .....
  <jonas-ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
    <jdbc-table-name>tJoin_AB</jdbc-table-name>
    <jonas-ejb-relationship-role>
      <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
      <foreign-key-jdbc-mapping>
        <foreign-key-jdbc-name>cfk_id1b</foreign-key-jdbc-name>
        <key-jdbc-name>c_id1b</key-jdbc-name>
      </foreign-key-jdbc-mapping>
      <foreign-key-jdbc-mapping>
        <foreign-key-jdbc-name>cfk_id2b</foreign-key-jdbc-name>
        <key-jdbc-name>c_id2b</key-jdbc-name>
      </foreign-key-jdbc-mapping>
    </jonas-ejb-relationship-role>
    <jonas-ejb-relationship-role>
      <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
      <foreign-key-jdbc-mapping>
        <foreign-key-jdbc-name>cfk_id1a</foreign-key-jdbc-name>
        <key-jdbc-name>c_id1a</key-jdbc-name>
      </foreign-key-jdbc-mapping>
      <foreign-key-jdbc-mapping>
        <foreign-key-jdbc-name>cfk_id2a</foreign-key-jdbc-name>
        <key-jdbc-name>c_id2a</key-jdbc-name>
      </foreign-key-jdbc-mapping>
    </jonas-ejb-relationship-role>
  </jonas-ejb-relation>
  .....
```

# 2.6. Configuring JDBC DataSources with 'dbm' service

This section describes how to configure the Datasources for connecting application to databases when the **dbm** service is used.

## 2.6.1. Configuring DataSources

For both container-managed or bean-managed persistence, JOnAS makes use of relational storage systems through the JDBC interface. JDBC connections are obtained from an object, the `DataSource`, provided at the application server level. The `DataSource` interface is defined in the JDBC standard extensions.

A `DataSource` object identifies a database and a means to access it via JDBC (a JDBC driver). An application server may request access to several databases and thus provide the corresponding `DataSource` objects that will be registered in JNDI registry.

This section explains how `DataSource` objects can be defined and configured in the JOnAS server.

JOnAS provides a generic driver-wrapper that emulates the `XADataSource` interface on a regular JDBC driver. It is important to note that this driver-wrapper does not ensure a real two-phase commit for distributed database transactions.

Neither the EJB specification nor the Java EE specification describe how to define `DataSource` objects so that they are available to a Java EE platform. Therefore, this document, which describes how to define and configure DataSource objects, is specific to JOnAS. However, the way to use these `DataSource` objects in the Application Component methods is standard, that is, by using the resource manager connection factory references (refer to the example in the section Writing database access operations[5] of the Developing Entity Bean Guide[6]).

A `DataSource` object should be defined in a file called <DataSource name>.properties (for example Oracle1.properties for an Oracle DataSource or Postgres.properties for an PostgreSQL DataSource.These files must be located in `$JONAS_BASE/conf` directory.

In the `jonas.properties` file, to define a DataSource "Oracle1.properties" add the name "Oracle1" to the line onas.service.dbm.datasources, as follows:

```
jonas.service.dbm.datasources Oracle1, Sybase, PostgreSQL
```

The property file defining a `DataSource` may contain two types of information:

- connection properties
- JDBC Connection Pool properties

## 2.6.1.1. connection properties

| property name | Description |
|---|---|
| datasource.name | JNDI name of the DataSource |
| datasource.url | The JDBC database URL : jdbc:<database_vendor_subprotocol>:... |
| datasource.classname | Name of the class implementing the JDBC driver |
| datasource.username | Database user name |
| datasource.password | Database user password |
| datasource.isolationLevel | Database isolation level for transactions. Possible values are:<br>• none,<br>• serializable,<br>• read_committed,<br>• read_uncommitted,<br>• repeatable_read<br>The default depends on the database used. |

[5] ejb2_programmer_guide.html#ejb2.bmp
[6] ejb2_programmer_guide.html#ejb2.entity

| | |
|---|---|
| datasource.mapper | JORM database mapper (for possible values see here)[7] |

**Note**

If this datasource is used as a persistence unit, the persistence configuration defined in the `persistence.xml` file must be coherent to those properties, such as the datasource name and the dialect.

## 2.6.1.2. Connection Pool properties

Each `Datasource` is implemented as a connection manager and manages a pool of JDBC connections.

The pool can be configured via some additional properties described in the following table.

All these settings have default values and are not required. All these attributes can be reconfigured when JOnAS is running, with the console JonasAdmin.

| property | Description | Default value |
|---|---|---|
| jdbc.connchecklevel | JDBC connection checking level:<br>• 0 : no check<br>• 1: check connection still open<br>• 2: call the test statement before reusing a connection from the pool | 1 |
| jdbc.connteststmt | test statement in case jdbc.connchecklevel = 2. | select 1 |
| jdbc.connmaxage | nb of minutes a connection can be kept in the pool. After this time, the connection will be closed, if minconpool limit has not been reached. | 1440 mn (= 1 day) |
| jdbc.maxopentime | Maximum time (in mn) a connection can be left busy. If the caller has not issued a close() during this time, the connection will be closed automatically. | 1440 mn (= 1 day) |
| jdbc.minconpool | Minimum number of connections in the pool. Setting a positive value here ensures that the pool size will not go below this limit during the datasource lifetime. | 0 |
| jdbc.maxconpool | Maximum number of connections in the pool. Limiting the max pool size avoids errors from the database. | no limit |
| jdbc.samplingperiod | Sampling period for JDBC monitoring. nb of seconds between 2 measures. | 60 sec |

| | | |
|---|---|---|
| jdbc.maxwaittime | Maximum time (in seconds) to wait for a connection in case of shortage. This is valid only if maxconpool has been set. | 10 sec |
| jdbc.maxwaiters | Maximum of concurrent waiters for a JDBC Connection. This is valid only if maxconpool has been set. | 1000 |
| jdbc.pstmtmax | Maximum number of prepared statements cached in a Connection. Setting this to a bigger value (120 for example) will lead to better performance, but will use more memory. The recommendation is to set this value to the number of different queries that are used the most often. This is to be tuned by administrators. | 12 |

When a user requests a jdbc connection, the **dbm** connection manager first checks to see if a connection is already open for its transaction. If not, it tries to get a free connection from the free list. If there are no more connections available, the **dbm** connection manager creates a new jdbc connection (if jdbc.maxconpool is not reached).

If it cannot create new connections, the user must wait (if jdbc.maxwaiters is not reached) until a connection is released. After a limited time (jdbc.maxwaittime), the `getConnection` returns an exception.

When the user calls `close()` on its connection, it is put back in the free list.

Many statistics are computed (every jdbc.samplingperiod seconds) and can be viewed by JonasAdmin. This is useful for tuning these parameters and for seeing the server load at any time.

When a connection has been open for too long a time (jdbc.connmaxage), the pool will try to release it from the freelist. However, the **dbm** connection manager always tries to keep open at least the number of connections specified in jdbc.minconpool.

When the user has forgotten to close a jdbc connection, the system can automatically close it, after jdbc.maxopentime minutes. Note that if the user tries to use this connection later, thinking it is still open, it will return an exception (socket closed).

When a connection is reused from the freelist, it is possible to verify that it is still valid. This is configured in jdbc.connchecklevel. The maximum level is to try a dummy statement on the connection before returning it to the caller. This statement is configured in jdbc.conntestst mt

## 2.6.1.3. DataSource example:

Here is the template for an Oracle dataSource.properties file that can be found in $JONAS_ROOT/conf:

```
###################### Oracle DataSource configuration example
#


#####
#  DataSource configuration
#
datasource.name  jdbc_1
datasource.url  jdbc:oracle:thin:@<your-hostname>:1521:<your-db>
```

```
datasource.classname oracle.jdbc.driver.OracleDriver
datasource.username <your-username>
datasource.password <user-password>
datasource.mapper rdb.oracle


#####
#  ConnectionManager configuration
#

#  JDBC connection checking level.
#     0 = no special checking
#     1 = check physical connection is still open before reusing it
#     2 = try every connection before reusing it
jdbc.connchecklevel 0

#  Max age for jdbc connections
#  nb of minutes a connection can be kept in the pool
jdbc.connmaxage 1440

# Maximum time (in mn) a connection can be left busy.
# If the caller has not issued a close() during this time, the connection
# will be closed automatically.
jdbc.maxopentime 60

#  Test statement
jdbc.connteststmt select * from dual

# JDBC Connection Pool size.
# Limiting the max pool size avoids errors from database.
jdbc.minconpool 10
jdbc.maxconpool 30

# Sampling period for JDBC monitoring :
# nb of seconds between 2 measures.
jdbc.samplingperiod 30

# Maximum time (in seconds) to wait for a connection in case of shortage.
# This may occur only when maxconpool is reached.
jdbc.maxwaittime 5

# Maximum of concurrent waiters for a JDBC Connection
# This may occur only when maxconpool is reached.
jdbc.maxwaiters 100
```

# Chapter 3. Developing Message Driven Beans

## 3.1. EJB Programmer's Guide: Message-drivenBeans

The EJB 2.1 specification defines a new kind of EJB component for receiving asynchronous messages. This implements some type of "asynchronous EJB component method invocation" mechanism. The Message-driven Bean (also referred to as MDB in the following) is an Enterprise JavaBean, not an Entity Bean or a Session Bean, which plays the role of a JMS MessageListener.

The EJB 2.1 specification contains detailed information about MDB. The Java Message Service Specification 1.1 contains detailed information about JMS. This chapter focuses on the use of Message-driven beans within the JOnAS server.

### 3.1.1. Description of a Message-driven Bean

A Message-driven Bean is an EJB component that can be considered as a JMS MessageListener, i.e., processing JMS messages asynchronously; it implements the onMessage(javax.jms.Message) method, defined in the javax.jms.MessageListener interface. It is associated with a JMS destination, i.e., a Queue for "point-to-point" messaging or a Topic for "publish/subscribe." The onMessage method is activated on receipt of messages sent by a client application to the corresponding JMS destination. It is possible to associate a JMS message selector to filter the messages that the Message-driven Bean should receive.

JMS messages do not carry any context, thus the onMessage method will execute without pre-existing transactional context. However, a new transaction can be initiated at this moment (refer to the Section 3.1.5, “Transactional aspects” section for more details). The onMessage method can call other methods on the MDB itself or on other beans, and can involve other resources by accessing databases or by sending messages. Such resources are accessed the same way as for other beans (entity or session), i.e., through resource references declared in the deployment descriptor.

The JOnAS container maintains a pool of MDB instances, allowing large volumes of messages to be processed concurrently. An MDB is similar in some ways to a stateless session bean: its instances are relatively short-lived, it retains no state for a specific client, and several instances may be running at the same time.

### 3.1.2. Developing a Message-drivenBean

The MDB class must implement the javax.jms.MessageListener and the javax.ejb.MessageDrivenBean interfaces. In addition to the onMessage method, the following must be implemented:

- A public constructor with no argument.
- public void ejbCreate() : with no arguments, called at the bean instantiation time. It may be used to allocate some resources, such as connection factories, for example if the bean sends messages, or datasources or if the bean accesses databases.
- public void ejbRemove() : usually used to free the resources allocated in the ejbCreate method.
- public void setMessageDrivenContext(MessageDrivenContext mdc) : called by the container after the instance creation, with no transaction context. The JOnAS container provides the bean with a container context that can be used for transaction management, e.g., for calling setRollbackOnly(), getRollbackOnly(), getUserTransaction() .

The following is an example of an MDB class:

```
public class MdbBean  implements MessageDrivenBean, MessageListener {

    private transient MessageDrivenContext mdbContext;

    public MdbBean() {}

    public void setMessageDrivenContext(MessageDrivenContext ctx) {
        mdbContext = ctx;
    }

    public void ejbRemove() {}

    public void ejbCreate() {}

    public void onMessage(Message message) {
        try {
            TextMessage mess = (TextMessage)message;
            System.out.println( "Message received: "+mess.getText());
        }catch(JMSException ex){
            System.err.println("Exception caught: "+ex);
        }
    }
}
```

The destination associated to an MDB is specified in the deployment descriptor of the bean. A destination is a JMS-administered object, accessible via JNDI. The description of an MDB in the EJB 2.0 deployment descriptor contains the following elements, which are specific to MDBs:

- the JMS acknowledgment mode: auto-acknowledge or dups-ok-acknowledge (refer to the JMS specification for the definition of these modes)
- an eventual JMS message selector: this is a JMS concept which allows the filtering of the messages sent to the destination
- a message-driven-destination, which contains the destination type (Queue or Topic) and the subscription durability (in case of Topic)

The following example illustrates such a deployment descriptor:

```
<enterprise-beans>
  <message-driven>
    <description>Describe here the message driven bean Mdb</description>
    <display-name>Message Driven Bean Mdb</display-name>
    <ejb-name>Mdb</ejb-name>
    <ejb-class>samplemdb.MdbBean</ejb-class>
    <transaction-type>Container</transaction-type>
    <message-selector>Weight >= 60.00 AND LName LIKE 'Sm_th'</message-selector>
    <message-driven-destination>
      <destination-type>javax.jms.Topic</destination-type>
      <subscription-durability>NonDurable</subscription-durability>
    </message-driven-destination>
    <acknowledge-mode>Auto-acknowledge</acknowledge-mode>
  </message-driven>
</enterprise-beans>
```

If the transaction type is "container," the transactional behavior of the MDB's methods are defined as for other enterprise beans in the deployment descriptor, as in the following example:

```
<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>Mdb</ejb-name>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
```

```
    </container-transaction>
  </assembly-descriptor>
```

For the onMessage method, only the Required or NotSupported transaction attributes must be used, since there can be no pre-existing transaction context.

For the message selector specified in the previous example, the sent JMS messages are expected to have two properties, "Weight" and "LName," for example assigned in the JMS client program sending the messages, as follows:

```
message.setDoubleProperty("Weight",75.5);
message.setStringProperty("LName","Smith");
```

Such a message will be received by the Message-driven bean. The message selector syntax is based on a subset of the SQL92. Only messages whose headers and properties match the selector are delivered. Refer to the JMS specification for more details.

The JNDI name of a destination associated with an MDB is defined in the JOnAS-specific deployment descriptor, within a jonas-message-driven element, as illustrated in the following:

```
<jonas-message-driven>
  <ejb-name>Mdb</ejb-name>
  <jonas-message-driven-destination>
    <jndi-name>sampleTopic</jndi-name>
  </jonas-message-driven-destination>
</jonas-message-driven>
```

Once the destination is established, a client application can send messages to the MDB through a destination object obtained via JNDI as follows:

```
Queue q = context.lookup("sampleTopic");
```

If the client sending messages to the MDB is an EJB component itself, it is preferable that it use a resource environment reference to obtain the destination object. The use of resource environment references is described in the JMS User's Guide (Writing JMS operations within an application component / Accessing the destination object section).

## 3.1.3. Administration aspects

It is assumed at this point that the JOnAS server will make use of an existing JMS implementation, e.g., Joram, SwiftMQ.

The default policy is that the MDB developer and deployer are not concerned with JMS administration. This means that the developer/deployer will not create or use any JMS Connection factories and will not create a JMS destination (which is necessary for performing JMS operations within an EJB component, refer to the JMS User's Guide); they will simply define the type of destination in the deployment descriptor and identify its JNDI name in the JOnAS-specific deployment descriptor, as described in the previous section. This means that JOnAS will implicitly create the necessary administered objects by using the proprietary administration APIs of the JMS implementation (since the administration APIs are not standardized). To perform such administration operations, JOnAS uses wrappers to the JMS provider administration API. For Joram, the wrapper is org.objectweb.jonas_jms.JmsAdminForJoram (which is the default wrapper class defined by the jonas.service.jms.mom property in the jonas.properties file). For SwiftMQ, a com.swiftmq.appserver.jonas.JmsAdminForSwiftMQ class can be obtained from the SwiftMQ [1] site.

[1] http://www.swiftmq.com/

For the purpose of this implicit administration phase, the deployer must add the 'jms' service in the list of the JOnAS services. For the example provided, the jonas.properties file should contain the following:

```
// The jms service must be added
jonas.services                registry,security,jtm,dbm,jms,ejb
jonas.service.ejb.descriptors  samplemdb.jar
jonas.service.jms.topics       sampleTopic    // not mandatory
```

The destination objects may or may not pre-exist. The EJB server will not create the corresponding JMS destination object if it already exists. (Refer also to the JMS administration guide). The `sampleTopic` should be explicitly declared only if the JOnAS Server is going to create it first, even if the Message-driven bean is not loaded, or if it is used by another client before the Message-driven bean is loaded. In general, it is not necessary to declare the `sampleTopic` .

JOnAS uses a pool of threads for executing Message-driven bean instances on message reception, thus allowing large volumes of messages to be processed concurrently. As previously explained, MDB instances are stateless and several instances may execute concurrently on behalf of the same MDB. The default size of the pool of thread is 10, and it may be customized via the jonas property jonas.service.ejb.mdbthreadpoolsize , which is specified in the jonas.properties file as in the following example:

```
           jonas.service.ejb.mdbthreadpoolsize   50
```

# 3.1.4. Running a Message-driven Bean

To deploy and run a Message-driven Bean, perform the following steps:

- Verify that a registry is running.
- Start the Message-Oriented Middleware (the JMS provider implementation). Refer to the section " Section 3.1.4.1, “Launching the Message-OrientedMiddleware”."
- Create and register in JNDI the JMS destination object that will be used by the MDB.

  This can be done automatically by the JMS service or explicitly by the proprietary administration facilities of the JMS provider (See JMS administration). The JMS service creates the destination object if this destination is declared in the jonas.properties file (as specified in the previous section).
- Deploy the MDB component in JOnAS.

  Note that, if the destination object is not already created when deploying an MDB, the container asks the JMS service to create it based on the deployment descriptor content.
- Run the EJB client application.
- Stop the application.

  When using JMS, it is very important to stop JOnAS using the jonas stop command; it should not be stopped directly by killing it.

## 3.1.4.1. Launching the Message-OrientedMiddleware

If the configuration property jonas.services contains the jms service, then the JOnAS JMS service will be launched and may try to launch a JMS implementation (a MOM).

For launching the MOM, three possibilities can be considered:

1. Launching the MOM in the same JVM as JOnAS

   This is the default situation obtained by assigning the `true` value to the configuration property jonas.service.jms.collocated in the jonas.properties file.

   ```
   // The jms service must be in the list
   jonas.services               security,jtm,dbm,jms,ejb
   jonas.service.jms.collocated  true
   ```

   In this case, the MOM is automatically launched by the JOnAS JMS service at the JOnAS launching time (command `jonas start` ).

2. Launching the MOM in a separate JVM

   The Joram MOM can be launched using the command:

   `JmsServer`

   For other MOMs, the proprietary command should be used.

   The configuration property jonas.service.jms.collocated must be set to `false` in the jonas.properties file. Setting this property is sufficient if the JORAM's JVM runs on the same host as JONAS, and if the MOM is launched with its default options (unchanged a3servers.xml configuration file under JONAS_BASE/conf or JONAS_ROOT/conf if JONAS_BASE not defined).

   ```
   // The jms service must be in the list
   jonas.services              security,jtm,dbm,jms,ejb
   jonas.service.jms.collocated false
   ```

   To use a specific configuration for the MOM, such as changing the default host (which is localhost) or the default connection port number (which is 16010), requires defining the additional jonas.service.jms.url configuration property as presented in the following case.

3. Launching the MOM on another host

   This requires defining the jonas.service.jms.url configuration property. When using Joram, its value should be the Joram URL `joram://host:port` where `host` is the host name, and `port` is the connection port (by default, 16010). For SwiftMQ, the value of the URL is similar to the following: `smqp://host:4001/timeout=10000` .

   ```
   // The jms service must be in the list
   jonas.services               security,jtm,dbm,jms,ejb
   jonas.service.jms.collocated  false
   jonas.service.jms.url         joram://host2:16010
   ```

- Change Joram default configuration

  As mentioned previously, the default host or default connection port number may need to be changed. This requires modifying the a3servers.xml configuration file provided by the JOnAS delivery in JONAS_ROOT/conf directory. For this, JOnAS must be configured with the property jonas.service.jms.collocated set to `false` , and the property jonas.service.jms.url set to `joram://host:port` . Additionally, the MOM must have been previously launched with the JmsServer command. This command defines a `Transaction` property set to `fr.dyade.aaa.util.NullTransaction` . If the messages need to be persistent, replace the `-DTransaction=fr.dyade.aaa.util.NullTransaction` option with the `-DTransaction=fr.dyade.aaa.util.NTransaction` option. Refer to the Joram

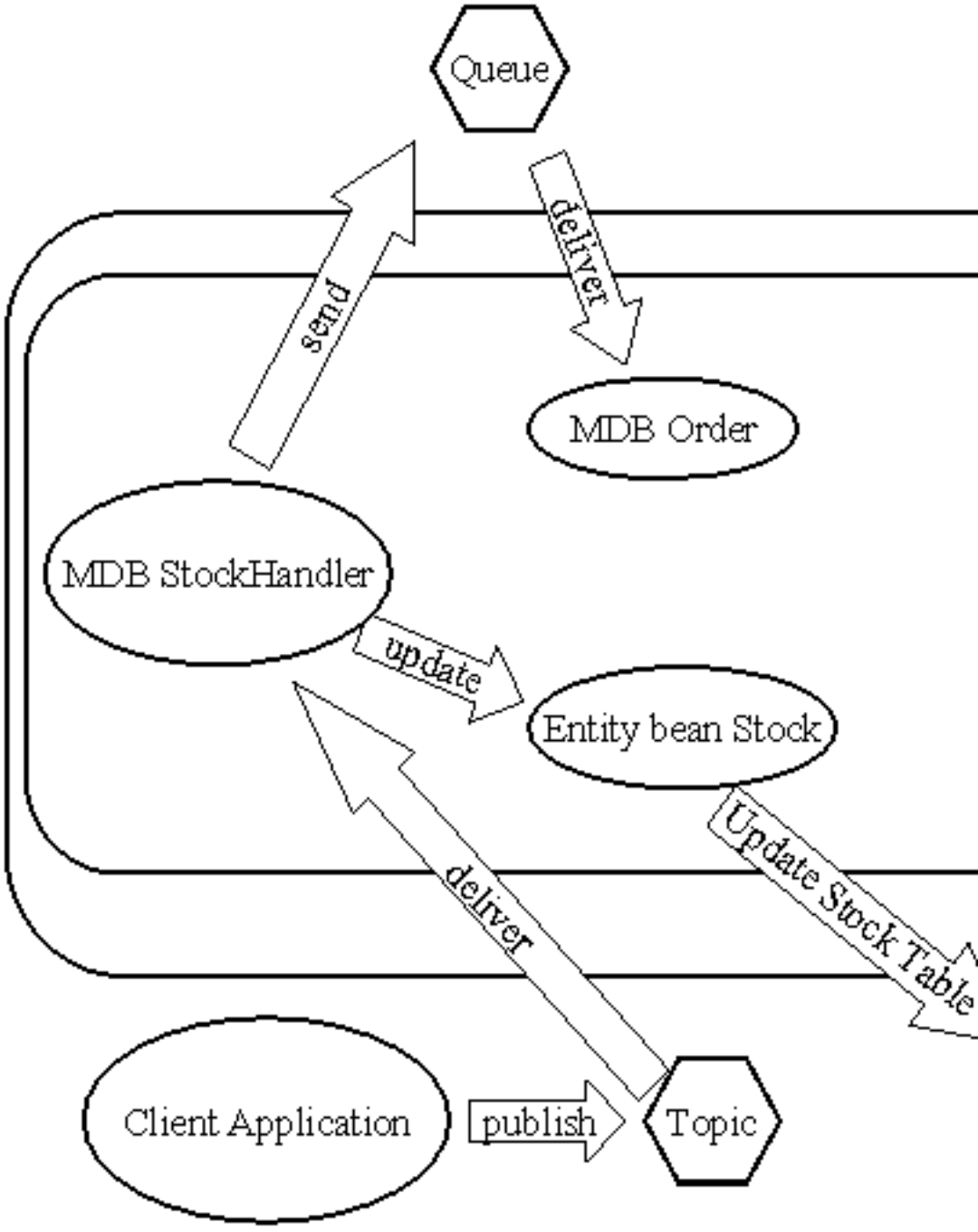
documentation for more details about this command. To define a more complex configuration (e.g., distribution, multi-servers), refer to the Joram documentation on http://joram.objectweb.org [2] .

## 3.1.5. Transactional aspects

Because a transactional context cannot be carried by a message (according to the EJB 2.0 specification), an MDB will never execute within an existing transaction. However, a transaction may be started during the onMessage method execution, either due to a "required" transaction attribute (container-managed transaction) or because it is explicitly started within the method (if the MDB is bean-managed transacted). In the second case, the message receipt will not be part of the transaction. In the first case, container-managed transaction, the container will start a new transaction before de-queueing the JMS message (the receipt of which will, thus, be part of the started transaction), then enlist the resource manager associated with the arriving message and all the resource managers accessed by the onMessage method. If the onMessage method invokes other enterprise beans, the container passes the transaction context with the invocation. Therefore, the transaction started at the onMessage method execution may involve several operations, such as accessing a database (via a call to an entity bean, or by using a "datasource" resource), or sending messages (by using a "connection factory" resource).

## 3.1.6. Example

JOnAS provides examples that are located in the examples/src/mdb install directory. samplemdb is a very simple example, the code of which is used in the previous topics for illustrating how to use Message-driven beans. sampleappli is a more complex example that shows how the sending of JMS messages and updates in a database via JDBC may be involved in the same distributed transaction. The following figure illustrates the architecture of this example application.


Queue
deliver
send
MDB Order
MDB StockHandler
update
Entity bean Stock
Update Stock Table
deliver
Client Application
publish
Topic

There are two Message-driven beans in this example:

- `StockHandlerBean` is a Message-driven bean listening to a topic and receiving Map messages. The `onMessage` method runs in the scope of a transaction started by the container. It sends a Text message on a Queue (OrdersQueue) and updates a Stock element by decreasing the stock quantity. If the stock quantity becomes negative, an exception is received and the current transaction is marked for rollback.
- `OrderBean` is another Message-driven bean listening on the OrdersQueue Queue. On receipt of a Text message on this queue, it writes the corresponding String as a new line in a file ("Order.txt").

The example also includes a CMP entity bean `Stock` that handles a stock table.

A Stock item is composed of a Stockid (String), which is the primary key, and a Quantity (int). The method decreaseQuantity(int qty) decreases the quantity for the corresponding stockid, but can throw a RemoteException "Negative stock."

The client application `SampleAppliClient` is a JMS Client that sends several messages on the topic StockHandlerTopic . It uses Map messages with three fields: "CustomerId," "ProductId," "Quantity." Before sending messages, this client calls the `EnvBean` for creating the StockTable in the database with known values in order to check the results of updates at the end of the test. Eleven messages are sent, the corresponding transactions are committed, and the last message sent causes the transaction to be rolled back.

## 3.1.6.1. Compiling this example

To compile `examples/src/mdb/sampleappli` , use Ant with the `$JONAS_ROOT/examples/src/build.xml` file.

## 3.1.6.2. Running this example

The default configuration of the JMS service in `jonas.properties` is the following:

```
// The jms service must be added
jonas.services                jmx,security,jtm,dbm,jms,ejb
jonas.service.ejb.descriptors  sampleappli.jar
jonas.service.jms.topics       StockHandlerTopic
jonas.service.jms.queues       OrdersQueue
jonas.service.jms.collocated   true
```

This indicates that the JMS Server will be launched in the same JVM as the JOnAS Server, and the JMS-administered objects `StockHandlerTopic` (Topic) and `OrdersQueue` (Queue) will be created and registered in JNDI, if not already existing.

- Run the JOnAS Server.

```
jonas start
```

- Deploy the sampleappli container.

```
jonas admin -a sampleappli.jar
```

- Run the EJB client.

```
jclient sampleappli.SampleAppliClient
```

- Stop the server.

```
jonas stop
```

# 3.2. Tuning Message-driven Bean Pool

A pool is handled by JOnAS for each Message-driven bean. The pool can be configured in the JOnAS-specific deployment descriptor with the following tags:

## 3.2.1. min-pool-size

This optional integer value represents the minimum instances that will be created in the pool when the bean is loaded. This will improve bean instance creation time, at least for the first beans. The default value is 0.

## 3.2.2. max-cache-size

This optional integer value represents the maximum number of instances of ServerSession that may be created in memory. The purpose of this value is to keep JOnAS scalable. The policy is the following: When the ConnectionConsumer ask for a ServerSession instance (in order to deliver a new message) JOnAS tries to give an instance from the ServerSessionPool . If the pool is empty, a new instance is created only if the number of yet created instances is smaller than the max-cache-size parameter. When the max-cache-size is reached, the ConnectionConsumer is blocked and it cannot deliver new messages until a ServerSession is eventually returned in the pool. A ServerSession is pushed into the pool at the end of the onMessage method. The default value is no limit (this means that a new instance of ServerSession is always created when the pool is empty).

The values for `max-cache-size` should be set accordingly to `jonas.service.ejb.maxworkthreads` value. See Configuring JMS Service [3] .

## 3.2.3. example

```
<jonas-ejb-jar>
  <jonas-message-driven>
    <ejb-name>Mdb</ejb-name>
    <jndi-name>mdbTopic</jndi-name>
    <max-cache-size>20</max-cache-size>
    <min-pool-size>10</min-pool-size>
  </jonas-message-driven>
</jonas-ejb-jar>
```

[3] configuration_guide.html#N10D43

# Chapter 4. General Issues Around EJB 2.1

## 4.1. EJB2 Transactional Behaviour

### 4.1.1. Declarative Transaction Management

For container-managed transaction management, the transactional behaviour of an enterprise bean is defined at configuration time and is part of the assembly-descriptor element of the standard deployment descriptor. It is possible to define a common behaviour for all the methods of the bean, or to define the behaviour at the method level. This is done by specifying a transactional attribute, which can be one of the following:

- NotSupported: if the method is called within a transaction, this transaction is suspended during the time of the method execution.
- Required : if the method is called within a transaction, the method is executed in the scope of this transaction, else, a new transaction is started for the execution of the method, and committed before the method result is sent to the caller.
- RequiresNew : the method will always be executed within the scope of a new transaction. The new transaction is started for the execution of the method, and committed before the method result is sent to the caller. If the method is called within a transaction, this transaction is suspended before the new one is started and resumed when the new transaction has completed.
- Mandatory : the method should always be called within the scope of a transaction, else the container will throw the TransactionRequired exception.
- Supports : the method is invoked within the caller transaction scope; if the caller does not have an associated transaction, the method is invoked without a transaction scope.
- Never : The client is required to call the bean without any transaction context; if it is not the case, a java.rmi.RemoteException is thrown by the container.

This is illustrated in the following table:

| **Transaction Attribute** | **Client transaction** | **Transaction associated with enterprise Bean method** |
|---|---|---|
| NotSupported | - | - |
| | T1 | - |
| Required | - | T2 |
| | T1 | T1 |
| RequiresNew | - | T2 |
| | T1 | T2 |
| Mandatory | - | ERROR |
| | T1 | T1 |
| Supports | - | - |
| | T1 | T1 |
| Never | - | - |

| Transaction Attribute | Client transaction | Transaction associated with enterprise Bean method |
|---|---|---|
| | T1 | ERROR |

In the deployment descriptor, the specification of the transactional attributes appears in the assembly-descriptor as follows:

```
  <assembly-descriptor>
    <container-transaction>
      <method>
      <ejb-name>AccountImpl</ejb-name>
      <method-name>*</method-name>
      </method>
      <trans-attribute>Supports</trans-attribute>
    </container-transaction>
    <container-transaction>
      <method>
      <ejb-name>AccountImpl</ejb-name>
      <method-name>getBalance</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
    <container-transaction>
      <method>
      <ejb-name>AccountImpl</ejb-name>
      <method-name>setBalance</method-name>
      </method>
      <trans-attribute>Mandatory</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
```

In this example, for all methods of the AccountImpl bean which are not explicitly specified in a container-transaction element, the default transactional attribute is Supports (defined at the bean-level), and the transactional attributes are Required and Mandatory (defined at the method-name level) for the methods getBalance and setBalance respectively.

## 4.1.2. Bean-managed Transaction

A bean that manages its transactions itself must set the `transaction-type` element in its standard deployment descriptor to:

```
  <transaction-type>Bean</transaction-type>
```

To demarcate the transaction boundaries in a bean with bean-managed transactions, the bean programmer should use the javax.transaction.UserTransaction interface, which is defined on an EJB server object that may be obtained using the EJBContext.getUserTransaction() method (the SessionContext object or the EntityContext object depending on whether the method is defined on a session or on an entity bean). The following example shows a session bean method "doTxJob" demarcating the transaction boundaries; the UserTransaction object is obtained from the sessionContext object, which should have been initialized in the setSessionContext method (refer to the example of the session bean [1] ).

```
public void doTxJob() throws  RemoteException {
     UserTransaction ut = sessionContext.getUserTransaction();
     ut.begin();
     ... // transactional operations
     ut.commit();
}
```

[1] ejb2_programmer_guide.html#ejb2.session.example

Another way to do this is to use JNDI and to retrieve UserTransaction with the name `java:comp/UserTransaction` in the initial context.

### 4.1.3. Distributed Transaction Management

As explained in the previous section, the transactional behaviour of an application can be defined in a declarative way or coded in the bean and/or the client itself (transaction boundaries demarcation). In any case, the distribution aspects of the transactions are completely transparent to the bean provider and to the application assembler. This means that a transaction may involve beans located on several JOnAS servers and that the platform itself will handle management of the global transaction. It will perform the two-phase commit protocol between the different servers, and the bean programmer need do nothing.

Once the beans have been developed and the application has been assembled, it is possible for the deployer and for the administrator to configure the distribution of the different beans on one or several machines, and within one or several JOnAS servers. This can be done without impacting either the beans code or their deployment descriptors. The distributed configuration is specified at launch time. In the environment properties of an EJB server, the following can be specified:

- which enterprise beans the JOnAS server will handle,
- if a Java Transaction Monitor will be located in the same Java Virtual Machine (JVM) or not.

To achieve this goal, two properties must be set in the jonas.properties file, jonas.service.ejb.descriptors and jonas.service.jtm.remote . The first one lists the beans that will be handled on this server (by specifying the name of their ejb-jar files), and the second one sets the Java Transaction Monitor (JTM) launching mode:

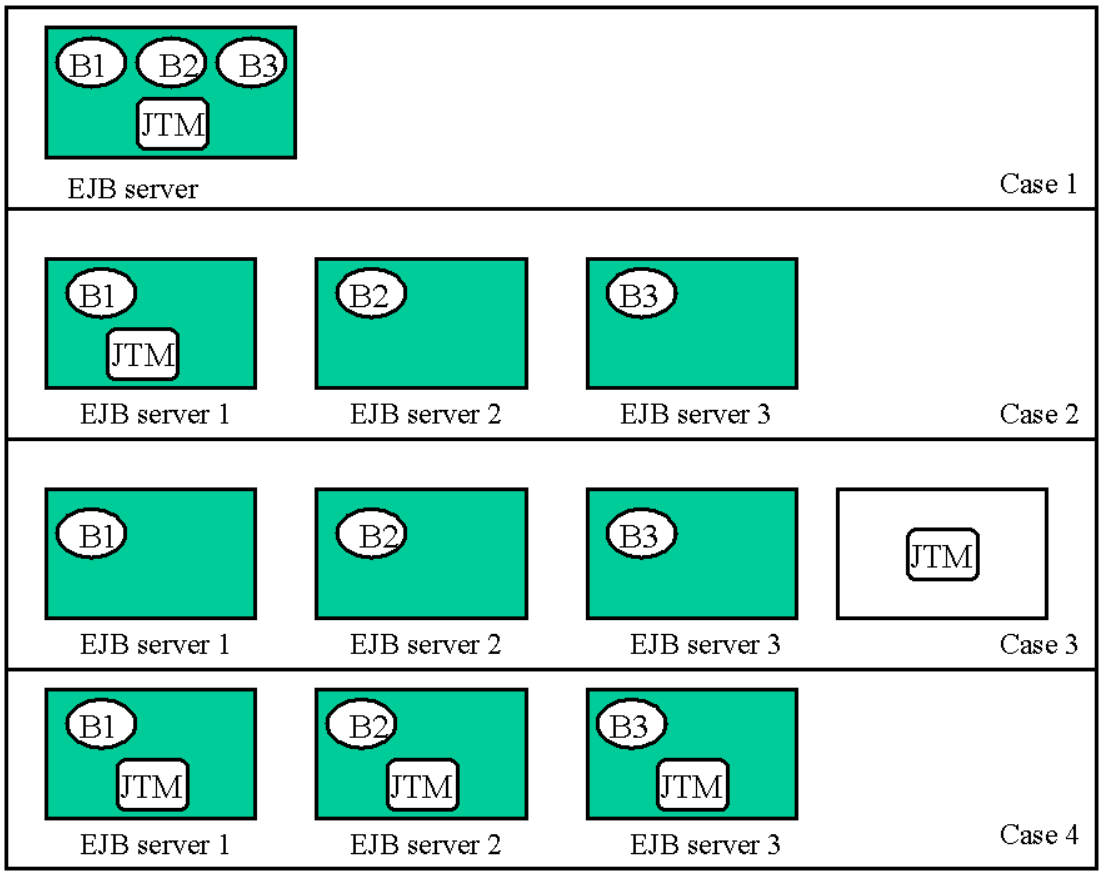
- if set to `true` , the JTM is remote, i.e. the JTM must be launched previously in another JVM,
- if set to `false` , the JTM is local, i.e. it will run in the same JVM as the EJB Server.

Example:

```
jonas.service.ejb.descriptors       Bean1.jar, Bean2.jar
jonas.service.jtm.remote            false
```

Using these configuration facilities, it is possible to adapt the beans distribution to the resources (cpu and data) location, for optimizing performance.

The following figure illustrates four cases of distribution configuration for three beans.




1. Case 1: The three beans B1, B2, and B3 are located on the same JOnAS server, which embeds a Java Transaction Monitor.

2. Case 2: The three beans are located on different JOnAS servers, one of them running the Java Transaction Monitor, which manages the global transaction.

3. Case 3: The three beans are located on different JOnAS servers, the Java Transaction Monitor is running outside of any JOnAS server.

4. Case 4: The three beans are located on different JOnAS servers. Each server is running a Java Transaction Monitor. One of the JTM acts as the master monitor, while the two others are slaves.

These different configuration cases may be obtained by launching the JOnAS servers and eventually the JTM (case 3) with the adequate properties. The rational when choosing one of these configurations is resources location and load balancing. However, consider the following pointers:

- if the beans should run on the same machine, with the same server configuration, case 1 is the more appropriate;

- if the beans should run on different machines, case 4 is the more appropriate, since it favours local transaction management;

- if the beans should run on the same machine, but require different server configurations, case 2 is a good approach.

# 4.2. EJB2 Environment

## 4.2.1. Introduction

The enterprise bean environment is a mechanism that allows customization of the enterprise bean's business logic during assembly or deployment. The environment is a way for a bean to refer to a value, to a resource, or to another component so that the code will be independent of the actual referred object. The actual value of such environment references (or variables) is set at deployment time, according to what is contained in the deployment descriptor. The enterprise bean's environment allows the enterprise bean to be customized without the need to access or change the enterprise bean's source code.

The enterprise bean environment is provided by the container (i.e. the JOnAS server) to the bean through the JNDI interface as a JNDI context. The bean code accesses the environment using JNDI with names starting with "java:comp/env/".

## 4.2.2. Environment Entries

The bean provider declares all the bean environment entries in the deployment descriptor via the env-entry element. The deployer can set or modify the values of the environment entries.

A bean accesses its environment entries with a code similar to the following:

```
InitialContext ictx = new InitialContext();
Context myenv = ictx.lookup("java:comp/env");
Integer min = (Integer) myenv.lookup("minvalue");
Integer max = (Integer) myenv.lookup("maxvalue");
```

In the standard deployment descriptor, the declaration of these variables are as follows:

```
<env-entry>
  <env-entry-name>minvalue</env-entry-name>
  <env-entry-type>java.lang.Integer</env-entry-type>
  <env-entry-value>12</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>maxvalue</env-entry-name>
  <env-entry-type>java.lang.Integer</env-entry-type>
  <env-entry-value>120</env-entry-value>
</env-entry>
```

## 4.2.3. Resource References

The resource references are another examples of environment entries. For such entries, using subcontexts is recommended:

- java:comp/env/jdbc for references to DataSources objects.
- java:comp/env/jms for references to JMS connection factories.

In the standard deployment descriptor, the declaration of a resource reference to a JDBC connection factory is:

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

And the bean accesses the datasource as in the following:

```
InitialContext ictx = new InitialContext();
DataSource ds = ictx.lookup("java:comp/env/jdbc/AccountExplDs");
```

Binding of the resource references to the actual resource manager connection factories that are configured in the EJB server is done in the JOnAS-specific deployment descriptor using the jonas-resource element.

```
<jonas-resource>
  <res-ref-name>jdbc/AccountExplDs</res-ref-name>
  <jndi-name>jdbc_1</jndi-name>
</jonas-resource>
```

## 4.2.4. Resource Environment References

The resource environment references are another example of environment entries. They allow the Bean Provider to refer to administered objects that are associated with resources (for example, JMS Destinations), by using logical names. Resource environment references are defined in the standard deployment descriptor.

```
<resource-env-ref>
  <resource-env-ref-name>jms/stockQueue</resource-env-ref-name>
  <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
</resource-env-ref>
```

Binding of the resource environment references to administered objects in the target operational environment is done in the JOnAS-specific deployment descriptor using the jonas-resource-env element.

```
<jonas-resource-env>
  <resource-env-ref-name>jms/stockQueue</resource-env-ref-name>
  <jndi-name>myQueue<jndi-name>
</jonas-resource-env>
```

## 4.2.5. EJB References

The EJB reference is another special entry in the enterprise bean's environment. EJB references allow the Bean Provider to refer to the homes of other enterprise beans using logical names. For such entries, using the subcontext java:comp/env/ejb is recommended.

The declaration of an EJB reference used for accessing the bean through its remote home and component interfaces in the standard deployment descriptor is shown in the following example:

```
<ejb-ref>
  <ejb-ref-name>ejb/ses1</ejb-ref-name>
  <ejb-ref-type>session</ejb-ref-type>
  <home>tests.SS1Home</home>
  <remote>tests.SS1</remote>
</ejb-ref>
```

The declaration of an EJB reference used for accessing the bean through its local home and component interfaces in the standard deployment descriptor is shown in the following example:

```
<ejb-local-ref>
```

```
    <ejb-ref-name>ejb/locses1</ejb-ref-name>
    <ejb-ref-type>session</ejb-ref-type>
    <local-home>tests.LocalSS1Home</local-home>
    <local>tests.LocalSS1</local>
    <ejb-link>LocalBean</ejb-link>
  </ejb-local-ref>
```

Local interfaces are available in the same JVM as the bean providing this interface. The use of these interfaces also implies that the classloader of the component performing a lookup (bean or servlet component) is a child of the EJB classloader providing the local interface. Local interfaces, then, are not available to outside WARs or outside EJB-JARs even if they run in the same JVM. This is due to the fact that classes of the local interfaces are not visible on the client side. Putting them under the WEB-INF/lib folder of a WAR would not change anything as the two classes would be loaded by different classloaders, which will throw a "ClassCastException".

To summarize, local interfaces are available only for

- beans in a same ejb jar file.
- from servlets to beans or ejbs to ejbs but in the same ear file.

If the referred bean is defined in the same ejb-jar or EAR file, the optional ejb-link element of the ejb-ref element can be used to specify the actual referred bean. The value of the ejb-link element is the name of the target enterprise bean, i.e. the name defined in the ejb-name element of the target enterprise bean. If the target enterprise bean is in the same EAR file, but in a different ejb-jar file, the name of the ejb-link element may be the name of the target bean, prefixed by the name of the containing ejb-jar file followed by '#' (e.g. "My_EJBs.jar#bean1"); prefixing by the name of the ejb-jar file is necessary only if some ejb-name conflicts occur, otherwise the name of the target bean is enough. In the following example, the ejb-link element has been added to the ejb-ref (in the referring bean SSA) and a part of the description of the target bean (SS1) is shown:

```
<session>
  <ejb-name>SSA</ejb-name>
  ...
  <ejb-ref>
    <ejb-ref-name>ejb/ses1</ejb-ref-name>
    <ejb-ref-type>session</ejb-ref-type>
    <home>tests.SS1Home</home>
    <remote>tests.SS1</remote>
    <ejb-link>SS1</ejb-link>
  </ejb-ref>
...
</session>
...
<session>
  <ejb-name>SS1</ejb-name>
  <home>tests.SS1Home</home>
  <local-home>tests.LocalSS1Home</local-home>
  <remote>tests.SS1</remote>
  <local>tests.LocalSS1</local>
  <ejb-class>tests.SS1Bean</ejb-class>
  ...
</session>
 ...
```

If the bean SS1 was not in the same ejb-jar file as SSA, but in another file named product_ejbs.jar, the ejb-link element could have been:

```
    <ejb-link>product_ejbs.jar#SS1</ejb-link>
```

If the referring component and the referred bean are in separate files and not in the same EAR, the current JOnAS implementation does not allow use of the ejb-link element. In this case, to resolve the reference, the jonas-ejb-ref element in the JOnAS-specific deployment descriptor would be used to bind the environment JNDI name of the EJB reference to the actual JNDI name of the associated

enterprise bean home. In the following example, it is assumed that the JNDI name of the SS1 bean home is SS1Home_one.

```
    <jonas-session>
      <ejb-name>SSA</ejb-name>
      <jndi-name>SSAHome</jndi-name>
      <jonas-ejb-ref>
        <ejb-ref-name>ejb/ses1</ejb-ref-name>
        <jndi-name>SS1Home_one</jndi-name>
      </jonas-ejb-ref>
    </jonas-session>
    ...
    <jonas-session>
      <ejb-name>SS1</ejb-name>
      <jndi-name>SS1Home_one</jndi-name>
      <jndi-local-name>SS1LocalHome_one</jndi-local-name>
    </jonas-session>
    ...
```

The bean locates the home interface of the other enterprise bean using the EJB reference with the following code:

```
    InitialContext ictx = new InitialContext();
    Context myenv = ictx.lookup("java:comp/env");
    SS1Home home = (SS1Home)javax.rmi.PortableRemoteObject.narrow(myEnv.lookup("ejb/ses1"),
SS1Home.class);
```

# 4.3. Security Management

## 4.3.1. Introduction

The EJB architecture encourages the Bean programmer to implement the enterprise bean class without hard-coding the security policies and mechanisms into the business methods.

## 4.3.2. Declarative Security Management

The application assembler can define a security view of the enterprise beans contained in the ejb-jar file. The security view consists of a set of security roles . A security role is a semantic grouping of permissions for a given type of application user that allows that user to successfully use the application. The application assembler can define (declaratively in the deployment descriptor) method permissions for each security role. A method permission is a permission to invoke a specified group of methods for the enterprise beans' home and remote interfaces. The security roles defined by the application assembler present this simplified security view of the enterprise beans application to the deployer; the deployer's view of security requirements for the application is the small set of security roles, rather than a large number of individual methods.

### 4.3.2.1. Security roles

The application assembler can define one or more security roles in the deployment descriptor. The application assembler then assigns groups of methods of the enterprise beans' home and remote interfaces to the security roles in order to define the security view of the application.

The scope of the security roles defined in the `security-role` elements is the ejb-jar file level, and this includes all the enterprise beans in the ejb-jar file.

```
  ...
  <assembly-descriptor>
     <security-role>
        <role-name>tomcat</role-name>
     </security-role>
     ...
```

```
   </assembly-descriptor>
```

### 4.3.2.2. Method permissions

After defining security roles for the enterprise beans in the ejb-jar file, the application assembler can also specify the methods of the remote and home interfaces that each security role is allowed to invoke.

Method permissions are defined as a binary relationship in the deployment descriptor from the set of security roles to the set of methods of the home and remote interfaces of the enterprise beans, including all their super interfaces (including the methods of the `javax.ejb.EJBHome` and `javax.ejb.EJBObject` interfaces). The method permissions relationship includes the pair (R, M) only if the security role R is allowed to invoke the method M .

The application assembler defines the method permissions relationship in the deployment descriptor using the `method-permission` element as follows:

- Each `method-permission` element includes a list of one or more security roles and a list of one or more methods. All the listed security roles are allowed to invoke all the listed methods. Each security role in the list is identified by the `role-name` element, and each method is identified by the `method` element.
- The method permissions relationship is defined as the union of all the method permissions defined in the individual `method-permission` elements.
- A security role or a method can appear in multiple `method-permission` elements.

It is possible that some methods are not assigned to any security roles. This means that these methods can be accessed by anyone.

The following example illustrates how security roles are assigned to methods' permissions in the deployment descriptor:

```
   ...
   <method-permission>
      <role-name>tomcat</role-name>
      <method>
         <ejb-name>Op</ejb-name>
         <method-name>*</method-name>
      </method>
   </method-permission>
   ...
```

## 4.3.3. Programmatic Security Management

Because not all security policies can be expressed declaratively, the EJB architecture also provides a simple programmatic interface that the Bean programmer can use to access the security context from the business methods.

The `javax.ejb.EJBContext` interface provides two methods that allow the Bean programmer to access security information about the enterprise bean's caller.

```
public interface javax.ejb.EJBContext {
   ...
   //
   // The following two methods allow the EJB class
   // to access security information
   //
   java.security.Principal getCallerPrincipal() ;
   boolean isCallerInRole (String roleName) ;
   ...
}
```

#### 4.3.3.1. Use of getCallerPrincipal()

The purpose of the `getCallerPrincipal()` method is to allow the enterprise bean methods to obtain the current caller principal's name. The methods might, for example, use the name as a key to access information in a database.

An enterprise bean can invoke the `getCallerPrincipal()` method to obtain a `java.security.Principal` interface representing the current caller. The enterprise bean can then obtain the distinguished name of the caller principal using the `getName()` method of the `java.security.Principal` interface.

#### 4.3.3.2. Use of isCallerInRole(String roleName)

The main purpose of the `isCallerInRole(String roleName)` method is to allow the Bean programmer to code the security checks that cannot be easily defined declaratively in the deployment descriptor using method permissions. Such a check might impose a role-based limit on a request, or it might depend on information stored in the database.

The enterprise bean code uses the `isCallerInRole(String roleName)` method to test whether the current caller has been assigned to a given security role or not. Security roles are defined by the application assembler in the deployment descriptor and are assigned to principals by the deployer.

#### 4.3.3.3. Declaration of security roles referenced from the bean's code

The Bean programmer must declare in the `security-role-ref` elements of the deployment descriptor all the security role names used in the enterprise bean code. Declaring the security roles' references in the code allows the application assembler or deployer to link the names of the security roles used in the code to the actual security roles defined for an assembled application through the `security-role` elements.

```
...
<enterprise-beans>
   ...
   <session>
      <ejb-nameOp</ejb-name>
      <ejb-class>sb.OpBean</ejb-class>
      ...
      <security-role-ref>
         <role-name>role1</role-name>
      </security-role-ref>
      ...
   </session>
   ...
</enterprise-beans>
...
```

The deployment descriptor in this example indicates that the enterprise bean `Op` makes the security checks using `isCallerInRole("role1")` in at least one of its business methods.

#### 4.3.3.4. Linking security role references and security roles

If the `security-role` elements have been defined in the deployment descriptor, all the security role references declared in the `security-role-ref` elements must be linked to the security roles defined in the `security-role` elements.

The following deployment descriptor example shows how to link the security role references named `role1` to the security role named `tomcat` .

```
...
<enterprise-beans>
   ...
```

```
      <session>
         <ejb-name>Op</ejb-name>
         <ejb-class>sb.OpBean</ejb-class>
         ...
         <security-role-ref>
            <role-name>role1</role-name>
            <role-link>tomcat</role-link>
         </security-role-ref>
         ...
      </session>
      ...
   </enterprise-beans>
   ...
```

In summary, the role names used in the EJB code (in the isCallerInRole method) are, in fact, references to actual security roles, which makes the EJB code independent of the security configuration described in the deployment descriptor. The programmer makes these role references available to the Bean deployer or application assembler via the `security-role-ref` elements included in the `session` or `entity` elements of the deployment descriptor. Then, the Bean deployer or application assembler must map the security roles defined in the deployment descriptor to the "specific" roles of the target operational environment (e.g. groups on Unix systems). However, this last mapping step is not currently available in JOnAS.

# 4.4. Defining the EJB2 Deployment Descriptor

The target audience for this section is the Enterprise Bean provider, i.e. the person in charge of developing the software components on the server side. It describes how the bean provider should build the deployment descriptors of its components. Refer to Section A.1, “xml Tips” for advices about writing xml files.

## 4.4.1. Principles

The bean programmer is responsible for providing the deployment descriptor associated with the developed Enterprise Beans. The Bean Provider's responsibilities and the Application Assembler's responsibilities is to provide an XML deployment descriptor that conforms to the deployment descriptor's XML schema as defined in the EBJ specification version 2.1. (Refer to `$JONAS_ROOT/xml/ejb-jar_2_1.xsd` or http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd [2] ).

To deploy Enterprise JavaBeans on the EJB server, information not defined in the standard XML deployment descriptor may be needed. For example, this information may include the mapping of the bean to the underlying database for an entity bean with container-managed persistence. This information is specified during the deployment step in another XML deployment descriptor that is specific to JOnAS. The JOnAS-specific deployment descriptor's XML schema is located in `$JONAS_ROOT/xml/jonas-ejb-jar_X_Y.xsd` . The file name of the JOnAS-specific XML deployment descriptor must be the file name of the standard XML deployment descriptor prefixed by `' jonas- '`.

The parser gets the specified schema via the classpath (schemas are packaged in the $JONAS_ROOT/lib/common/ow_jonas.jar file).

The standard deployment descriptor should contain structural information for each enterprise bean that includes the following:

- the Enterprise bean's name,
- the Enterprise bean's class,
- the Enterprise bean's home interface,

[2] http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd

- the Enterprise bean's remote interface,
- the Enterprise bean's type,
- a re-entrancy indication for the entity bean,
- the session bean's state management type,
- the session bean's transaction demarcation type,
- the entity bean's persistence management,
- the entity bean's primary key class,
- container-managed fields,
- environment entries,
- the bean's EJB references,
- resource manager connection factory references,
- transaction attributes.

The JOnAS-specific deployment descriptor contains information for each enterprise bean including:

- the JNDI name of the Home object that implement the Home interface of the enterprise bean,
- the JNDI name of the DataSource object corresponding to the resource manager connection factory referenced in the enterprise bean's class,
- the JNDI name of each EJB references,
- the JNDI name of JMS administered objects,
- information for the mapping of the bean to the underlying database, if it is an entity with container-managed persistence.

## 4.4.2. Example of Session Descriptors

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<ejb-jar xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
   http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd"
   version="2.1">

  <description>secured session bean JOnAS example</description>

  <display-name>secusb (earsample)</display-name>

  <enterprise-beans>
    <session>
      <ejb-name>EarOp</ejb-name>
      <home>org.ow2.jonas.earsample.beans.secusb.OpHome</home>
      <remote>org.ow2.jonas.earsample.beans.secusb.Op</remote>
      <local-home>org.ow2.jonas.earsample.beans.secusb.OpLocalHome</local-home>
      <local>org.ow2.jonas.earsample.beans.secusb.OpLocal</local>
      <ejb-class>org.ow2.jonas.earsample.beans.secusb.OpBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>

  <assembly-descriptor>
    <security-role>
      <role-name>tomcat</role-name>
    </security-role>
    <security-role>
      <role-name>jetty</role-name>
```

```
    </security-role>

    <method-permission>
      <role-name>tomcat</role-name>
      <method>
     <ejb-name>EarOp</ejb-name>
     <method-name>*</method-name>
      </method>
    </method-permission>

    <method-permission>
      <role-name>jetty</role-name>
      <method>
     <ejb-name>EarOp</ejb-name>
     <method-name>*</method-name>
      </method>
    </method-permission>

    <container-transaction>
      <method>
 <ejb-name>EarOp</ejb-name>
 <method-name>*</method-name>
      </method>
      <trans-attribute>Supports</trans-attribute>
    </container-transaction>

  </assembly-descriptor>

</ejb-jar>
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>


<jonas-ejb-jar xmlns="http://www.objectweb.org/jonas/ns"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.objectweb.org/jonas/ns
    http://www.objectweb.org/jonas/ns/jonas-ejb-jar_4_0.xsd" >
  <jonas-session>
    <ejb-name>EarOp</ejb-name>
    <jndi-name>EarOpHome</jndi-name>
  </jonas-session>
</jonas-ejb-jar>
```

## 4.4.3. Example of Container-managed Persistence Entity Descriptors (CMP 2.x)

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

<ejb-jar xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
   http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd"
   version="2.1">

  <description>Deployment descriptor for the cmp2 JOnAS example</description>
  <display-name>cmp2 example</display-name>
  <enterprise-beans>

 <entity>
      <ejb-name>CustomerEJB</ejb-name>
      <home>com.titan.customer.CustomerHomeRemote</home>
      <remote>com.titan.customer.CustomerRemote</remote>
      <local-home>com.titan.customer.CustomerHomeLocal</local-home>
      <local>com.titan.customer.CustomerLocal</local>
      <ejb-class>com.titan.customer.CustomerBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.Integer</prim-key-class>
      <reentrant>false</reentrant>
      <cmp-version>2.x</cmp-version>
      <abstract-schema-name>JE2_Customer</abstract-schema-name>
   <cmp-field><field-name>id</field-name></cmp-field>
      <cmp-field><field-name>lastName</field-name></cmp-field>
      <cmp-field><field-name>firstName</field-name></cmp-field>
      <cmp-field><field-name>hasGoodCredit</field-name></cmp-field>
      <primkey-field>id</primkey-field>
      <ejb-local-ref>
  <ejb-ref-name>ejb/AddressHomeLocal</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
```

```
<local-home>com.titan.address.AddressHomeLocal</local-home>
<local>com.titan.address.AddressLocal</local>
<ejb-link>AddressEJB</ejb-link>
    </ejb-local-ref>
    <ejb-local-ref>
<ejb-ref-name>ejb/CreditCardHome</ejb-ref-name>
<ejb-ref-type>Entity</ejb-ref-type>
      <local-home>com.titan.customer.CreditCardHomeLocal</local-home>
      <local>com.titan.customer.CreditCardLocal</local>
      <ejb-link>CreditCardEJB</ejb-link>
    </ejb-local-ref>
    <ejb-local-ref>
      <ejb-ref-name>ejb/PhoneHomeLocal</ejb-ref-name>
      <ejb-ref-type>Entity</ejb-ref-type>
      <local-home>com.titan.phone.PhoneHomeLocal</local-home>
      <local>com.titan.phone.PhoneLocal</local>
      <ejb-link>PhoneEJB</ejb-link>
    </ejb-local-ref>
    <security-identity><use-caller-identity/></security-identity>
    <query>
      <query-method>
        <method-name>findAllCustomers</method-name>
        <method-params></method-params>
      </query-method>
      <ejb-ql>SELECT OBJECT(c) FROM JE2_Customer AS c</ejb-ql>
    </query>
    <query>
      <query-method>
        <method-name>findByName</method-name>
        <method-params>
          <method-param>java.lang.String</method-param>
          <method-param>java.lang.String</method-param>
        </method-params>
      </query-method>
      <ejb-ql>
      SELECT OBJECT(c) FROM JE2_Customer c
      WHERE c.lastName LIKE ?1 AND c.firstName LIKE ?2
      </ejb-ql>
    </query>
    <query>
      <query-method>
        <method-name>findSmith90</method-name>
        <method-params></method-params>
      </query-method>
      <ejb-ql>
      SELECT OBJECT(c) FROM JE2_Customer c
      WHERE c.lastName = 'Smith90'
      </ejb-ql>
    </query>
    <query>
      <query-method>
        <method-name>findByExactName</method-name>
        <method-params>
          <method-param>java.lang.String</method-param>
          <method-param>java.lang.String</method-param>
        </method-params>
      </query-method>
      <ejb-ql>
      SELECT OBJECT(c) FROM JE2_Customer c
      WHERE c.lastName = ?1 AND c.firstName = ?2
      </ejb-ql>
    </query>
    <query>
      <query-method>
        <method-name>findByNameAndState</method-name>
        <method-params>
          <method-param>java.lang.String</method-param>
          <method-param>java.lang.String</method-param>
          <method-param>java.lang.String</method-param>
        </method-params>
      </query-method>
      <ejb-ql>
      SELECT OBJECT(c) FROM JE2_Customer c
      WHERE c.lastName LIKE ?1 AND c.firstName LIKE ?2 AND c.homeAddress.state = ?3
      </ejb-ql>
    </query>
    <query>
      <query-method>
        <method-name>findByGoodCredit</method-name>
        <method-params></method-params>
      </query-method>
      <ejb-ql>
      SELECT OBJECT(c) FROM JE2_Customer c
```

```
        WHERE c.hasGoodCredit = TRUE
        </ejb-ql>
      </query>
      <query>
        <query-method>
          <method-name>findByCity</method-name>
          <method-params>
            <method-param>java.lang.String</method-param>
            <method-param>java.lang.String</method-param>
          </method-params>
        </query-method>
        <ejb-ql>
        SELECT OBJECT(c) FROM JE2_Customer c
        WHERE c.homeAddress.city = ?1 AND c.homeAddress.state = ?2
        </ejb-ql>
      </query>
      <query>
        <query-method>
          <method-name>findInHotStates</method-name>
          <method-params></method-params>
        </query-method>
        <ejb-ql>
           SELECT OBJECT(c) FROM JE2_Customer c
           WHERE c.homeAddress.state IN ('FL','TX','AZ','CA')
        </ejb-ql>
      </query>
      <query>
        <query-method>
          <method-name>findWithNoReservations</method-name>
          <method-params></method-params>
        </query-method>
        <ejb-ql>
            SELECT OBJECT(c) FROM JE2_Customer c
            WHERE c.reservations IS EMPTY
       </ejb-ql>
      </query>
      <query>
        <query-method>
          <method-name>findOnCruise</method-name>
          <method-params>
            <method-param>com.titan.cruise.CruiseLocal</method-param>
          </method-params>
        </query-method>
        <ejb-ql>
         SELECT OBJECT(cust) FROM JE2_Customer cust, JE2_Cruise cr, IN (cr.reservations) AS
res
         WHERE cr = ?1 AND cust MEMBER OF res.customers
       </ejb-ql>
      </query>
      <query>
        <query-method>
          <method-name>findByState</method-name>
          <method-params>
            <method-param>java.lang.String</method-param>
          </method-params>
        </query-method>
        <ejb-ql>
         SELECT OBJECT(c) FROM JE2_Customer c
         WHERE c.homeAddress.state = ?1
         ORDER BY c.lastName,c.firstName
       </ejb-ql>
      </query>
    </entity>

    <entity>
      <ejb-name>AddressEJB</ejb-name>
      <local-home>com.titan.address.AddressHomeLocal</local-home>
      <local>com.titan.address.AddressLocal</local>
      <ejb-class>com.titan.address.AddressBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.Object</prim-key-class>
      <reentrant>false</reentrant>
      <cmp-version>2.x</cmp-version>
      <abstract-schema-name>JE2_Address</abstract-schema-name>
      <cmp-field><field-name>street</field-name></cmp-field>
      <cmp-field><field-name>city</field-name></cmp-field>
      <cmp-field><field-name>state</field-name></cmp-field>
      <cmp-field><field-name>zip</field-name></cmp-field>
      <security-identity><use-caller-identity/></security-identity>
      <query>
        <query-method>
          <method-name>findAllAddress</method-name>
          <method-params></method-params>
```

```
      </query-method>
      <ejb-ql>SELECT OBJECT(c) FROM JE2_Address c</ejb-ql>
    </query>
    <query>
      <query-method>
        <method-name>ejbSelectZipCodes</method-name>
        <method-params>
          <method-param>java.lang.String</method-param>
        </method-params>
      </query-method>
      <ejb-ql>SELECT a.zip FROM JE2_Address AS a  WHERE a.state = ?1</ejb-ql>
    </query>
    <query>
      <query-method>
        <method-name>ejbSelectAll</method-name>
        <method-params></method-params>
      </query-method>
      <ejb-ql>SELECT OBJECT(a) FROM JE2_Address AS a</ejb-ql>
    </query>
    <query>
      <query-method>
        <method-name>ejbSelectCustomer</method-name>
        <method-params>
          <method-param>com.titan.address.AddressLocal</method-param>
        </method-params>
      </query-method>
      <ejb-ql>SELECT OBJECT(c) FROM JE2_Customer AS c WHERE c.homeAddress = ?1</ejb-ql>
    </query>
  </entity>

  <entity>
    <ejb-name>PhoneEJB</ejb-name>
    <local-home>com.titan.phone.PhoneHomeLocal</local-home>
    <local>com.titan.phone.PhoneLocal</local>
    <ejb-class>com.titan.phone.PhoneBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.Object</prim-key-class>
    <reentrant>false</reentrant>
    <cmp-version>2.x</cmp-version>
    <abstract-schema-name>JE2_Phone</abstract-schema-name>
    <cmp-field><field-name>number</field-name></cmp-field>
    <cmp-field><field-name>type</field-name></cmp-field>
    <security-identity><use-caller-identity/></security-identity>
    <query>
      <query-method>
        <method-name>findAllPhones</method-name>
        <method-params></method-params>
      </query-method>
      <ejb-ql>SELECT OBJECT(c) FROM JE2_Phone c</ejb-ql>
    </query>
  </entity>

  <entity>
    <ejb-name>CreditCardEJB</ejb-name>
    <local-home>com.titan.customer.CreditCardHomeLocal</local-home>
    <local>com.titan.customer.CreditCardLocal</local>
    <ejb-class>com.titan.customer.CreditCardBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.Object</prim-key-class>
    <reentrant>false</reentrant>
    <cmp-version>2.x</cmp-version>
    <abstract-schema-name>JE2_CreditCard</abstract-schema-name>
    <cmp-field><field-name>expirationDate</field-name></cmp-field>
    <cmp-field><field-name>number</field-name></cmp-field>
    <cmp-field><field-name>nameOnCard</field-name></cmp-field>
    <cmp-field><field-name>creditOrganization</field-name></cmp-field>
    <security-identity><use-caller-identity/></security-identity>
    <query>
      <query-method>
        <method-name>findAllCreditCards</method-name>
        <method-params></method-params>
      </query-method>
      <ejb-ql>SELECT OBJECT(cc) FROM JE2_CreditCard cc</ejb-ql>
    </query>
  </entity>

  <entity>
    <ejb-name>CruiseEJB</ejb-name>
    <local-home>com.titan.cruise.CruiseHomeLocal</local-home>
    <local>com.titan.cruise.CruiseLocal</local>
    <ejb-class>com.titan.cruise.CruiseBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.Object</prim-key-class>
```

```
    <reentrant>false</reentrant>
    <cmp-version>2.x</cmp-version>
    <abstract-schema-name>JE2_Cruise</abstract-schema-name>
    <cmp-field><field-name>name</field-name></cmp-field>
    <security-identity><use-caller-identity/></security-identity>
    <query>
      <query-method>
        <method-name>findAllCruises</method-name>
        <method-params></method-params>
      </query-method>
      <ejb-ql>SELECT OBJECT(cc) FROM JE2_Cruise cc</ejb-ql>
    </query>
    <query>
      <query-method>
        <method-name>findByName</method-name>
        <method-params>
          <method-param>java.lang.String</method-param>
        </method-params>
      </query-method>
      <ejb-ql>SELECT OBJECT(c) FROM JE2_Cruise c WHERE c.name = ?1</ejb-ql>
    </query>
  </entity>

  <entity>
    <ejb-name>ShipEJB</ejb-name>
    <local-home>com.titan.ship.ShipHomeLocal</local-home>
    <local>com.titan.ship.ShipLocal</local>
    <ejb-class>com.titan.ship.ShipBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.Integer</prim-key-class>
    <reentrant>false</reentrant>
    <cmp-version>2.x</cmp-version>
    <abstract-schema-name>JE2_Ship</abstract-schema-name>
    <cmp-field><field-name>id</field-name></cmp-field>
    <cmp-field><field-name>name</field-name></cmp-field>
    <cmp-field><field-name>tonnage</field-name></cmp-field>
    <primkey-field>id</primkey-field>
    <security-identity><use-caller-identity/></security-identity>
    <query>
      <query-method>
        <method-name>findAllShips</method-name>
        <method-params></method-params>
      </query-method>
      <ejb-ql>SELECT OBJECT(cc) FROM JE2_Ship cc</ejb-ql>
    </query>
    <query>
      <query-method>
        <method-name>findByTonnage</method-name>
        <method-params>
          <method-param>java.lang.Double</method-param>
        </method-params>
      </query-method>
      <ejb-ql>
       SELECT OBJECT(s) FROM JE2_Ship s
       WHERE s.tonnage = ?1
      </ejb-ql>
    </query>
    <query>
      <query-method>
        <method-name>findByTonnage</method-name>
        <method-params>
          <method-param>java.lang.Double</method-param>
          <method-param>java.lang.Double</method-param>
        </method-params>
      </query-method>
      <ejb-ql>
      SELECT OBJECT(s) FROM JE2_Ship s
      WHERE s.tonnage BETWEEN ?1 AND ?2
      </ejb-ql>
    </query>
  </entity>

  <entity>
    <ejb-name>ReservationEJB</ejb-name>
    <local-home>com.titan.reservation.ReservationHomeLocal</local-home>
    <local>com.titan.reservation.ReservationLocal</local>
    <ejb-class>com.titan.reservation.ReservationBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.Object</prim-key-class>
    <reentrant>false</reentrant>
    <cmp-version>2.x</cmp-version>
    <abstract-schema-name>JE2_Reservation</abstract-schema-name>
    <cmp-field><field-name>amountPaid</field-name></cmp-field>
```

```
    <cmp-field><field-name>date</field-name></cmp-field>
    <security-identity><use-caller-identity/></security-identity>
    <query>
      <query-method>
        <method-name>findAllReservations</method-name>
        <method-params></method-params>
      </query-method>
      <ejb-ql>SELECT OBJECT(cc) FROM JE2_Reservation cc</ejb-ql>
    </query>
  </entity>

  <entity>
    <ejb-name>CabinEJB</ejb-name>
    <home>com.titan.cabin.CabinHomeRemote</home>
    <remote>com.titan.cabin.CabinRemote</remote>
    <local-home>com.titan.cabin.CabinHomeLocal</local-home>
    <local>com.titan.cabin.CabinLocal</local>
    <ejb-class>com.titan.cabin.CabinBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.Integer</prim-key-class>
    <reentrant>false</reentrant>
    <cmp-version>2.x</cmp-version>
    <abstract-schema-name>JE2_Cabin</abstract-schema-name>
    <cmp-field><field-name>id</field-name></cmp-field>
    <cmp-field><field-name>name</field-name></cmp-field>
    <cmp-field><field-name>deckLevel</field-name></cmp-field>
    <cmp-field><field-name>bedCount</field-name></cmp-field>
    <primkey-field>id</primkey-field>
    <security-identity><use-caller-identity/></security-identity>
    <query>
      <query-method>
        <method-name>findAllOnDeckLevel</method-name>
        <method-params>
          <method-param>java.lang.Integer</method-param>
        </method-params>
      </query-method>
      <ejb-ql>
      SELECT OBJECT(c) FROM JE2_Cabin as c WHERE c.deckLevel = ?1
      </ejb-ql>
    </query>
    <query>
      <query-method>
        <method-name>findAllCabins</method-name>
        <method-params></method-params>
      </query-method>
      <ejb-ql>SELECT OBJECT(cc) FROM JE2_Cabin cc</ejb-ql>
    </query>
    <query>
      <query-method>
        <method-name>ejbSelectAllForCustomer</method-name>
        <method-params>
          <method-param>com.titan.customer.CustomerLocal</method-param>
        </method-params>
      </query-method>
      <ejb-ql>SELECT OBJECT(cab) FROM JE2_Customer AS cust,
              IN (cust.reservations) AS res, IN (res.cabins) AS cab
              WHERE cust = ?1
      </ejb-ql>
    </query>
  </entity>

  <session>
    <ejb-name>TravelAgentEJB</ejb-name>
    <home>com.titan.travelagent.TravelAgentHomeRemote</home>
    <remote>com.titan.travelagent.TravelAgentRemote</remote>
    <ejb-class>com.titan.travelagent.TravelAgentBean</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
    <ejb-local-ref>
      <ejb-ref-name>ejb/CabinHomeLocal</ejb-ref-name>
      <ejb-ref-type>Entity</ejb-ref-type>
      <local-home>com.titan.cabin.CabinHomeLocal</local-home>
      <local>com.titan.cabin.CabinLocal</local>
      <ejb-link>CabinEJB</ejb-link>
    </ejb-local-ref>
    <security-identity><use-caller-identity/></security-identity>
  </session>

  <session>
    <ejb-name>RTravelAgentEJB</ejb-name>
    <home>com.titan.travelagent.RTravelAgentHomeRemote</home>
    <remote>com.titan.travelagent.RTravelAgentRemote</remote>
    <ejb-class>com.titan.travelagent.RTravelAgentBean</ejb-class>
```

```
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <ejb-ref>
        <ejb-ref-name>ejb/CabinHomeRemote</ejb-ref-name>
        <ejb-ref-type>Entity</ejb-ref-type>
        <home>com.titan.cabin.CabinHomeRemote</home>
        <remote>com.titan.cabin.CabinRemote</remote>
      </ejb-ref>
      <security-identity><use-caller-identity/></security-identity>
    </session>

  </enterprise-beans>

  <relationships>
    <ejb-relation>
      <ejb-relation-name>Customer-Address</ejb-relation-name>
      <ejb-relationship-role>
        <ejb-relationship-role-name>Customer-has-a-Address</ejb-relationship-role-name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
          <ejb-name>CustomerEJB</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>homeAddress</cmr-field-name>
        </cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <ejb-relationship-role-name>Address-belongs-to-Customer</ejb-relationship-role-name>
        <multiplicity>One</multiplicity>
  <cascade-delete/>
        <relationship-role-source>
          <ejb-name>AddressEJB</ejb-name>
        </relationship-role-source>
      </ejb-relationship-role>
    </ejb-relation>

    <ejb-relation>
      <ejb-relation-name>Customer-CreditCard</ejb-relation-name>
      <ejb-relationship-role>
        <ejb-relationship-role-name>Customer-has-a-CreditCard</ejb-relationship-role-name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
          <ejb-name>CustomerEJB</ejb-name>
        </relationship-role-source>
        <cmr-field><cmr-field-name>creditCard</cmr-field-name></cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <ejb-relationship-role-name>CreditCard-belongs-to-Customer</ejb-relationship-role-
name>
        <multiplicity>One</multiplicity>
  <cascade-delete/>
        <relationship-role-source>
          <ejb-name>CreditCardEJB</ejb-name>
        </relationship-role-source>
        <cmr-field><cmr-field-name>customer</cmr-field-name></cmr-field>
      </ejb-relationship-role>
    </ejb-relation>

    <ejb-relation>
      <ejb-relation-name>Customer-Phones</ejb-relation-name>
      <ejb-relationship-role>
        <ejb-relationship-role-name>Customer-has-many-Phone-numbers</ejb-relationship-role-
name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
          <ejb-name>CustomerEJB</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>phoneNumbers</cmr-field-name>
          <cmr-field-type>java.util.Collection</cmr-field-type>
        </cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <ejb-relationship-role-name>Phone-belongs-to-Customer</ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
  <cascade-delete/>
        <relationship-role-source>
          <ejb-name>PhoneEJB</ejb-name>
        </relationship-role-source>
      </ejb-relationship-role>
    </ejb-relation>

    <ejb-relation>
```

```
      <ejb-relation-name>Customer-Reservation</ejb-relation-name>
      <ejb-relationship-role>
        <ejb-relationship-role-name>Customer-has-many-Reservations</ejb-relationship-role-
name>
        <multiplicity>Many</multiplicity>
        <relationship-role-source>
          <ejb-name>CustomerEJB</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>reservations</cmr-field-name>
          <cmr-field-type>java.util.Collection</cmr-field-type>
        </cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <ejb-relationship-role-name>Reservation-has-many-Customers</ejb-relationship-role-
name>
        <multiplicity>Many</multiplicity><relationship-role-source>
          <ejb-name>ReservationEJB</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>customers</cmr-field-name>
          <cmr-field-type>java.util.Set</cmr-field-type>
        </cmr-field>
      </ejb-relationship-role>
    </ejb-relation>

    <ejb-relation>
      <ejb-relation-name>Cruise-Ship</ejb-relation-name>
      <ejb-relationship-role>
        <ejb-relationship-role-name>Cruise-has-a-Ship</ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
        <relationship-role-source>
          <ejb-name>CruiseEJB</ejb-name>
        </relationship-role-source>
        <cmr-field><cmr-field-name>ship</cmr-field-name></cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <ejb-relationship-role-name>Ship-has-many-Cruises</ejb-relationship-role-name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
          <ejb-name>ShipEJB</ejb-name>
        </relationship-role-source>
      </ejb-relationship-role>
    </ejb-relation>

    <ejb-relation>
      <ejb-relation-name>Cruise-Reservation</ejb-relation-name>
      <ejb-relationship-role>
        <ejb-relationship-role-name>Cruise-has-many-Reservations</ejb-relationship-role-
name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
          <ejb-name>CruiseEJB</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>reservations</cmr-field-name>
          <cmr-field-type>java.util.Collection</cmr-field-type>
        </cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <ejb-relationship-role-name>Reservation-has-a-Cruise</ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
        <relationship-role-source>
          <ejb-name>ReservationEJB</ejb-name>
        </relationship-role-source>
        <cmr-field><cmr-field-name>cruise</cmr-field-name></cmr-field>
      </ejb-relationship-role>
    </ejb-relation>

    <ejb-relation>
      <ejb-relation-name>Cabin-Ship</ejb-relation-name>
      <ejb-relationship-role>
        <ejb-relationship-role-name>Cabin-has-a-Ship</ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
        <relationship-role-source>
          <ejb-name>CabinEJB</ejb-name>
        </relationship-role-source>
        <cmr-field><cmr-field-name>ship</cmr-field-name></cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <ejb-relationship-role-name>Ship-has-many-Cabins</ejb-relationship-role-name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
```

```
        <ejb-name>ShipEJB</ejb-name>
      </relationship-role-source>
    </ejb-relationship-role>
  </ejb-relation>

  <ejb-relation>
    <ejb-relation-name>Cabin-Reservation</ejb-relation-name>
    <ejb-relationship-role>
      <ejb-relationship-role-name>Cabin-has-many-Reservations</ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>CabinEJB</ejb-name>
      </relationship-role-source>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <ejb-relationship-role-name>Reservation-has-many-Cabins</ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>ReservationEJB</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>cabins</cmr-field-name>
        <cmr-field-type>java.util.Set</cmr-field-type>
      </cmr-field>
    </ejb-relationship-role>
  </ejb-relation>

</relationships>


<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>CabinEJB</ejb-name>
      <method-name>*</method-name>
    </method>
    <method>
      <ejb-name>RTravelAgentEJB</ejb-name>
      <method-name>*</method-name>
    </method>
    <method>
      <ejb-name>TravelAgentEJB</ejb-name>
      <method-name>*</method-name>
    </method>
    <method>
      <ejb-name>CustomerEJB</ejb-name>
      <method-name>*</method-name>
    </method>
    <method>
      <ejb-name>AddressEJB</ejb-name>
      <method-name>*</method-name>
    </method>
    <method>
      <ejb-name>CreditCardEJB</ejb-name>
      <method-name>*</method-name>
    </method>
    <method>
      <ejb-name>PhoneEJB</ejb-name>
      <method-name>*</method-name>
    </method>
    <method>
      <ejb-name>CruiseEJB</ejb-name>
      <method-name>*</method-name>
    </method>
    <method>
      <ejb-name>ShipEJB</ejb-name>
      <method-name>*</method-name>
    </method>
    <method>
      <ejb-name>ReservationEJB</ejb-name>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>

  <container-transaction>
    <method>
      <ejb-name>SequenceSession</ejb-name>
      <method-name>*</method-name>
    </method>
    <method>
      <ejb-name>Sequence</ejb-name>
      <method-name>*</method-name>
```

```
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>

  </assembly-descriptor>

</ejb-jar>
```

```
<?xml version = "1.0" encoding = "UTF-8"?>

<jonas-ejb-jar xmlns="http://www.objectweb.org/jonas/ns"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.objectweb.org/jonas/ns
    http://www.objectweb.org/jonas/ns/jonas-ejb-jar_4_0.xsd" >

   <jonas-entity>
     <ejb-name>CustomerEJB</ejb-name>
     <jndi-name>CustomerHomeRemote</jndi-name>
     <jdbc-mapping><jndi-name>jdbc_1</jndi-name></jdbc-mapping>
   </jonas-entity>

  <jonas-entity>
    <ejb-name>AddressEJB</ejb-name>
    <jdbc-mapping>
        <jndi-name>jdbc_1</jndi-name>
    </jdbc-mapping>
  </jonas-entity>

    <jonas-entity>
      <ejb-name>CabinEJB</ejb-name>
      <jdbc-mapping><jndi-name>jdbc_1</jndi-name></jdbc-mapping>
   </jonas-entity>

   <jonas-entity>
    <ejb-name>PhoneEJB</ejb-name>
    <jdbc-mapping>
       <jndi-name>jdbc_1</jndi-name>
    </jdbc-mapping>
   </jonas-entity>

  <jonas-entity>
    <ejb-name>CreditCardEJB</ejb-name>
    <jdbc-mapping>
       <jndi-name>jdbc_1</jndi-name>
    </jdbc-mapping>
   </jonas-entity>

  <jonas-entity>
    <ejb-name>CruiseEJB</ejb-name>
    <jdbc-mapping>
       <jndi-name>jdbc_1</jndi-name>
    </jdbc-mapping>
   </jonas-entity>

  <jonas-entity>
    <ejb-name>ShipEJB</ejb-name>
    <jdbc-mapping><jndi-name>jdbc_1</jndi-name></jdbc-mapping>
   </jonas-entity>

  <jonas-entity>
    <ejb-name>ReservationEJB</ejb-name>
    <jdbc-mapping>
       <jndi-name>jdbc_1</jndi-name>
    </jdbc-mapping>
   </jonas-entity>

   <jonas-session>
     <ejb-name>RTravelAgentEJB</ejb-name>
     <jndi-name>RTravelAgentHomeRemote</jndi-name>
         <jonas-ejb-ref>
         <ejb-ref-name>ejb/CabinHomeRemote</ejb-ref-name>
         <jndi-name>CabinHomeRemote</jndi-name>
       </jonas-ejb-ref>
   </jonas-session>

</jonas-ejb-jar>
```

# 4.5. EJB2 Packaging

## 4.5.1. Principles

Enterprise Beans are packaged for deployment in a standard Java programming language Archive file, called an ejb-jar file. This file must contain the following:

| | |
|---|---|
| The beans' class files | The class files of the remote and home interfaces, of the beans' implementations, of the beans' primary key classes (if there are any), and of all necessary classes. |
| The beans' deployment descriptor | The ejb-jar file must contain the deployment descriptors, which are made up of:<br>• The standard xml deployment descriptor, in the format defined in the EJB 2.1 specification. Refer to `$JONAS_ROOT/xml/ejb-jar_2_1.xsd` or http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd [3] . This deployment descriptor must be stored with the name META-INF/ejb-jar.xml in the ejb-jar file.<br>• The JOnAS-specific XML deployment descriptor in the format defined in $JONAS_ROOT/xml/jonas-ejb-jar_X_Y.xsd . This JOnAS deployment descriptor must be stored with the name META-INF/jonas-ejb-jar.xml in the ejb-jar file. |

## 4.5.2. Example

Before building the ejb-jar file of the Account entity bean example, the java source files must be compiled to obtain the class files and the two XML deployment descriptors must be written.

Then, the ejb-jar file (OpEB.jar) can be built using the jar command:

```
cd your_bean_class_directory
mkdir META-INF
cp .../eb/*.xml META-INF
jar cvf OpEB.jar sb/*.class META-INF/*.xml
```

# 4.6. ejb2 Service configuration

This service provides containers for EJB2.1 components.

An EJB container can be created from an `ejb-jar` file using one of the following possibilities:

- The `ejb-jar` file has been copied under $JONAS_BASE/deploy
- The `ejb-jar` file is packaged inside an ear file as a component of a Java EE application. The container will be created when the Java EE application will be deployed via the **ear** service.
- EJB containers may be dynamically created from ejb-jar files using the JonasAdmin tool.
- EJB containers may be dynamically created from ejb-jar files using the command jonas admin:

```
jonas admin -a <some-dir>/sb.jar
```

The **ejb** service can (and by default does) provide monitoring options. Monitoring provides the following values at a per-EJB basis for stateful and stateless beans:

- **Number of calls** done on all methods.
- **Total business time**, i.e. the time spent executing business (applicative) code.
- **Total time**, i.e. the total time spent executing code (business code + container code).

The **warningThreshold** option can be used to generate a warning each time a method takes more than **warningThreshold** milliseconds to execute. By default, **warningThreshold** is set to 20 seconds.

Here is the part of `jonas.properties` concerning the **ejb2** service:

```
###################### JOnAS EJB 2 Container service configuration
#
#  Set the name of the implementation class of the ejb2 service
jonas.service.ejb2.class    org.ow2.jonas.ejb2.internal.JOnASEJBService

#  Set the XML deployment descriptors parsing mode (with or without validation)
jonas.service.ejb2.parsingwithvalidation    true

# If enabled, the GenIC tool will be called if :
#  - JOnAS version of the ejb-jar is not the same version than the running JOnAS instance
#  - Stubs/Skels stored in the ejb-jar are not the same than the JOnAS current protocols.
# By default, this is enabled
jonas.service.ejb2.auto-genic    true

# Arguments for the auto GenIC (-invokecmd, -verbose, etc.)
jonas.service.ejb2.auto-genic.args -invokecmd

#  Note: these two settings can be overriden by the EJB descriptor.
#
#  If EJB monitoring is enabled, statistics about method call times will be
#  collected. This is a very lightweight measurement and should not have much
#  impact on performance.
jonas.service.ejb2.monitoringEnabled true
#  If EJB monitoring is enabled, this value indicates after how many
#  milliseconds spent executing an EJB method a warning message should be
#  displayed. If 0, no warning will ever be displayed.
jonas.service.ejb2.warningThreshold 20000
```

For customizing the **ejb2** service it is possible to:

- Set or not the XML validation at the deployment descriptor parsing time
- Set or not the automatic generation via the GenIC tool
- Specify the arguments to pass to the GenIC tool

# Appendix A. Appendix

## A.1. xml Tips

Although some characters, such as ">", are legal, it is good practice to replace them with XML entity references.

The following is a list of the predefined entity references for XML:

| | | |
|---|---|---|
| &lt; | < | less than |
| &gt; | > | greater than |
| &amp; | & | ampersand |
| &apos; | ' | apostrophe |
| &quot; | " | quotation mark |