Abstract

How to build up a JPA project in NetBeans, how to instantiate some many-to-many and one-to-many relationships between JPA entities, and how to make very simple CRUD operation using JPA

JPA TUTORIAL

Dario LORENZONI

Raffele NOCERINO

Marco PETTORALI

Riccardo XEFRAJ

Sommario

[0. JPA on NETBEANS 2](#_Toc23441706)

[0.1 Create a Database on MySQL 2](#_Toc23441707)

[0.2 Create a new Project on NetBeans 2](#_Toc23441708)

[0.3 Create a JPA Class 2](#_Toc23441709)

[1. How to create OneToMany/ManyToMany Relationships in JPA 4](#_Toc23441710)

[1.1 JPA One to Many Relationship 4](#_Toc23441711)

[1.1.1 Implementation of a JPA class with OneToMany relationship 5](#_Toc23441712)

[Book: 5](#_Toc23441713)

[BookCategory: 5](#_Toc23441714)

[Main: 6](#_Toc23441715)

[MySQL: 6](#_Toc23441716)

[1.2 JPA Many to Many relationship 8](#_Toc23441717)

[MySQL: 11](#_Toc23441718)

[2. CRUD Operations using JPA 13](#_Toc23441719)

[2.1 Insert Operations 13](#_Toc23441720)

[2.2 Read Operations 15](#_Toc23441721)

[2.3 Update Operations 17](#_Toc23441722)

[2.4 Delete Operations 19](#_Toc23441723)

# 0. JPA on NETBEANS

The first thing we need to do before start using the JPA framework is to create a new NetBeans Project and then a new JPA class. The necessary steps in order to do so are explained in the rest of the paragraph.

## 0.1 Create a Database on MySQL

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

create database DBName

use DBName

## 0.2 Create a new Project on NetBeans

File > New Project >

1. **Choose Project** 
   * Choose “Categories:” and then “Projects:”
   * Click on “Next >”
2. **Name and Location**
   * Select “Project Name:”
   * Click on “Finish”

## 0.3 Create a JPA Class

Select a Project

File > New File…>

1. **Choose File Type**

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

1. **Name And Location**

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

1. **Provider and Database**

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

A new .java file will be created in the project. By default, it has a Key, which type is “Long”, but it can be customized.

Now, create a new file if needed:

1. **Choose File Type**

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

1. **Name And Location**

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

*(N.B: “Primary Key Type” field is required, but we suggest modifying it directly on the newly generated .java file.)*

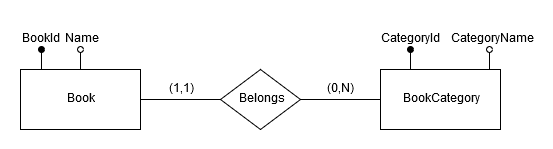
This new file has the same parameters of that one created before.

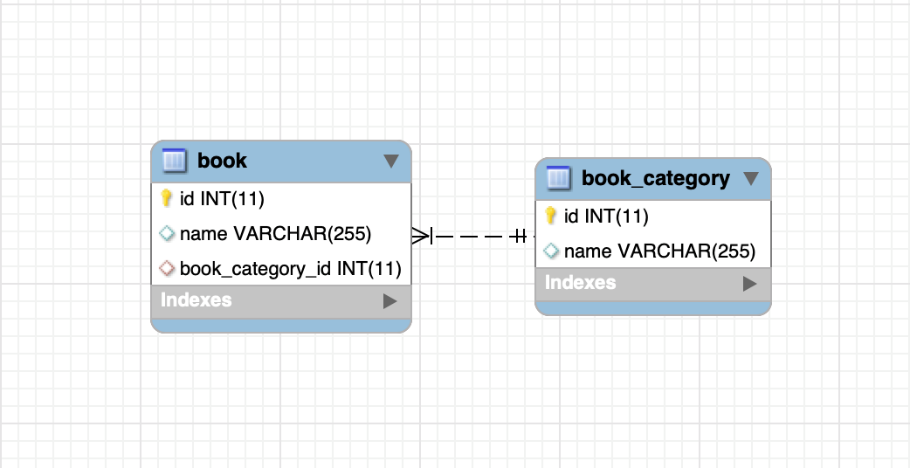
# 1. How to create OneToMany/ManyToMany Relationships in JPA

## 1.1 JPA One to Many Relationship

A *many-to-one relationship* is necessary when multiple records in a table (for instance called ‘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...), but many Books could relate to a single genre. In the ER-Diagram a OneToMany/ManyToOne requires a Relation between the entity Book and BookCategory (note that ‘book.book\_category\_id’ is a foreign key reference to book\_category.id).



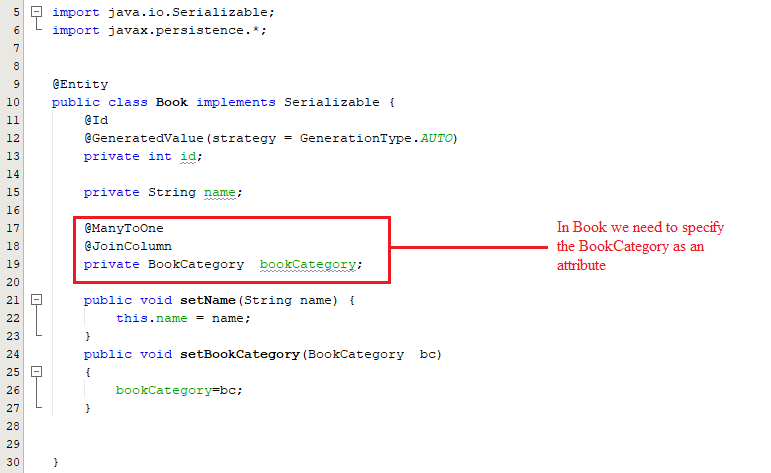


Thanks to JPA we don’t need to project the Database and specify the Table format but it will be automatically generated.

## 1.1.1 Implementation of a JPA class with OneToMany relationship

In the previous steps of the tutorial (in particular step 0.3) we have seen how to generate a JPA compatible class. Let’s now create ‘Book’ and ‘BookCategory’ classes.

### Book:



In Book class, we need to use the @ManyToOne annotation (many Books belong to One Category) and we need to use the annotation @JoinColumn before specifying the join attribute.

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

### BookCategory:

****

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

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

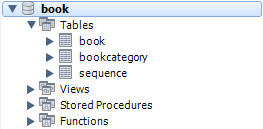
### Main:

****

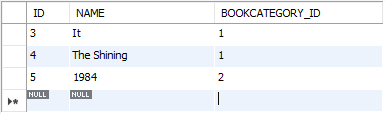
This main contains a simple Create operation for some Book Objects and BookCategory Objects using the EntityManager and EntityManagerFactory to store them in the DB.

### MySQL:

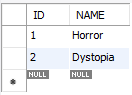
After the execution of the main JPA Automatically generate the tables ‘book’ and ‘bookcategory’:

****

#### Content of book table:

****

#### Content of bookcategory table:

****

## 1.2 JPA Many to Many relationship

A particular relationship that we could find in a E-R Diagram is the ‘many to many’ one. A *many-to-many relationship* is necessary 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 show us how to create a many to many relationship between two entities using JPA.

The example we take in account is related to a Social Network context, and represents the relationship between Post and Tag. We know that one post can contain zero or more tags and a tag can be related to zero or more posts.

Immagine che contiene screenshot

Descrizione generata automaticamente

This relation is made possible by using 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.
* In order to create the relationship we have to define a set of elements, representing the join table content, in both classes.

#### (Post.java file structure)

Immagine che contiene screenshot

Descrizione generata automaticamente

#### (From Post.java file)

Immagine che contiene screenshot

Descrizione generata automaticamente

#### (Tag.java file strucure)

Immagine che contiene screenshot

Descrizione generata automaticamente

#### (From Tag.java file)

#### 

N.B. The **mappedBy** attribute of the **posts’** association in the **Tag** entity indicates that, in this particular bidirectional relationship, the **Post** entity owns the association. This is necessary, since only one side can own a relationship, and changes are only propagated from this specific side to the database.

N.B. It’s strongly recommended to use Set class instead of List class in order to manage the Many to Many relationship, because if List class is used, Hibernate will remove all **post\_tag** rows associated to the given **post\_id** and reinserts the remaining ones back afterward instead of just deleting one **post\_tag** entry, and this is very inefficient.

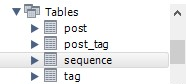
We can now create our elements of interest (posts and tags, in this example).

Immagine che contiene screenshot

Descrizione generata automaticamente

### MySQL:

After the execution of the main, JPA automatically generates Post, Tag and Post\_tag tables:



#### 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 Operations

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 by doing the following operations:

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 a table in a SQL db.

Immagine che contiene screenshot

Descrizione generata automaticamente

When we create an instance of Persistence entity, we are also creating a “record“ in the SQL table associated. 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 by doing:



The **persist()** method receives the object that we want to insert into the table as a parameter. We have to keep in mind that the Object attributes (in this example, the ID and the Title of Post ones) 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

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 queries on a database.

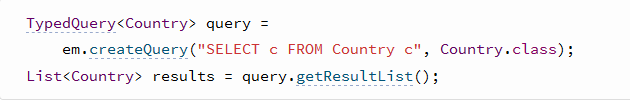
JPQL is a part of JPA, which defines a platform-independent object-oriented query language. In a very similar way to SQL, it allows us to do queries based on our entity model defined trough JPA. JPQL uses the entity object model instead of database tables to define a query. The syntax used by JPQL is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

Immagine che contiene screenshot

Descrizione generata automaticamente

For a better understanding, we can see two examples:

#### Example #1



#### Example #2

Immagine che contiene screenshot

Descrizione generata automaticamente

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

In some cases, we might have to perform very particular queries that are very difficult and that use some constructs that are not defined in JPQL (e.g. derived tables). JPA allows us to perform this type of complex queries using the SQL syntax due to respect the tables structure. In particular, the JPA method that allow us to create SQL queries is **createNativeQuery()**; this method receives the SQL query that we want to perform as a parameter.

Immagine che contiene screenshot

Descrizione generata automaticamente

In this way, we are able to perform, very complex queries on our database, just by using this particular method containing some constructs that we could not use through JPQL, like for example derived tables. Please, note that **when we use createNativeQuery(), we have to pay attention at the structure of the database created by JPA!**

In both Native and JPQL queries, we are still able to set parameters instead of inserting values manually. In particular, we have a function called **setParameter(int index,Object value) /setParameter(String id, Object value)**,that allow us to insert a specific value in a specific position in the query code. We have to keep in mind that we cannot mix this two methods to set parameters in the same query

#### .Named Parameters

When we have to set a parameter of a query, we can use the named parameters one. Inside the string, which represents the query, we have to put **:id** in correspondence of the attribute that we want to set using the setParameter(String id ,Object value). The id represents a JPQL identifier that is used after in order to explain to the setParameter method what the target parameter is.

**Immagine che contiene screenshot

Descrizione generata automaticamente**

#### .Ordinal Parameters

Another possible way that we can use in order to tell to the setParameter method what parameter has to be set, is using ordinal parameters one. In this case we use the syntax **?x** where x is a number used to identify the parameter. After when we will call the setParameter, we will pass to it the number of the parameter that has to be set and the corresponding value.

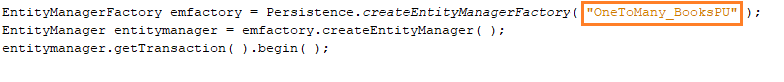
Immagine che contiene screenshot

Descrizione generata automaticamente

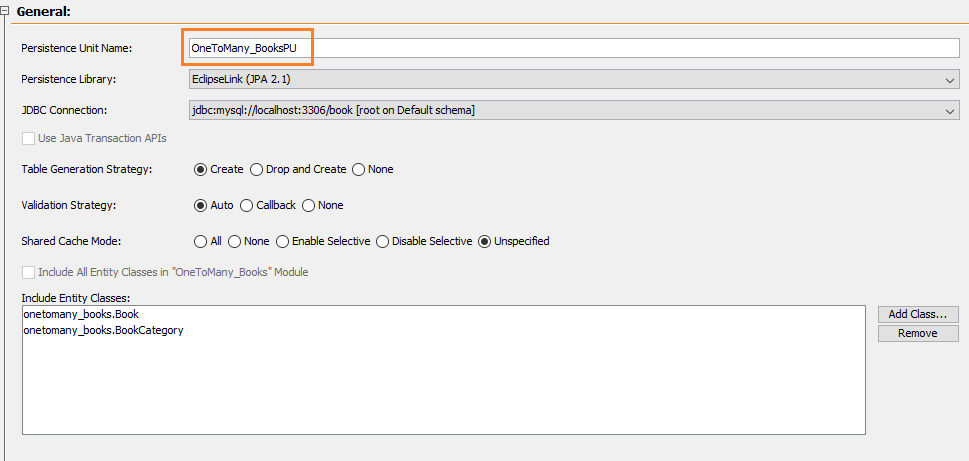
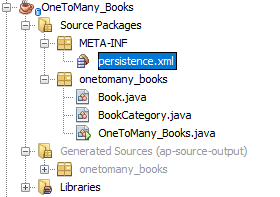
## 2.3 Update Operations

In order to update a tuple in the database:

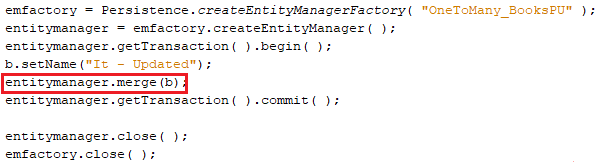
1. Create the EntityManagerFactory and the EntityManager



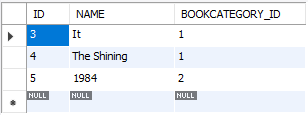
*(N.B: In the red box place, you have to insert the name of the Persistence Unit selected during the creation of the first JPA class. This information is also contained in the persistence.xml file.)*



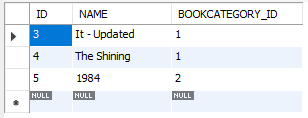
1. Modify the Object attribute which we want to be updated by using the Set Method (If the Object is not present, use first the “Find” method to get it)



#### Before the Update:



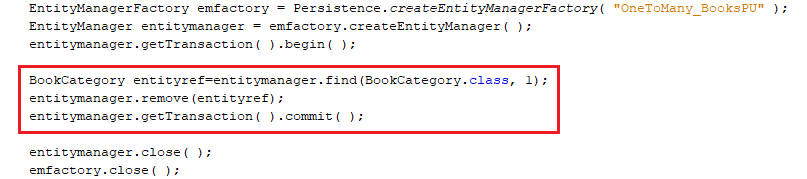
#### After the Update:



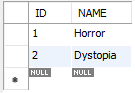
N.B: Join attributes can be modified in this way, too.

## 2.4 Delete Operations

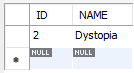
1. Create the EntityManagerFactory and the EntityManager
2. Use the Find method to get the reference to the desired Object
3. Delete the Object and Commit



#### Table BookCategory before the Delete:

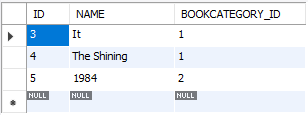


#### Table BookCategory after the Delete:



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

#### Table Book before the delete the tuple in BookCategory:



#### Table Book after the delete the tuple in BookCategory:

