EJB 2 - Complex Container Managed Relations (CMP) with JBoss

In this tutorial we will show you multiple examples to create CMP relations between EJB. We will use xDoclet to create the interfaces you will need. You will find nearly all combinations of 1:1 1:n and m:n relations. In addition, we provide an overview on one page for all types of relations.

We expect that you have worked through the basic EJB tutorials on our website. We use the MyEclipse plugin which uses xDoclet to generate the interfaces.

General

Author:
Sebastian Hennebrüder

http://www.laliluna.de/tutorials.html
Tutorials for Struts, EJB, xdoclet and eclipse.

Date:
February, 16th 2005

Source code:
The sources does not contain project files or libraries. So create a project first and copy the sources to this projects.

Development Tools
Eclipse 3.x
MyEclipse plugin 3.8
(A cheap and quite powerful Extension to Eclipse to develop Web Applications and EJB (J2EE) Applications. I think that there is a test version availalable at MyEclipse.)
Database
PostgreSQL 8.0 or MySQL
Application Server
Jboss 3.2.5

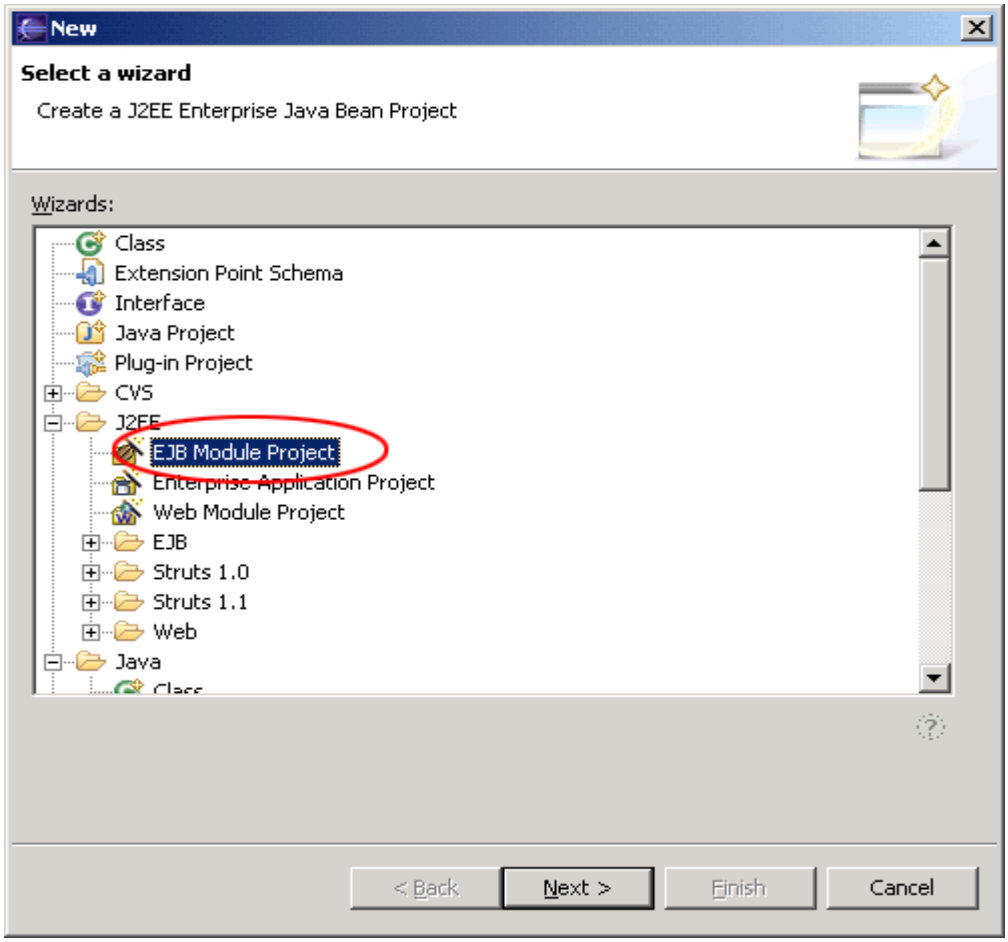
Downloads
PDF:
http://www.laliluna.de/download/complex-ejb-cmp-relation-tutorial.pdf
http://www.laliluna.de/download/complex-ejb-cmp-relation-overview.pdf

Sources: http://www.laliluna.de/download/complex-cmp-relations.zip

Table of Content

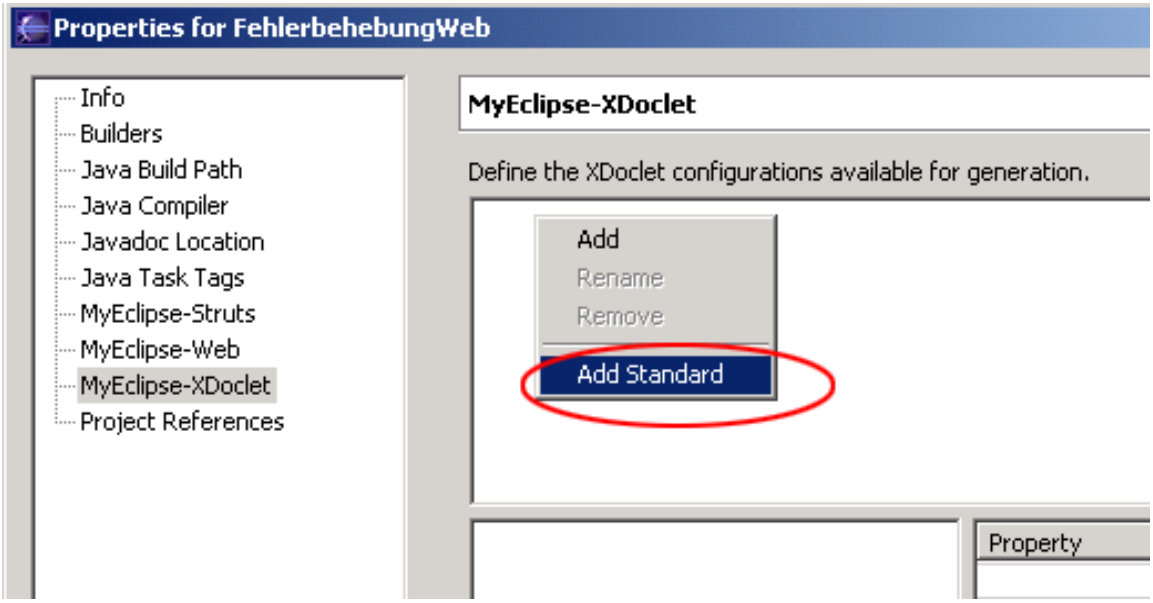
Create an EJB Module Projects

Create a new EJB Module Project. Right click on the package explorer or with shortcut „Strg + n“.

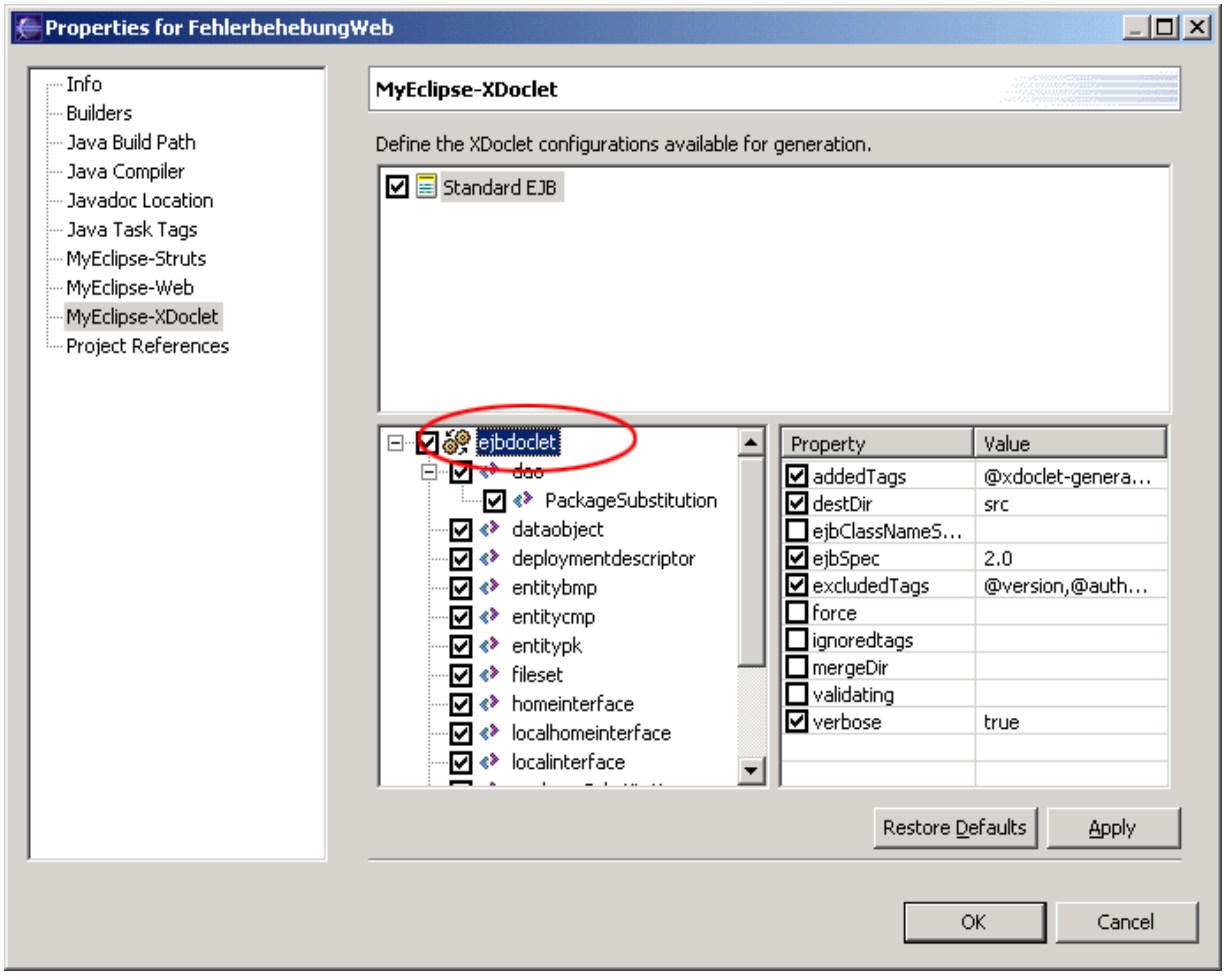


Add xDoclet functionality

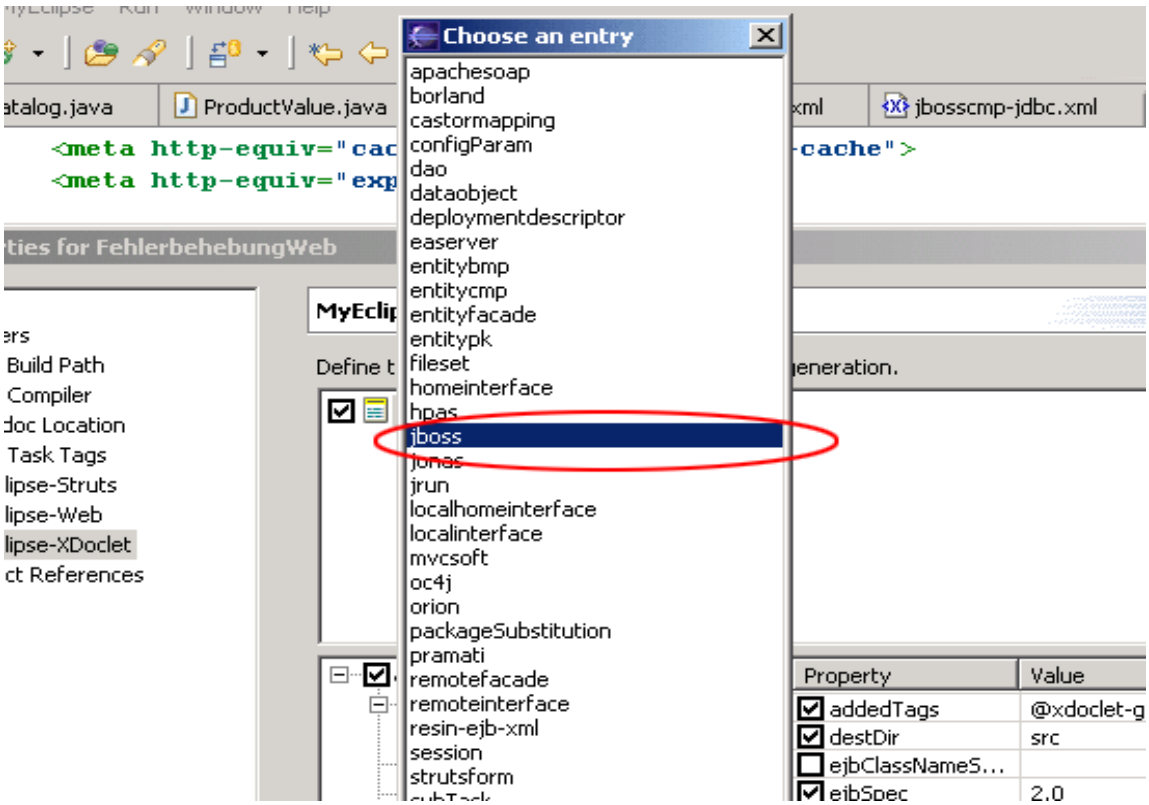
- Open the project properties
- Choose "MyEclipse-xDoclet"
- Right click in the right upper window and choose „Add Standard“.
- Selet Standard EJB and OK



Click on "Standard EJB" in the right upper window.

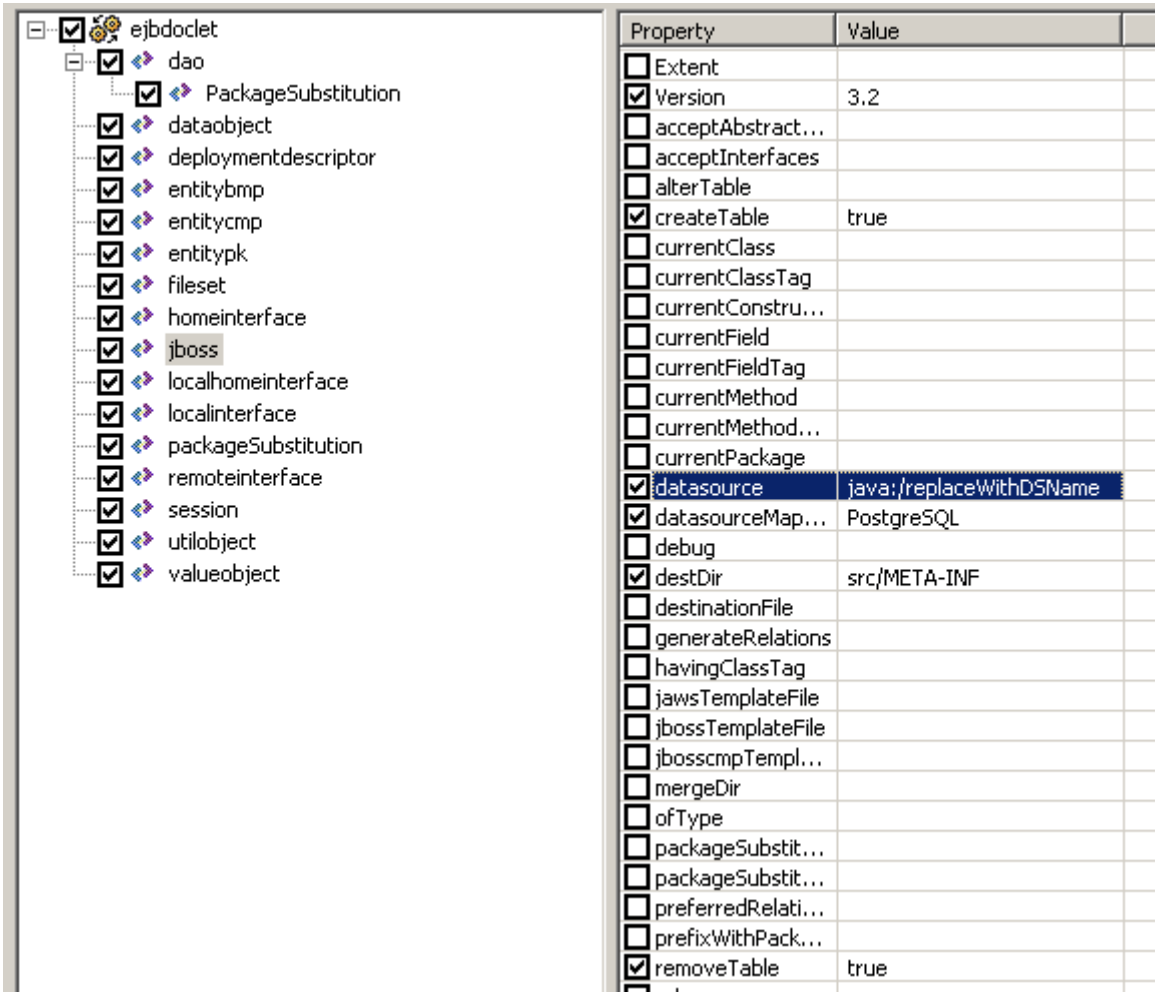


Right click on ejbdoclet and choose jboss from the list.



The following settings must be add.

- Jboss Version, it is 3.2 for all 3.2.x
- the destDir (where the the jboss.xml and jbosscmp-jdbc.xml will create)
- You need a datasource, we will prepare this later. Take a name here like "java:/tutorialDS"
- Datasource Mapping. Tells the Application Server what kind of field is used in the DB for a jdbc-type. Find out the name yourself by looking into {jboss_home}\server\default\conf\standardjbosscmp-jdbc.xml.
  For mySQL it is mySQL,
  For PostgreSQL it is PostgreSQL



Create Datasource Mapping

- Copy the driver to (you will find it on http://www.postgresql.org) to
- \jboss-3.2.4\server\default\lib
- Create the database in Postgre.
- You can find examples configuration files for all supported DBs in
- \jboss-3.2.4\docs\examples\jca\
- Copy the file postgres-ds.xml to
  \jboss-3.2.4\server\default\deploy

change the content of the file to:

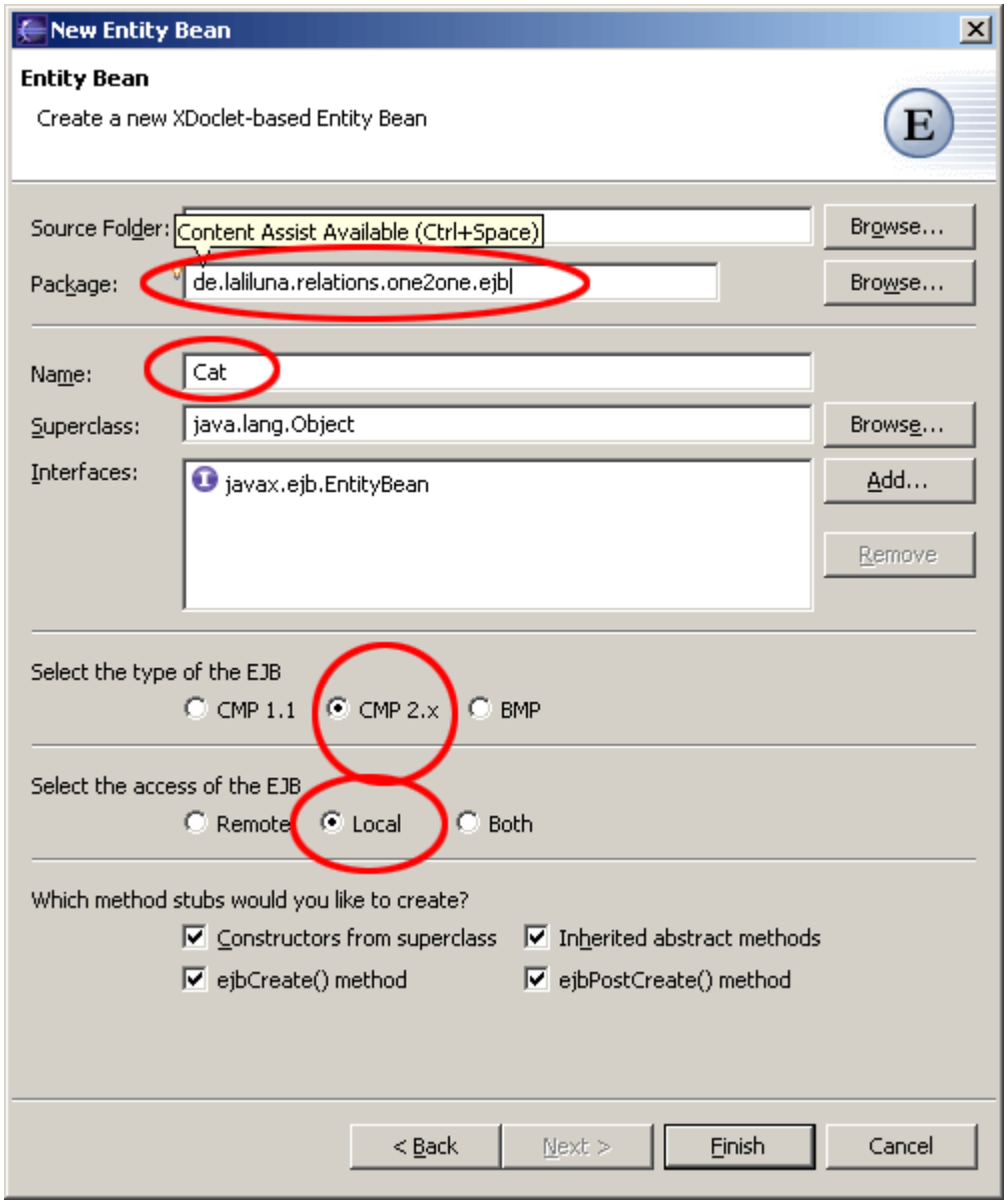
```
<datasources>
  <local-tx-datasource>
    <jndi-name>ReplaceWithDSName</jndi-name>
    <connection-url>
    jdbc:postgresql://localhost:5432/replaceWithDBName
    </connection-url>
    <driver-class>org.postgresql.Driver</driver-class>
    <user-name>username</user-name>
    <password>password</password>
  </local-tx-datasource>
</datasources>
```

Now we are prepared to start our first CMP beans.

1:1 with access from both sides (bidirectional)

Create the two entity beans

Press „Strg + n“ and choose Entity EJB to create a new one. Choose the settings as below. We will use local interfaces in all entity bean as we access them only with a session bean.



Add the following xDoclet tags in front of the class.

```
/**
* @ejb.bean name="Cat"
*           display-name="Name for Cat"
*           description="Description for Cat"
*           jndi-name="ejb/Cat"
*           type="CMP"
*           cmp-version="2.x"
*           view-type="local"
*      primkey-field = "id"
* @ejb.persistence table-name = "tcat"
* @jboss.persistence create-table = "true" remove-table = "true"
*
```

```
 */
public abstract class Cat implements EntityBean
```

Create the following abstract access methods for the field id (our primary key) and the field name

```
 /**
   * @ejb.interface-method view-type = "local"
   * @ejb.persistence column-name = "fid"
   * @ejb.pk-field
   * @return
   */
  public abstract String getId();

  /**
   * @ejb.interface-method view-type = "local"
   * @param id
   */
  public abstract void setId(String id);

  /**
   * @ejb.interface-method view-type = "local"
   * @ejb.persistence column-name = "fname"
   * @return
   */
  public abstract String getName();

  /**
   * @ejb.interface-method view-type = "local"
   * @param name
   */
  public abstract void setName(String name);
```

Create a second entity EJB and name it House. We assume that there is only one cat in one house, which might be wrong as there are people with more than one cat, but in our fictive country the houses are so small and the cats are so big that there can be only one cat in one house.
It is important that you understand the logic. ;-)

Add the following settings to your house bean.

```
/**
 * @ejb.bean name="House"
 *           display-name="Name for House"
 *           description="Description for House"
 *           jndi-name="ejb/House"
 *           type="CMP"
 *           cmp-version="2.x"
 *           view-type="local"
 *                                  primkey-field = "id"
 * @ejb.persistence table-name = "thouse"
 * @jboss.persistence create-table = "true" remove-table = "true"
 */
public abstract class House implements EntityBean
```

Then create the following methods to access the primary key and the field color.

```
/**
   * @ejb.interface-method view-type = "local"
   * @ejb.persistence column-name = "fid"
   * @ejb.pk-field
   * @return
   */
```

```
  public abstract String getId();

  /**
   * @ejb.interface-method view-type = "local"
   * @param id
   */
  public abstract void setId(String id);

  /**
   * @ejb.interface-method view-type = "local"
   * @ejb.persistence column-name = "fcolor"
   * @return
   */
  public abstract String getColor();

  /**
   * @ejb.interface-method view-type = "local"
   * @param color
   */
  public abstract void setColor(String color);
```

Run xDoclet a first time, to create the interfaces and the Util classes.

Then change your ejbCreate method in Cat to the following

```
  public String ejbCreate() throws CreateException
  {
    // [lalilуna] create a random ID
    this.setId(CatUtil.generateGUID(this.getClass()));
    return null;
  }
```

and in the house class as below.

```
  public String ejbCreate() throws CreateException
  {
    // [lalilуna] set a random id
    this.setId(HouseUtil.generateGUID(this.getClass()));
    return null;
  }
```

Create the relation

Add the following to the abstract cat class.

```
  /**
   * @ejb.interface-method view-type = "local"
   * @ejb.relation name = "cat-house" role-name = "cat-lives-in-house"
   * @jboss.relation related-pk-field = "id" fk-column = "cat_fk" fk-constraint
= "true"
   *
   * @return
   */

  public abstract HouseLocal getHouse();

  /**
   * @ejb.interface-method view-type = "local"
   *
   * @param targetEJB
   */
  public abstract void setHouse(HouseLocal houseLocal);
```

And the next snippet to the House class

```
/**
 * @ejb.interface-method view-type = "local"
 * @ejb.relation name = "cat-house" role-name = "house of the cat"
 *
 * @return
 */

public abstract CatLocal getCat();

/**
 * @ejb.interface-method view-type = "local"
 *
 * @param targetEJB
 */
public abstract void setCat(CatLocal catLocal);
```

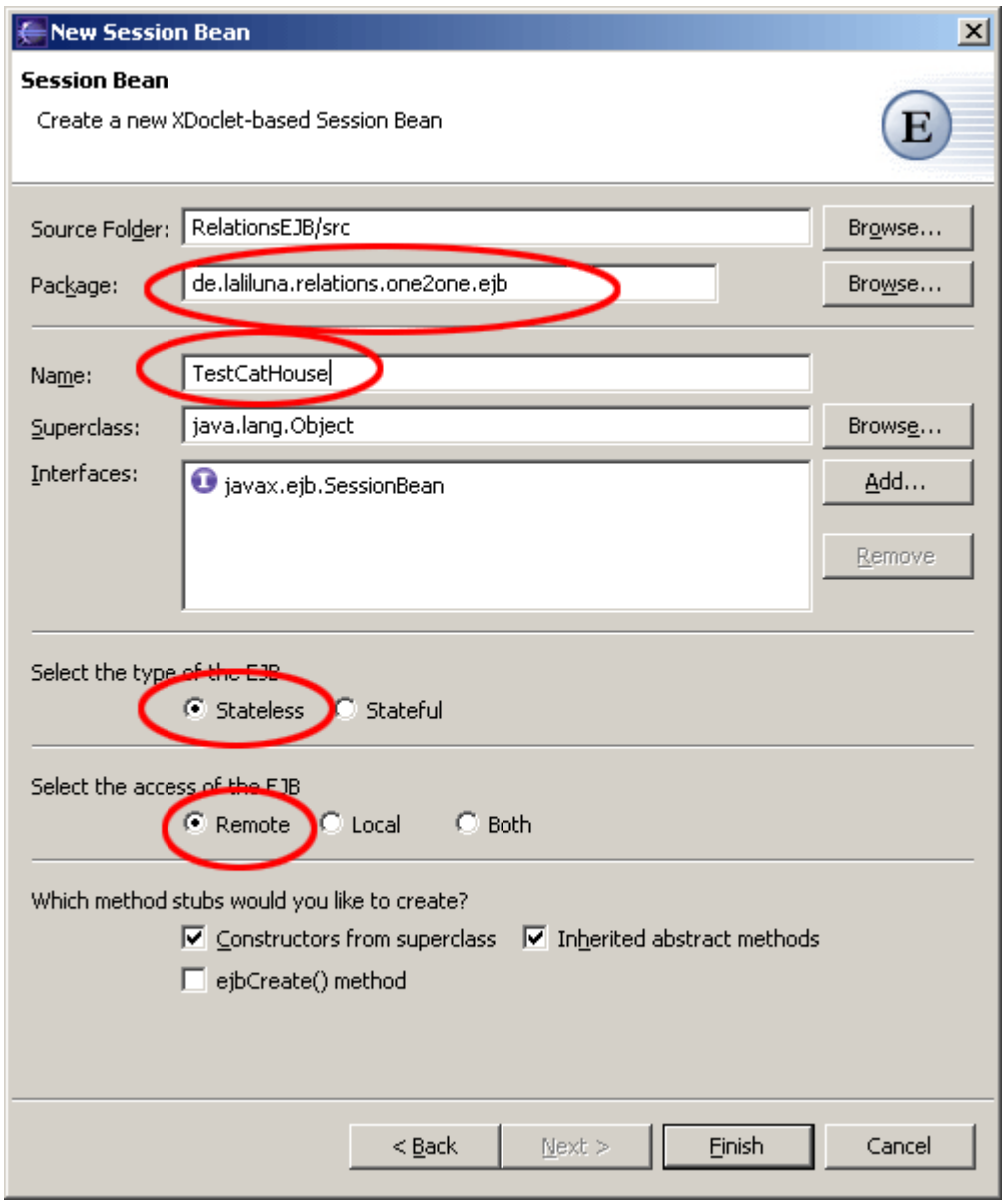
There are two things which are import.

First a container managed relationship does only work with local types. You cannot pass them remotely. So you should make sure, that you specify @ejb.interface-method view-type = "local" in order to prevent any mistakes.

Second the name of the relation must be the same on both sides.

Create a Session class to test our beans

Create a stateless session bean with the settings show below.



Then add the following method.

```
  public void createRelation() throws EJBException {
    // [lalilуna] 24.12.2004 we need a context to lookup the localHome
Interfaces
    InitialContext initialContext;
    try {
      initialContext = new InitialContext();
      CatLocalHome catLocalHome;
      /*
       * [lalilуna]
       * we use local interfaces, so there is not need to use
       * PortableRemoteObject.narrow
       */
      catLocalHome = (CatLocalHome) initialContext
          .lookup(CatLocalHome.JNDI_NAME);
      HouseLocalHome houseLocalHome;
      houseLocalHome = (HouseLocalHome)
initialContext.lookup(HouseLocalHome.JNDI_NAME);


      CatLocal catLocal = catLocalHome.create();
```

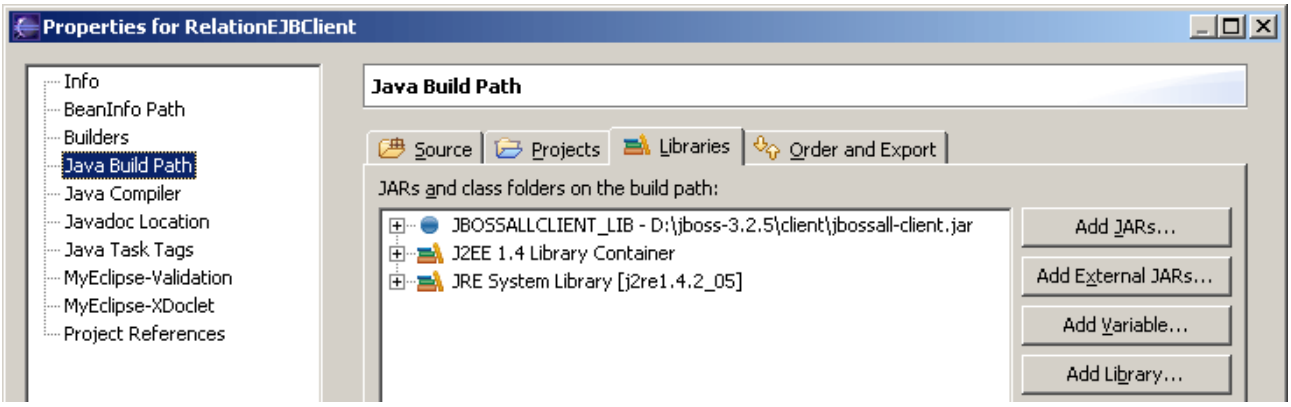
```
      catLocal.setName("Pete");
      HouseLocal houseLocal  = houseLocalHome.create();
      houseLocal.setColor("blue");

      // [lalilуna] now set the relation
      houseLocal.setCat(catLocal);

    } catch (NamingException e) {
      throw new EJBException(e);
    } catch (CreateException e) {
      throw new EJBException(e);
    }
  }
```

Run xDoclet and deploy your application to the jboss Server.

Create a simple Java Application as Client

Create a simple Java Project and open the project properties dialog.

Add the J2EE libraries with the Button „Add Library“ and the external jar „jbossall-client.jar“ which you can find the jboss directory.



Then change the class to the following.

```
public class TestRelations {

  private InitialContext context = null;

  /**
   * init the properties you need for the JNDI lookups
   * @return
   */
  private Properties initProperties(){
    Properties properties = new Properties();
    properties.put("java.naming.factory.initial","org.jnp.interfaces.NamingConte
xtFactory");
    properties.put("java.naming.factory.url.pkgs","org.jboss.naming:org.jnp.inte
rfaces");
    properties.put("java.naming.provider.url","jnp://localhost:1099");
    properties.put("jnp.disableDiscovery","true");
    return properties;
  }

  public static void main(String[] args) {
    TestRelations relations = new TestRelations();
```

```
    try {
      relations.context = new InitialContext(relations.initProperties());

      relations.testOne2OneCat();
    } catch (NamingException e) {
      e.printStackTrace();
    } catch (RemoteException e) {
      e.printStackTrace();
    } catch (CreateException e) {
      e.printStackTrace();
    }
  }

  /**
   * calls the session bean which tests the 1:1 relation
   * @throws NamingException
   * @throws RemoteException
   * @throws CreateException
   */
  private void testOne2OneCat() throws NamingException, RemoteException,
      CreateException {
    Object obj = context.lookup(TestCatHouseHome.JNDI_NAME);
    TestCatHouseHome catHouseHome = (TestCatHouseHome) PortableRemoteObject
        .narrow(obj, TestCatHouseHome.class);
    TestCatHouse catHouse = catHouseHome.create();
    catHouse.createRelation();
  }
}
```

Run it and look in the database if your entries have been created.

1:1 with access from one side (unidirectional)

Create two entity beans

As you know from dog theory books, each dog has only one favorite bone and does not share it at all.
Create two entity Beans Bone and Dog as shown in the first 1:1 example.
Bone has two attributes
String id as primary key
and
String size
Dog has also two attributes
String id as primary key
and
String name

Run xDoclet to have all interfaces generated.

Create the relation

In a unidirectional relation you have to use target tags to specify the other side of the relation.

```
   /**
   * @ejb.interface-method view-type = "local"
   * @ejb.relation name = "dog-bone" role-name = "dog has bone" target-role-name
= "bone of dog"
```

```
   * target-ejb = "Bone"
   * @jboss.relation fk-column = "bone_fk" related-pk-field = "id" fk-constraint
= "true"
   * @return
   */
  public abstract BoneLocal getBone();

  /**
   * @ejb.interface-method view-type = "local"
   * @param bone
   */
  public abstract void setBone(BoneLocal bone);
```

Create a session bean to test the relation

Add the following methods to your session bean. The first one is not very complicated. The second one show the behavior of a CMP relation when you delete the object which is part of a relation. The foreign key field is set to null.

```
  /**
   * An example business method
   *
   * @ejb.interface-method view-type = "remote"
   *
   * @throws EJBException Thrown if method fails due to system-level error.
   */
  public void createRelation() throws EJBException {
    // [lalilуna] 24.12.2004 we need a context to lookup the localHome
Interfaces
    InitialContext initialContext;
    try {
      initialContext = new InitialContext();
      DogLocalHome dogLocalHome;
      BoneLocalHome boneLocalHome;
      /*
       * [lalilуna]
       * we use local interfaces, so there is not need to use
       * PortableRemoteObject.narrow
       */
      dogLocalHome = (DogLocalHome)
initialContext.lookup(DogLocalHome.JNDI_NAME);
      boneLocalHome = (BoneLocalHome)
initialContext.lookup(BoneLocalHome.JNDI_NAME);

      DogLocal dogLocal = dogLocalHome.create();
      dogLocal.setName("Karl");
      BoneLocal boneLocal = boneLocalHome.create();
      boneLocal.setSize("big");

      // [lalilуna] now set the relation
      dogLocal.setBone(boneLocal);

    } catch (NamingException e) {
      throw new EJBException(e);
    } catch (CreateException e) {
      throw new EJBException(e);
    }
  }

  /**
   * An example business method
```

```
   *
   * @ejb.interface-method view-type = "remote"
   *
   * @throws EJBException Thrown if method fails due to system-level error.
   */
  public void deleteRelation() throws EJBException {
    // [laliluna] 24.12.2004 we need a context to lookup the localHome
Interfaces
    InitialContext initialContext;
    try {
      initialContext = new InitialContext();
      DogLocalHome dogLocalHome;
      BoneLocalHome boneLocalHome;
      /*
       * [laliluna]
       * we use local interfaces, so there is not need to use
       * PortableRemoteObject.narrow
       */
      dogLocalHome = (DogLocalHome)
initialContext.lookup(DogLocalHome.JNDI_NAME);
      boneLocalHome = (BoneLocalHome)
initialContext.lookup(BoneLocalHome.JNDI_NAME);

      DogLocal dogLocal = dogLocalHome.create();
      dogLocal.setName("Karl");
      BoneLocal boneLocal = boneLocalHome.create();
      boneLocal.setSize("big");

      // [laliluna] now set the relation
      dogLocal.setBone(boneLocal);
      boneLocal.remove();

    } catch (NamingException e) {
      throw new EJBException(e);
    } catch (CreateException e) {
      throw new EJBException(e);
    } catch (EJBException e) {
     throw new EJBException(e);
    } catch (RemoveException e) {
      // :TODO: Auto-generated catch block
      e.printStackTrace();
    }
  }
```

Changes to test client

Add the following method to your test client.

```
  private void testOne2OneDog() throws NamingException, RemoteException,
CreateException{
    Object object = context.lookup(TestDogBoneHome.JNDI_NAME);
    TestDogBoneHome dogBoneHome = (TestDogBoneHome)
PortableRemoteObject.narrow(object, TestDogBoneHome.class);
    TestDogBone testDogBone = dogBoneHome.create();
    testDogBone.createRelation();
    testDogBone.deleteRelation();
  }
```

1:n with access from both sides (bidirectional)

From each fish you can manufacture multiple fingers, the so called fish fingers. So this example

will help you for your next order from a food production company.

Create the two entity beans

Create an EJB named Fish and another one, named Finger. Both have two fields id as primary key and name as example field.

Generate the interfaces and then set the id with a UID in the ejbCreate method.

Have a look at the 1:1 bidirectional example, if you need more details.

It is a good idea to deploy the beans now to an application server to see if they work!

Create the relation

Changes to fish fish class.

```
 /**
   * @ejb.interface-method view-type = "local"
   * @ejb.relation name = "fish-fingers"
   * role-name = "fish becomes fingers"
   *
  * @return
   */
  public abstract Collection getFingers();

  /**
   * @ejb.interface-method view-type = "local"
   * @param fingers
   */
  public abstract void setFingers(Collection fingers);
```

Changes to Finger class.

```
 /**
   * @ejb.interface-method view-type = "local"
   * @ejb.relation name = "fish-fingers"
   * role-name = "fingers from fish"
   * cascade-delete = "yes"
   * @jboss.relation related-pk-field = "id" fk-column = "fish_id" fk-constraint
= "true"
   * @return
   */
  public abstract FishLocal getFish();

  /**
   * @ejb.interface-method view-type = "local"
   * @param fishLocal
   */
  public abstract void setFish(FishLocal fishLocal);
```

Important

Make sure that the view type is local. Relationship Management does only work with local interfaces.

The relation name must be identical in both declarations.

Create a SessionBean to test your classes

We have three methods testing different aspects.

```
/**
   * create relations between fish and finger
   *
   * @ejb.interface-method view-type = "remote"
   *
   * @throws EJBException Thrown if method fails due to system-level error.
   * @throws CreateException
   * @throws NamingException
   */
  public void createRelation1() throws EJBException, RemoteException {

    try {
      InitialContext initialContext = new InitialContext();
      FishLocalHome fishLocalHome;
      FingerLocalHome fingerLocalHome;
      // [lalil una] lookup the localHome interfaces in the JNDI context
      fishLocalHome = (FishLocalHome) initialContext
          .lookup(FishLocalHome.JNDI_NAME);
      fingerLocalHome = (FingerLocalHome) initialContext
          .lookup(FingerLocalHome.JNDI_NAME);
      // [lalil una] create entries in the db
      FishLocal fishLocal = (FishLocal) fishLocalHome.create();
      fishLocal.setName("Funny Fish");
      FingerLocal finger1 = (FingerLocal) fingerLocalHome.create();
      finger1.setName("small finger");
      FingerLocal finger2 = (FingerLocal) fingerLocalHome.create();
      finger2.setName("big finger");

      // [lalil una] create the relation
      finger1.setFish(fishLocal);
      finger2.setFish(fishLocal);

      // [lalil una] see if the fish knows its finger now
      Collection collection = fishLocal.getFingers();
      System.out.println("Fingers of fish " + fishLocal.getId());
      for (Iterator iter = collection.iterator(); iter.hasNext();) {
        FingerLocal finger = (FingerLocal) iter.next();
        System.out.println("ID: " + finger.getId() + " name: "
            + finger.getName());
      }

    } catch (NamingException e) {
      e.printStackTrace();
      throw new EJBException(e);
    } catch (CreateException e) {
      // :TODO: Auto-generated catch block
      e.printStackTrace();
    }

  }

  /**
   * create relations between fish and finger
   *
   * @ejb.interface-method view-type = "remote"
   *
   * @throws EJBException Thrown if method fails due to system-level error.
   * @throws CreateException
   * @throws NamingException
```

```
 */
public void createRelation2() throws EJBException, RemoteException {

  try {
    InitialContext initialContext = new InitialContext();
    FishLocalHome fishLocalHome;
    FingerLocalHome fingerLocalHome;
    // [lalilуna] lookup the localHome interfaces in the JNDI context
    fishLocalHome = (FishLocalHome) initialContext
        .lookup(FishLocalHome.JNDI_NAME);
    fingerLocalHome = (FingerLocalHome) initialContext
        .lookup(FingerLocalHome.JNDI_NAME);
    // [lalilуna] create entries in the db
    FishLocal fishLocal = (FishLocal) fishLocalHome.create();
    fishLocal.setName("Green Fish");
    FingerLocal finger1 = (FingerLocal) fingerLocalHome.create();
    finger1.setName("blue finger");
    FingerLocal finger2 = (FingerLocal) fingerLocalHome.create();
    finger2.setName("red finger");

    // [lalilуna] create the relation
    fishLocal.getFingers().add(finger1);
    fishLocal.getFingers().add(finger2);

    // [lalilуna] see if the fingers knows the fish now
    System.out.println(finger1.getFish());
    System.out.println(finger2.getFish());

    // [lalilуna] delete a finger
    finger1.remove();

    //    [lalilуna] see the remaining finger
    Collection collection = fishLocal.getFingers();
    System.out.println("Fingers of fish " + fishLocal.getId());
    for (Iterator iter = collection.iterator(); iter.hasNext();) {
      FingerLocal finger = (FingerLocal) iter.next();
      System.out.println("ID: " + finger.getId() + " name: "
          + finger.getName());
    }
  } catch (NamingException e) {
    e.printStackTrace();
    throw new EJBException(e);
  } catch (CreateException e) {
    // :TODO: Auto-generated catch block
    e.printStackTrace();
  } catch (EJBException e) {
    // :TODO: Auto-generated catch block
    e.printStackTrace();
  } catch (RemoveException e) {
    // :TODO: Auto-generated catch block
    e.printStackTrace();
  }

}

/**
 * create relations between fish and finger
 *
 * @ejb.interface-method view-type = "remote"
 *
 * @throws EJBException Thrown if method fails due to system-level error.
 * @throws CreateException
 * @throws NamingException
 */
```

```
public void createRelation3() throws EJBException, RemoteException {

  try {
    InitialContext initialContext = new InitialContext();
    FishLocalHome fishLocalHome;
    FingerLocalHome fingerLocalHome;
    // [laliluna] lookup the localHome interfaces in the JNDI context
    fishLocalHome = (FishLocalHome) initialContext
        .lookup(FishLocalHome.JNDI_NAME);
    fingerLocalHome = (FingerLocalHome) initialContext
        .lookup(FingerLocalHome.JNDI_NAME);
    // [laliluna] create entries in the db
    FishLocal fishLocal = (FishLocal) fishLocalHome.create();
    fishLocal.setName("Crazy Fish");
    FingerLocal finger1 = (FingerLocal) fingerLocalHome.create();
    finger1.setName("long finger");
    FingerLocal finger2 = (FingerLocal) fingerLocalHome.create();
    finger2.setName("very long finger");

    // [laliluna] create the relation
    finger1.setFish(fishLocal);
    finger2.setFish(fishLocal);

    // [laliluna] remove an item from the collection

    // [laliluna] see if the fish knows its finger now
    Collection collection = fishLocal.getFingers();
    collection.remove(finger2);

    System.out.println("Finger 2 is lonly and the fish is "
        + finger2.getFish());

  } catch (NamingException e) {
    e.printStackTrace();
    throw new EJBException(e);
  } catch (CreateException e) {
    // :TODO: Auto-generated catch block
    e.printStackTrace();
  }

}
```

1:n with access from the one side (unidirectional)

Our next example is from the color theory of beds. As you know it is best to paint your bed in only one color. This is basic knowledge. But there may be two beds having the same color. We decide that only the bed must know the colors and not vice versa. So we are having a unidirectional relationship.

Create the two entity beans

Create an EJB named Bed and another one, named Colour. Both have two fields id as primary key and name as example field.

Generate the interfaces and then set the id with a UID in the ejbCreate method.

Have a look at the 1:1 bidirectional example, if you need more details.

It is a good idea to deploy the beans now to an application server to see if they work before creating the relation.!

Create the relation

Changes to bed class.

```
      /**
       * @ejb.interface-method view-type = "local"
       * @ejb.relation name = "bed-colour"
       *  role-name = "bed has colour"
       *  cascade-delete = "yes"
       *  target-ejb = "Colour"
       *  target-role-name = "colour of bed"
       *  target-multiple = "yes"
       * @jboss.relation related-pk-field = "id" fk-column = "colour_fk" fk-
constraint = "true"
       *
       * @return
       */
      public abstract ColourLocal getColour();

      /**
       * @ejb.interface-method view-type = "local"
       *
       * @param name
       */
      public abstract void setColour(ColourLocal colour);
```

Important

Make sure that the view type is local. Relationship Management does only work with local interfaces.

The relation name must be identical in both declarations.

Create a SessionBean to test your classe

Create a SessionBean what was explained in more details in the „1:1 with access from both sides (bidirectional)“ above.

Add a business method as the following to test your beans.

```
/**
   * An example business method
   *
   * @ejb.interface-method view-type = "remote"
   *
   * @throws EJBException Thrown if method fails due to system-level error.
   */
  public void createRelation() throws EJBException {
    System.out.println("creating bed colour relation");
    try {
      InitialContext initialContext = new InitialContext();
      BedLocalHome bedLocalHome = (BedLocalHome) initialContext
          .lookup(BedLocalHome.JNDI_NAME);
      ColourLocalHome colourLocalHome = (ColourLocalHome) initialContext
          .lookup(ColourLocalHome.JNDI_NAME);

      BedLocal bedLocal = bedLocalHome.create();
      bedLocal.setName("my bed");
      ColourLocal colourLocal = colourLocalHome.create();
```

```
      colourLocal.setName("blue");

      bedLocal.setColour(colourLocal);

    } catch (NamingException e) {
      e.printStackTrace();
    } catch (CreateException e) {
      e.printStackTrace();
    }

  }
```

Deploy your project and add a method to your test client application.

1:n with access from the one side (unidirectional)

A tree has many leafs. This is not very complicated. If you grew up in a desert just accept it, as it is.

This example as the one before is unidirectional. Only one side knows the other side and has getter and setter to access the other side.

Create the two entity beans

Create an EJB named Tree and another one, named Leaf. Both have two fields id as primary key and name as example field.

Generate the interfaces and then set the id with a UID in the ejbCreate method.

Have a look at the 1:1 bidirectional example, if you need more details.

It is a good idea to deploy the beans now to an application server to see if they work before creating the relation.!

Create the relation

Changes to Tree class.

```
/**
   * @ejb.interface-method view-type = "local"
   * @ejb.relation name = "tree-leafs"
   * role-name = "tree has leafs"
   * target-ejb = "Leaf"
   * target-role-name = "leaf of tree"
   * target-multiple = "no"
   * target-cascade-delete = "yes"
   * @jboss.target-relation fk-column = "leaf_fk"
   * fk-constraint = "true"
   * related-pk-field = "id"
   *
   * @return
   */
  public abstract Collection getLeafs();

  /**
   * @ejb.interface-method view-type = "local"
   * @param leafs
   */
```

```
  public abstract void setLeafs(Collection leafs);
```

The target-cascade-delete tag will delete the target, i.e. the leaf, when the tree is deleted. If you prefer that the leaf continues to exist, you should specify target-cascade-delete=”no”. In this case the foreign key field will be set to null.

Important

Make sure that the view type is local. Relationship Management does only work with local interfaces.

The relation name must be identical in both declarations.

Create a SessionBean to test your classe

Create a SessionBean what was explained in more details in the „1:1 with access from both sides (bidirectional)“ above.

Add a business method as the following to test your beans.

```
  public void createRelation() throws EJBException
  {
    try {
      InitialContext initialContext = new InitialContext();
      TreeLocalHome treeLocalHome =
(TreeLocalHome)initialContext.lookup(TreeLocalHome.JNDI_NAME);
      LeafLocalHome  leafLocalHome = (LeafLocalHome)
initialContext.lookup(LeafLocalHome.JNDI_NAME);

      TreeLocal treeLocal = treeLocalHome.create();
      treeLocal.setName("Birke");

      LeafLocal leafLocal = leafLocalHome.create();
      leafLocal.setName("new leaf");
      treeLocal.getLeafs().add(leafLocal);

      /* laliluna
       * add this line to test the  behaviour of a cascade delete = "true" or
"false"
       * treeLocal.remove();
       */


    } catch (NamingException e) {
      e.printStackTrace();
    } catch (CreateException e) {
      e.printStackTrace();
    } catch (EJBException e) {
      e.printStackTrace();
    }
  }
```

m:n relation with access from both sides(bidirectional)

Some time ago I visited a school. I had many teachers and each teacher had many pupils and everybody had a lot of homework to do. We will forget about the homework and have a look at the pupil teacher relation. Our relations where actually quite good, some of the teacher were really fantastic, bla bli bla and then bla bla bla ...

But what is important is that we have a m:n relation between teachers and pupils.

Create the two entity beans

Create an EJB named Pupil and another one, named Teacher. Both have two fields id as primary key and name as example field.

Generate the interfaces and then set the id with a UID in the ejbCreate method.

Have a look at the 1:1 bidirectional example, if you need more details.

It is a good idea to deploy the beans now to an application server to see if they work before creating the relation.!

Create the relation

Changes to Teacher class.

```
/**
   * @ejb.interface-method view-type = "local"
   * @ejb.relation name = "teacher-pupil" role-name = "teacher has pupils"
   * @jboss.relation related-pk-field = "id" fk-column = "pupil_id"
   *  fk-constraint = "true"
   *
   *
   * @return
   */
  public abstract Collection getPupils();

  /**
   * @ejb.interface-method view-type = "local"
   * @param pupil
   */
  public abstract void setPupils(Collection pupil);
```

Changes to the Pupil class.

```
  /**
   * @ejb.interface-method view-type = "local"
   * @ejb.relation name = "teacher-pupil" role-name = "pupil belongs to teacher"
   * @jboss.relation fk-column = "teacher_id" related-pk-field = "id" fk-
constraint = "true"
   * @return
   */
  public abstract Collection getTeachers();

  /**
   * @ejb.interface-method view-type = "local"
   * @param teacherLocal
   */
  public abstract void setTeachers(Collection teachers);
```

Important

Make sure that the view type is local. Relationship Management does only work with local interfaces.

The relation name must be identical in both declarations.

Create a SessionBean to test your class

Create a SessionBean what was explained in more details in the „1:1 with access from both sides (bidirectional)“ above.

Add a business method as the following to test your beans.

```
  public void createRelation() throws EJBException
  {
    try {
      InitialContext initialContext = new InitialContext();
      PupilLocalHome pupilLocalHome = (PupilLocalHome)
initialContext.lookup(PupilLocalHome.JNDI_NAME);
      TeacherLocalHome teacherLocalHome = (TeacherLocalHome)
initialContext.lookup(TeacherLocalHome.JNDI_NAME);

      PupilLocal pupilLocal = pupilLocalHome.create();
      TeacherLocal teacherLocal = teacherLocalHome.create();

      teacherLocal.getPupils().add(pupilLocal);
    } catch (NamingException e) {
      e.printStackTrace();
    } catch (CreateException e) {
      e.printStackTrace();
    }
  }
```

m:n with access from one side (unidirectional)

Idiots do ask a lot of question. It is not very nice to say this. In Germany people say that there are no idiotic questions but only idiotic answers. Any way, we have a m:n relationship but this time we will only configure it on one side, the question side.

Create the two entity beans

Create an EJB named Idiot and another one, named Question. Both have two fields id as primary key and name as example field.

Generate the interfaces and then set the id with a UID in the ejbCreate method.

Have a look at the 1:1 bidirectional example, if you need more details.

It is a good idea to deploy the beans now to an application server to see if they work before creating the relation.!

Create the relation

Changes to Question class.

```
  /**
   * @ejb.interface-method view-type = "both"
   * @ejb.relation name = "idiots-questions"
   * role-name = "question of idiot"
   * target-role-name = "idiot has question"
   * target-ejb = "Idiot"
   * target-multiple = "yes"
   * @jboss.relation fk-column = "idiot_id" fk-constraint = "true" related-pk-
```

```
field = "id"
   * @jboss.target-relation fk-column = "question_id" fk-constraint = "true"
related-pk-field = "id"
   * @return
   */
  public abstract Collection getIdiots();

  /**
   * @ejb.interface-method view-type = "both"
   * @param idiots
   */
  public abstract void setIdiots(Collection idiots);
```

Important

Make sure that the view type is local. Relationship Management does only work with local interfaces.

The relation name must be identical in both declarations.

Create a SessionBean to test your class

Create a SessionBean what was explained in more details in the „1:1 with access from both sides (bidirectional)“ above.

Add a business method as the following to test your beans.

```
 public void createRelation() throws EJBException
  {
    System.out.println("test idiot question");
    try {
      InitialContext initialContext = new InitialContext();
      IdiotLocalHome idiotLocalHome = (IdiotLocalHome)
initialContext.lookup(IdiotLocalHome.JNDI_NAME);
      QuestionLocalHome questionLocalHome = (QuestionLocalHome)
initialContext.lookup(QuestionLocalHome.JNDI_NAME);

      IdiotLocal idiotLocal = idiotLocalHome.create();
      idiotLocal.setName("Sebastian - this is me");
      QuestionLocal q1 = questionLocalHome.create();
      q1.setName("Bla");
      QuestionLocal q2 = questionLocalHome.create();
      q2.setName("Blu");
      QuestionLocal q3 = questionLocalHome.create();
      q3.setName("Bli");

      q1.getIdiots().add(idiotLocal);
      q2.getIdiots().add(idiotLocal);
      q3.getIdiots().add(idiotLocal);


    } catch (NamingException e) {
      e.printStackTrace();
    } catch (CreateException e) {
      e.printStackTrace();
    }


  }
```

Congratulations that's it. You have finished all types of relations.

We have provide a separate PDF including a “LaLiLuna schnelle Seite”, which shows all informations in a short form on one page.