



JPA Tutorial

ONE-TO-MANY RELATIONSHIPS AND CRUD OPERATIONS

One-to-many relationship



In this example we consider a 1:N relationship between the Progetto table and the Finanziamento table.

Entities

```
@Entity
@Table(name = "finanziamento", schema = "esercizio1", catalog = "")
public class FinanziamentoEntity {
    private int id;
    private Integer budget;

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)

    //get and set

    @Basic
    @Column(name = "budget", nullable = true)

    //get and set

    //...more code...
```

```
@Entity
@Table(name = "progetto", schema = "esercizio1", catalog = "")
public class ProgettoEntity {
    private int id;
    private String nome;
    private String descrizione;
    private Integer budget;
    private AziendaEntity azienda;

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
    //get and set

    @Basic
    @Column(name = "nome", nullable = true, length = 500)

    //get and set

    //...more code...
```

This is the representation of the traduction as JPA Entities of the table Progetto and Finanziamento without considering the Possesso relationship

Annotations

```
public class FinanziamentoEntity {  
  
    //...more code...  
  
    private ProgettoEntity progetto;  
    @ManyToOne(fetch = FetchType.LAZY)  
        @JoinColumn(name = "progetto_id", nullable = false)  
    public ProgettoEntity getProgetto(){ return this.progetto ;};  
    public void setProgetto(ProgettoEntity _progetto){ this.progetto  
= _progetto; };  
  
    //...more code...  
  
}
```

```
public class ProgettoEntity {  
  
    //...more code...  
  
    private List<FinanziamentoEntity> myStakes;  
  
    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true, fetch =  
FetchType.LAZY, mappedBy = "progetto")  
    public List<FinanziamentoEntity> getMyStakes(){ return  
this.myStakes ;};  
    public void setMyStakes(List<FinanziamentoEntity> _stakes){  
this.myStakes = _stakes; };  
  
    //...more code...  
  
}
```

If you need to know which project is linked to each funding you need to use the annotation **@ManyToOne**. In this way it will be possible to create a column, in the financing table, "financing_id" that contains the key of the Project entity, through the annotation **@JoinColumn**. To achieve this you need to insert an object of type **ProjectEntity** in the class **FinancingEntity** that is instantiated each time a project is created.

In case we need to keep in memory all the funds owned by a single project, to be able to access them through the entity **ProjectEntity** instead you need to use the annotation **@OneToMany**

Annotations: Solution adopted

In our case we only want access the fundings for each project. For Hibernate it would be convenient to use only a one-way relationship: in this case it would be enough to access ProgettoEntity, without making further calls on the database.

Create

A FinancingEntity instance is built as a normal Java object. To insert this object into the database, an i.e. transaction is opened first. `e.getTransaction().begin()`

A call to this function associates the 'Object' object to an EntityManager and changes its status to Managed.

The new object will then be stored in the database when the transaction commit is called. The entity manager will then be closed.

```
public <T> T create( T entity ){
    if( entity == null || em == null)
        return null;

    try {
        em.getTransaction().begin();
        em.persist(entity);
        em.getTransaction().commit();
        return entity;
    } catch( Exception ex ){
        ex.printStackTrace();
        return null;
    }
}
```

The `persist()` function throws a `TransactionRequiredException` exception if there is no active transaction and a `IllegalArgumentException` if the argument of the function is not an instance of a class, in fact only instances of entity classes can be stored within the database.

Read

Each object can be identified and retrieved within the database using its class and primary key.

To read an object from the database it is possible, once a transaction is opened, to use the `find()` function that will return an instance of the same class passed as first argument. The entity manager will then be closed by the caller of that function.

```
public <T> T read(Class<T> tableClass, String id) {  
    if (em == null)  
        return null;  
  
    T result = null;  
  
    try {  
        em.getTransaction().begin();  
        result = em.find(tableClass, id);  
    } catch (Exception ex) {  
        ex.printStackTrace();  
    } finally {  
        em.getTransaction().commit();  
        return result;  
    }  
}
```

In this case the function throws an exception of type `IllegalArgumentException` if the class passed as argument is not an Entity Class.

Update

Before calling this function that will update the entity in the database, the object is first taken and passed as an argument.

A transaction is opened and then merged the entity -i.e. `em.merge(entity)`- with what is already in the database, updating it accordingly.

Finally, the transaction is committed and the `entityManager` is closed by the caller of that function.

```
public <T> T update(T entity){
    if( entity == null || em == null )
        return null;

    try{
        em.getTransaction().begin();
        T result = em.merge(entity);
        em.getTransaction().commit();
        return result;
    }catch(Exception ex){
        ex.printStackTrace();
        return null;
    }
}
```

The function throws an exception of type `IllegalArgumentException` if the class passed as argument is not an Entity Class.

Delete

The argument for this function is the type and primary key of the object to be deleted from the database. This two pieces of information are used to retrieve the object once a transaction has been opened. This object is then passed to the construct remove -i.e. `em.remove(old)`- through which the object will actually be removed deleted once the transaction is committed. The entity manager will finally be closed by the caller of the transaction.

```
public <T> T delete(Class<T> type, int id) {  
    if (em == null)  
        return null;  
    try {  
        em.getTransaction().begin();  
        T old = em.getReference(type, id);  
        em.remove( old );  
        em.getTransaction().commit();  
        return old;  
    } catch (EntityNotFoundException ex) {  
        return null;  
    }  
    catch (Exception ex) {  
        ex.printStackTrace();  
        return null;  
    }  
}
```

If there is no object with such a class and primary key, an exception of the type `EntityNotFoundException`