University of Pisa
MSc in Computer Engineering
Information Systems

Social Network - JPA report

Sara Lotano
Lorenzo Simi
Filippo Storniolo
Matteo Suffredini

Academic Year 2019/2020

Contents

1 Introduction

The Java Persistance API (JPA) is an approach to ORM, which stands for Object-Relational Mapping and it is used to map Java objects to database tables and vice versa, allowing to work directly with objects rather than with SQL statements.

The mapping between database tables and Java objects is defined exploiting persistence metadata, typically defined via annotation in the Java class or in special xml file. One row in the table will represent a persisted instance of the class.

In JPA there is the concept of Persistence Context, that in practice is a cache with its own connection to the database and the latter is not shared with other.

There are several implementations available for JPA and the most popular ones are Hibernate, EclipseLink and Apache OpenJPA.

From now on it will be presented the case of Hibernate.

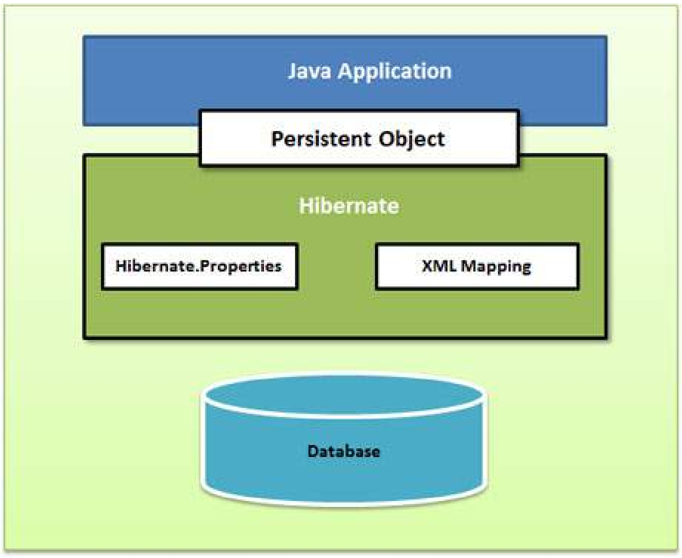

Figure 1: Architecture

In Hibernate implementation, the Persistence Context consists of the EntityManager, which provides the operations from and to the database, e.g. find objects, persists them, remove objects from the database, etc.
The EntityManager is created by another component named EntityManagerFactory, which has to manage resources efficiently (e.g. a pool of sockets) and to provide an efficient way to construct even multiple EntityManager instances for the database.

1.1 Persistence Context

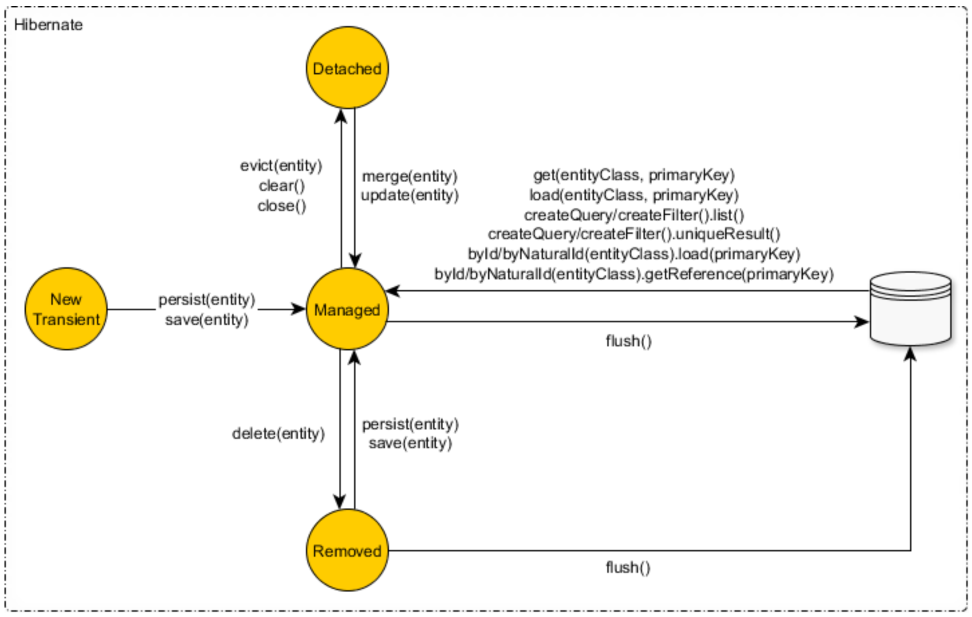
Hibernate
Detached
evict(entity)
clear()
close()
merge(entity)
update(entity)
get(entityClass, primaryKey)
load(entityClass, primaryKey)
createQuery/createFilter().list()
createQuery/createFilter().uniqueResult()
byId/byNaturalId(entityClass).load(primaryKey)
byId/byNaturalId(entityClass).getReference(primaryKey)
New Transient
persist(entity)
save(entity)
Managed
flush()
delete(entity)
persist(entity)
save(entity)
Removed
flush()

Figure 2: States managed by Entity Manager

1.1.1 Transient

The entity has just been instantiated and is not associated with a persistence context. It has no persistent representation in the database and typically no identifier value has been assigned (unless the assigned generator was used).

1.1.2 Managed or Persistent

The entity has an associated identifier and is associated with a persistence context. It may or may not physically exist in the database yet.

1.1.3 Detached

The entity has an associated identifier but is no longer associated with a persistence context (usually because the persistence context was closed or the instance was evicted from the context).

1.1.4 Removed

The entity has an associated identifier and is associated with a persistence context, however, it is scheduled for removal from the database.

2 Simple tutorial

Before starting there are many files to be configured.

2.1 Setting up dependencies

In the POM file (*pom.xml*), some dependencies must be added as shown in the following code:

```
<dependencies>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.4.4.Final</version>
    </dependency>

    <dependency>
        <groupId>org.hibernate.javax.persistence</groupId>
        <artifactId>hibernate-jpa-2.1-api</artifactId>
        <version>1.0.2.Final</version>
    </dependency>
</dependencies>
```

2.2 Setting up persistence unit

The persistence unit is described by the *persistence.xml* file in the *META-INF* directory and it is a set of entities which are logically connected. The entities will be grouped via a persistence unit.
The persistence unit must be inserted in the project path *src/main/resources/META-INF/persistence.xml* and inside this file it is possible to set some parameters:

- the name of the persistence unit (the same name as the database name)
- the URL of database server
- the server credentials (username and password)
- the driver used to communicate with the server (jdbc Driver in the example)
- some optional parameters (in the example they are related to Hibernate)

Here there is an example of persistence.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
↪   xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit name="SocialNetwork">

        <properties>
            <property name="javax.persistence.jdbc.url"
            ↪   value="jdbc:mysql://localhost:3306/SocialNetwork?
            ↪   serverTimezone=Europe/Rome" />
            <property name="javax.persistence.jdbc.user" value="root"
            ↪   />
            <property name="javax.persistence.jdbc.password"
            ↪   value="root" />
            <property name="javax.persistence.jdbc.driver"
            ↪   value="com.mysql.cj.jdbc.Driver" />
            <property name="hibernate.hbm2ddl.auto" value="update"/>
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.format_sql" value="true" />
            <property name="hibernate.connection.pool_size"
            ↪   value="100"/>
        </properties>
    </persistence-unit>
</persistence>
```

2.3 Create Entity class

The database tables are mapped by Entity classes and they exploit annotations to map the class members with the table columns.
The most used annotations are:

- **@Id**: Identifies the unique ID of the database entry
- **@GeneratedValue**: Together with an ID this annotation defines that this value is generated automatically
- **@Transient**: Field will not be saved in database

- **@Table(name = ”tableName”)**: allows to define the name of the table associated to the class, if it is different from the class name
- **@Column(name=”columnName”, length=50, nullable=false, unique=true)**: allows to specify the name of the table column associated to the class member with some optional parameters

An example of entity is shown in the following code:

```
@Entity
@Table(name = "Person")
public class Person{
  @Column(name="idPerson")
  @Id
  @GeneratedValue
  private Long id;

  @Column(name="Username", length=50, nullable=false, unique=true)
  private String firstName;
  @Column(name="Surname", length=50, nullable=false, unique=true)
  private String lastName;

}
```

2.4 CRUD operations

For the sake of simplicity, it is presented here a class which implements all the CRUD operations (create/insert, read, update, delete) through custom methods.
It is necessary also to add two more methods that allow to create the EntityManagerFactory instance and to close it at the end of the operations.

```
public class DBManager{
  private EntityManagerFactory emf;
  private EntityManager em;

  public DBManager() {
    emf = Persistence.createEntityManagerFactory("SocialNetwork");
  }
```

```java
@Override
public boolean create(Person person) {
  boolean ret = false;
  try{
      em = emf.createEntityManager();
      em.getTransaction().begin();
      em.persist(person);
      em.getTransaction().commit();
      ret = true;
  }
  catch(Exception e){
      //handle exception
      ret = false;
  }
  finally{
      em.close();
  }
  return ret;
}

@Override
public Person read(Long id) {
  Person person = null;
  try{
      em = emf.createEntityManager();
      em.getTransaction().begin();
      person = em.find(Person.class, id);
      em.getTransaction().commit();
  }
  catch(Exception e){
      //handle exception
      person = null;
  }
  finally{
      em.close();
  }
  return person;
}
```

```
@Override
public boolean update(Person person){
  boolean ret = false;
  try{
      em = emf.createEntityManager();
      em.getTransaction().begin();
      person = em.merge(person);
      em.getTransaction().commit();
      ret = true;
  }
  catch(Exception e){
      //handle exception
      ret = false;
   }
  finally{
      em.close();
  }
  return ret;
}

@Override
public boolean delete(Long idPerson) {
  boolean ret = false;
  try{
      em = emf.createEntityManager();
      em.getTransaction().begin();
      Person person = em.getReference(Person.class, idPerson);
      em.remove(person);
      em.getTransaction().commit();
      ret = true;
  }
  catch(Exception e){
      //handle exception
      ret = false;
  }
  finally{
      em.close();
  }
  return ret;
}
```

```
  @Override
  public void close() {
    emf.close();
  }
}
```

Example of some operations executed with DBManager class methods:

```
//MainClass.java
public static void main(String[] args){
    DBManager db = new DBManager();
    //...
    Person person = new Person("Marco", "Rossi");
    db.create(person);
    person.setName("Mario");
    db.update(person);
    // Suppose there is a person with idPerson = 3
    Person newPerson = db.read(3);
    db.delete(3);
    //...
    db.close();
    //...
}
```

Before starting a new operation it is necessary to begin a new transaction. Entities which are managed by an Entity Manager will automatically propagate the changes to the database (if this happens within a commit statement).
If the Entity Manager is closed (via close()) then the managed entities are in a detached state. In order to synchronize them again with the database, the Entity Manager provides the merge() method.

3 Advanced features

3.1 Complex query

With JPA, in particular with Hibernate implementation, it is possible to execute more complex queries. Let's assume that we want to delete people with a specific couple name and surname:

```java
public boolean advancedDelete(String name, String Surname){

    // Suppose em private member of the class
    boolean ret = false;
    try{
        Query q = em.createQuery("SELECT p FROM Person p WHERE
         ↪  p.firstName = :firstName AND p.lastName = :lastName");
        q.setParameter("firstName", name);
        q.setParameter("lastName", surname);
        Person person = (Person) q.getSingleResult();
        em.remove(person);
        em.getTransaction().commit();
        ret = true;
    }
    catch(Exception e){
        //handle exception
    }
    finally{
        em.close();
    }
    return ret;
}
```

As can be seen from the code, it is enough to write the query and specify the custom parameters to be used.

3.2 Relationship

JPA allows to define relationships between classes, e.g. it can be defined that a class is part of another class (containment). Possible relationship annotations:

- **@OneToOne**
- **@OneToMany**

- **@ManyToOne**
- **@ManyToMany**

A relationship can be bidirectional or unidirectional, e.g. in an unidirectional case only one class has a reference to the other class, instead in a bidirectional relationship both classes store a reference to each other. Within a bidirectional relationship you need to specify the owning side of this relationship in the other class, with the attribute "mappedBy", e.g.

```
    @ManyToMany(mappedBy="attributeOfTheOwningClass")
```

3.2.1 One to one relation

Take for example two entities, Person and Address, which are linked by a one-to-one relationship

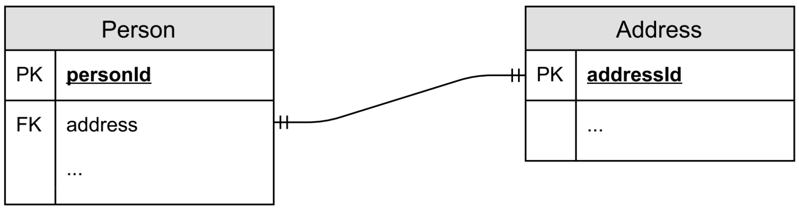

Figure 3: OneToOne Relationship

```
public class Person{
    @Id
    private Long idPerson;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "address", referencedColumnName = "idAddress")
    private Address address;
    //...
}
```

We place the *@OneToOne* annotation on the related entity field which is Address. Also, we need to place the *@JoinColumn* annotation to configure the column name in the Person table in order to map the primary key in the Address table.

Note that in the next entity we do not use the *@JoinColumn* annotation. This is because we only need it on the owning side of the foreign key relationship:

```java
public class Address {
    @Id
    private Long idAddress;

    @OneToOne(mappedBy = "address")
    private Person person;
    //...
}
```

From now on, the two entities are connected by a relationship *@OneToOne*, reflecting the one present in the database.

3.2.2 Many to One and One to Many relations

Take for example two entities, Person and Post, such that one Person can insert more than one Post, but a Post is inserted by only one Person, so we have on the Person side a relationship *@OneToMany* and on the Post side *@ManyToOne*.

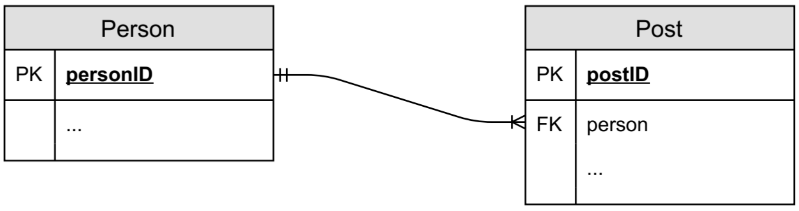

Figure 4: ManyToOne - OneToMany Relationship

```java
public class Person{
    @Id
    private Long idPerson;
    @OneToMany(mappedBy = "person", cascade = CascadeType.ALL)
    private List<Post> posts;
    //...
}
```

The *mappedBy = "person"* attribute indicates the field that owns the relationship, all operations should be cascaded automatically to entity objects that are referenced by the *"person"* field thanks to *cascade = CascadeType.ALL*.

```
public class Post{
    @Id
    private Long idPost;

    @ManyToOne
    @JoinColumn(name="person")
    private Person person;
    //...
}
```

We use *@JoinColumn* to specify the column that holds the foreign key. The owner of the relationship is therefore the entity Post, *@ManyToOne* annotation is always associated with the owner of a relationship and therefore does not allow *mappedBy* attribute.

3.2.3 Many to many relation

Let's suppose that a Person can be associated to more groups and a Group can have more people associated. This is a clear example of *@ManyToMany* relation and in the following example we will assume that the owner is a Person entity.
Since both sides should be able to reference the other, we need to create a separate table to hold the foreign keys and we call it *groupMembership*.

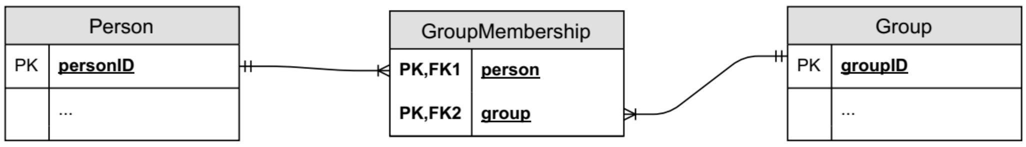

Figure 5: ManyToMany Relationship

```
public class Person{
    @Id
    private Long idPerson;

    @ManyToMany
    @JoinTable(
      name = "groupMembership",
      joinColumns = @JoinColumn(name = "person"),
      inverseJoinColumns = @JoinColumn(name = "group"))
    Set<Group> groups;
    //...
}
```

```
public class Group{
    @Id
    private Long idGroup;

    @ManyToMany
    Set<Person> persons;
    //...
}
```

On the owner side we configure the relationship and, just as already said, in this example is Person; we can do this with the *@JoinTable* annotation in the Person class.
We provide the name of the join table (*groupMembership*), and the foreign keys with the *@JoinColumn* annotations. The *joinColumns* attribute will connect to the owner side of the relationship, and the *inverseJoinColumns* to the other side.

References

[1] URL: https://docs.jboss.org/hibernate/orm/5.4/userguide/html%5C_single/Hibernate%5C_User%5C_Guide.html.

[2] URL: https://examples.javacodegeeks.com/enterprise-java/jpa-one-many-example/.

[3] URL: http://www.ross.net/crc/download/crc_v3.txt.

[4] URL: https://www.html.it/articoli/jpa-2-e-la-persistenza-in-java/.

[5] URL: https://en.wikibooks.org/wiki/Java%5C_Persistence/Persisting/.

[6] *Figure 1.* URL: https://goldenpackagebyanuj.blogspot.com/2010/05/hibernet-dialect-properties.html.

[7] *Figure 2.* URL: https://vladmihalcea.com/jpa-persist-merge-hibernate-save-update-saveorupdate.

[8] *Java Persistence with Hibernate, Second Edition.*