**0. JPA on NETBEANS**

Follow this STEPs to create a Netbeans Project able to use the JPA framework.

**STEP 1 – Create a Database on MySQL**

Create a new Database on MySQL. It’s not needed to create any table inside it.

create database DBName

use DBName

**STEP 2 – Create a new Project on NetBeans**

File > New Project >

1. **Choose Project**

* Choose the “Categories:” and “Projects:”
* Click on “Next >”

1. **Name and Location**

* Select the “Project Name:”
* Click on “Finish”

**STEP 3 – Create a JPA Class**

Select a Project

File > New File…>

1. **Choose File Type**

* On “Categories:” select the directory named “Persistence”
* On “File Types” select the voice “Entity Class”
* Click on “Next >”

1. **Name And Location**

* Select the “File Name”
* Click on “Next >”

1. **Provider and Database**

* Write the “Persistence Unit Name” (by Default ProjectNamePU)
* Select in the Combo Box “Persistence Library:” the voice “EclipseLink(JPA)”
* Select in the Combo Box “Database Connection” the voice “New Database Connection …”
* Now it will appear a new window named **Local Driver**
* For “Driver:” select “MySQL (Connector/ J Driver)”
* Click on “Next >”
* Now it will appear a new window named **Customize Connection**
* Select the Host , Port and Database name
* Click on “Finish”
* Now Select the “Table Generation Strategy” (Create or Drop and Create)
* Click on “Finish”

Now in your project will appear a new file .java. It has by default a Key of type Long that can be modified ( … ).

Now if it’s need to create a new file:

1. **Choose File Type**

* On “Categories:” select the directory named “Persistence”
* On “File Types” select the voice “Entity Class”
* Click on “Next>”

1. **Name And Location**

* Select the “File Name”
* Click on “Next >”

It will have the same parameter of the first created.

N.B: in **2. Name and Location** the “Primary Key Type” is asked but we suggest modifying it on

File .java generated.

**1. How to create OneToMany/ManyToMany Relationship in JPA**

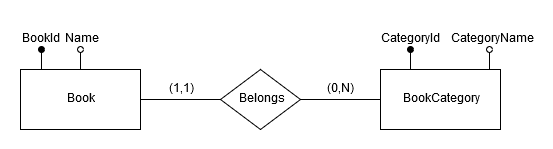
* 1. **JPA One to Many Relationship**

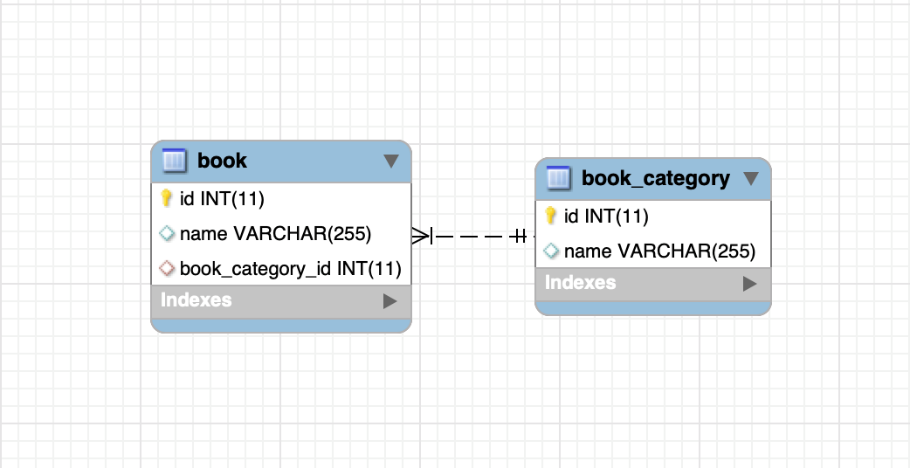
A *many-to-one relationship* occurs when multiple records in a table (table\_1) are associated with one record in another table (table\_2) and a record in table\_2 is associated with more records of table\_1.

Let’s make an example:

Each Book belongs to a specific genre (Horror, Fiction, Fantasy ecc...) to one genre belongs many Books.

In the ER-Diagram a OneToMany/ManyToOne needs a Relation between the entity Book and BookCategory.





book.book\_category\_id is a foreign key references to book\_category.id.

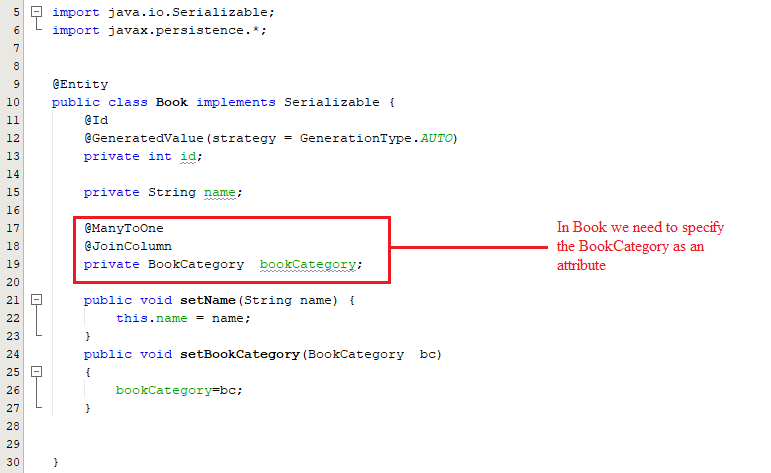
Thanks to JPA we don’t need to project the Database and specify the Table format but it will be automatically generated.

**1.1.1 Implementation of a JPA class with OneToMany relationship**

If we follow the previous STEPs of the tutorial (in particular STEP 3) it will be generated a class JPA compatible.

Following the tutorial let’s create the class Book and BookCategory.

**Book:**



In Book we need to use the annotation @ManyToOne (Many Books belongs to One Category) and before specifying the join attribute we need to use the annotation @JoinColumn.

Each Object Book has inside a reference to the Object BookCategory to which it belongs. The attribute BookCategory must be set before committing a Book Object in the DB.

**BookCategory:**

****

In BookCategory we need to use the annotation @OneToMany (to One Category belongs many Books) before specifying the set of books Object that join with our BookCategory Object.

N.B: inside the annotation @OneToMany must be specified the name of the attribute specified in the class that has the ManyToOne relationship with our class (In this case Book.bookCategory) followed by “private Set<Class\_That\_Joins\_With\_This\_Class> AttributeName;”.

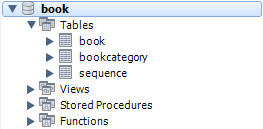
**Main:**

****

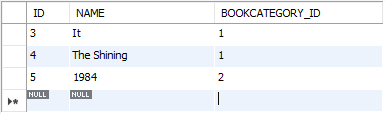
This main contains a simple Create operation for some Book Objects and BookCategory Objects using the EntityManager and EntityManagerFactory to store them in the DB.

**In MySQL:**

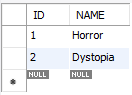
After the execution of the main JPA Automatically generate the tables book and bookcategory:

****

Content of book:

****

Content of bookcategory:

****

**1.2 JPA Many to Many relationship**

A particular relationship that we could find in a E-R Diagram is the many to many. A *many-to-many relationship* occurs when multiple records in a table are associated with multiple records in another table. For example, a many-to-many relationship exists between customers and products: customers can purchase various products, and products can be purchased by many customers.

Let’s see an example that allows us to create a many to many relationship between two entities using JPA.

The example that we take in account represents the relationship between Post (for ex. In a Social Network) and Tag. We know that one post can contain 0 or more tags and a tag can be related to 0 or more posts.

Immagine che contiene screenshot

Descrizione generata automaticamente

This relation is made possible by the use of a third table, called post\_tag, which contains all the possible association between posts and tags. **How we can create this using JPA?**

* In order to create the entity we have to follow the steps that we have seen previously, defining all the attributes of the entities classically.
* To create the relationship we have to define in both the classes a set of elements which will represent the join table content.

(Entire structure Post.java)

Immagine che contiene screenshot

Descrizione generata automaticamente

(From file Post.java)

Immagine che contiene screenshot

Descrizione generata automaticamente

(Entire strucure Tag.java)

Immagine che contiene screenshot

Descrizione generata automaticamente

(From file Tag.java)



P.S.1: **The mappedBy attribute of the posts association in the Tag entity marks that, in this bidirectional relationship, the Post entity owns the association. This is needed since only one side can own a relationship, and changes are only propagated to the database from this particular side.**

P.S.2 **: Is very recommended to use the class Set instead the class List in order to manage the Many to Many relationship. This because if class List is used, Hibernate instead of deleting just one post\_tag entry, removes all post\_tag rows associated to the given post\_id and reinserts the remaining ones back afterward and so is very inefficient.**

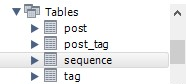
* After that we can create our interested elements. In this example post and tag:

**Immagine che contiene screenshot

Descrizione generata automaticamente**

**In MySQL:**

After the execution of the main JPA Automatically generate the tables Post, Tag and Post\_tag:



Content of Post table:

Immagine che contiene screenshot

Descrizione generata automaticamente

Content of Tag table:

Immagine che contiene screenshot

Descrizione generata automaticamente

Content of the join table post\_tag:

Immagine che contiene screenshot

Descrizione generata automaticamente

**2. CRUD Operations using JPA**

**2.1 Insert Operation On The DB**

As we have seen in the previous chapter, we have the possibility to perform queries in SQL format using the createNativeQuery( ) method. In particular we can implement an **insert** operation on the db doing:

Immagine che contiene screenshot

Descrizione generata automaticamente

In this way, which is not properly efficient, we are losing the advantages introduced by JPA and in particular by the concept of entity.

We already know that a Persistence Class, also called entity, represents into a SQL db a table.

Immagine che contiene screenshot

Descrizione generata automaticamente

When we create an instance of a Persistence entity, we are also creating a “record“ of the SQL table associated. So we can use the created instance in order to insert a record into the DB using a particular method called **Persist( )** . The related syntax is:

Immagine che contiene screenshot

Descrizione generata automaticamente

Now that we have created the “record” represented by the Post object, we can insert it into the DB doing:



The **persist( )** method receives as a parameter the object that we want to insert into the table.

We have to remember that the Object attributes (in this example the attributes ID, Title of Post object) will represent the column of the database’s table.

After the commit, in the table we will have:

Immagine che contiene screenshot

Descrizione generata automaticamente

**2.2 Read Operations On DB**

When we have to extract information from a DB we have to perform a read operation in order to collect all the desired information. We know how to do this in SQL language, but **how we can perform a query using JPA?**

JPA gives us a lot of tools in order to perform query on a database.

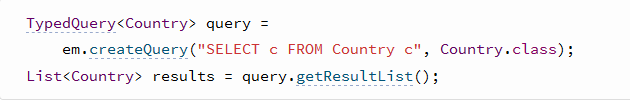
JPQL is a part of JPA which defines platform-independent object-oriented query language. It allows us to do, in a very similar way to SQL, queries based on our entity model defined trough JPA. JPQL uses the entity object model instead of database tables to define a query. The syntax used by JPQL is strictly similar to the SQL syntax except for some constructs that are not defined in JPQL (f.e: IN clause etc.). The way to create and perform these queries is:

Immagine che contiene screenshot

Descrizione generata automaticamente

For a better understanding we can see two examples:

1.



2.

Immagine che contiene screenshot

Descrizione generata automaticamente

Using the method **getResultsList( )** on the Query/TypedQuery< > object, we execute the query and store the result into a list of object.

In some cases we have to perform very particular queries that are very difficult and which use some constructs that are not defined in JPQL(f.e. Derived Table). JPA allows us to perform this type of complex queries using the SQL syntax due to respect the databases structure. In particular the JPA method that permits us to create SQL queries is : **createNativeQuery()**. This method receives as a parameter the SQL query that we want to perform.

Immagine che contiene screenshot

Descrizione generata automaticamente

Using this particular method we are able to perform, on our database, very complex queries which contain some constructs that we cannot use through JPQL like for example the Derived Table. **When we use the createNativeQuery( ) we have to pay attention on the structure of the database created by JPA!**

In both Native queries and JPQL queries we are able to set parameters instead of insert values manually. In particular we have a function called **setParameter(int index, int value)/setParameter(String name, String value)** that allow us to insert in specified position of the query the value that we want.

**.Named Parameters**

**Immagine che contiene screenshot

Descrizione generata automaticamente**

**.Ordinal Parameters**

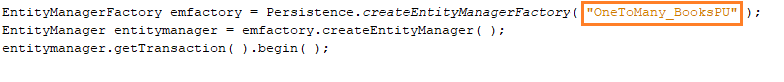
Immagine che contiene screenshot

Descrizione generata automaticamente

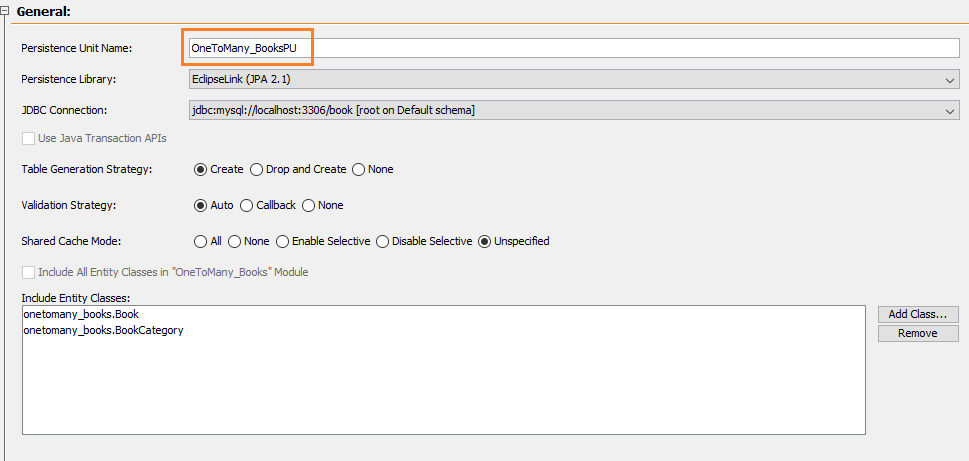
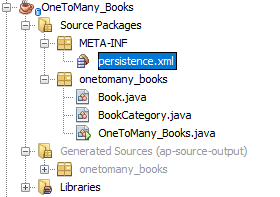
**2.3 Update Operation**

To update a tuple in the database:

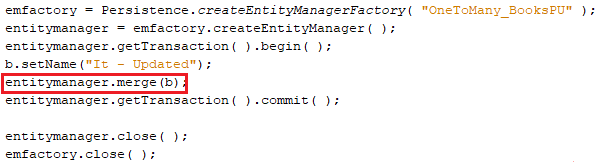
1. Create the EntityManagerFactory and the EntityManager



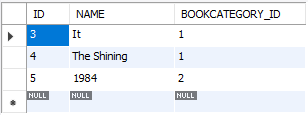
N.B: In the red box insert the name of the Persistent Unit selected during the creation of the first JPA class. This information is also contained in the persistence.xml file.



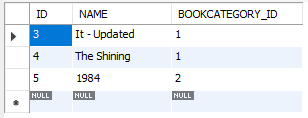
1. Modify the Object attribute that need to be updated using the Set Method (If the Object is not present use the “Find” method to get it)



Before the Update:



After the Update:



N.B: Even the Join attribute can be modified in this way

**2.4 Delete Operation**

1. Create the EntityManagerFactory and the EntityManager
2. Use the Find to get the reference of the desired Object
3. Delete the Object and Commit

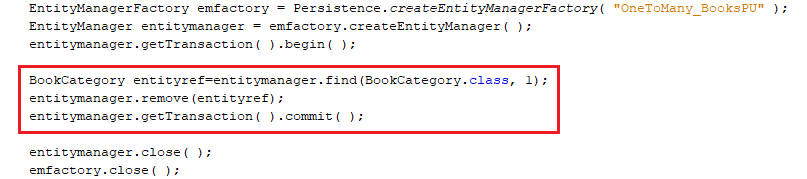


Table BookCategory before the Delete:

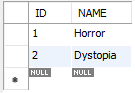
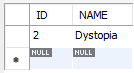


Table BookCategory after the Delete:



N.B: We are deleting an Object containing the Join column of an Object. After the delete operation each Object saved in the Database that has the deleted Object as Join attribute will be also deleted. In this case will be deleted from the Database also every book with BOOKCATEGORY\_ID = = 1.

Table Book before the delete the tuple in BookCategory:

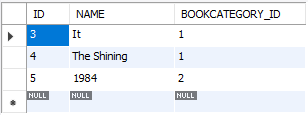


Table Book after the delete the tuple in BookCategory:

