Exercise - JPA Entity Mappings – 2

Relationship Mapping

Object oriented programming is to a large degree about creating classes which relate to each other. If a class cannot solve a problem by itself, it delegates the job to another class.

Relations between classes have **cardinality**, and can be either **bi**- or *unidirectional*.

There are 8 different combinations of Cardinality and Direction, and they can all easily be implemented in JPA with some nice NetBeans-assistance.

Cardinality	Direction
One-to-one	Unidirectional
One-to-one	Bidirectional
One-to-many	Unidirectional
Many-to-one/one-to-many	Bidirectional
Many-to-one	Unidirectional
Many-to-many	Unidirectional
Many-to-many	Bidirectional

In the following, we will test four of these combinations

Reference: http://www.tutorialspoint.com/jpa/jpa_entity_relationships.htm

This exercise draws on the Entity Class Customer generated in JPA-day1. So you should either continue with this project or create a new database and project, and include the Customer Entity as explained yesterday.

1) Add a new *Entity* Class **Address**

Add the following fields + relevant constructors (remember always to have a no-argument constructor):

- `private String street;`
- `private String city;`

One to One – Unidirectional

2) If you have started a new project, create an Entity Class `Customer`, and add a *name* field (String) + relevant constructors and getter/setters, else just continue with the project used day-1.

3) Provider the Customer with an Address field:

```
private Address address;
```

Add the cursor on the field and press ALT + ENTER → Select Create *unidirectional one to one relationship*

Make sure you understand the "things" changed in the Customer and (if any) the Address class.

Regenerate the schema and investigate the generated tables (observe the location of the *foreign key*)

One to One – Bidirectional

4) Now remove the @OneToOne annotation and create a bidirectional *one to one* relationship *(*again, right-click, and select the option you want).

Make sure you understand what is meant by bidirectional, before you continue (how would you show bidirectional using UML?)

Creating a bidirectional relationship will obviously require you to add a reference in Address, pointing back to Customer. Provide a name when requested by the wizard (customer) and select the default for the owing side[1].

- Go to the Address class. Investigate and understand the generated code.
- Run the project and investigate the generated tables (the foreign key). Is there any difference compared to the previous exercise. If not explain why.

http://www.tutorialspoint.com/jpa/jpa_entity_relationships.htmOneToMany (unidirectional)

4) Remove the generated code in both classes and use the wizard one more time, this time to generate a *one to Many* relationship. You obviously can't do that with your current Address field since this can hold only one instance, so change it into:

```
private List<Address> addresses = new ArrayList();
```

Now, a Customer can have more than one address. If you feel the opposite makes more sense; an address can have more Customers (i.e two customers are married, and live together), just do that instead.

Now, use the wizard to generate a *One to Many Unidirectional relationship*.

- Observe the generated code.
- Run the project and investigate the generated tables. Make sure to press Refresh so see all tables.
    - How many tables were generated? Explain the purpose of each of the tables.
- If you (as me) don't like the number of generated tables generated by this strategy, you can use the `@JoinColumn` annotation to implement the relation using a foreign key.
- Create a "test" method and insert a number of Customers with Addresses into the tables, using JPA.

OneToMany (bidirectional)

4) Remove the generated code in both classes and comment out your test code.

Now, use the wizard to generate a *One to Many Bidirectional relationship*. Make sure you understand the suggestions given by the wizard, before you accept.

- Observe the generated code, especially where we find the *mappedBy* value. Explain.
- Run the project and investigate the generated tables (the foreign key).

[1] *The owning side of the relation is the side of the relation that* owns *the foreign key in the database*

- Create a "test" method and insert a number of Customers with Addresses into the tables, using JPA. Which extra step is required for this strategy compared to OneToMany unidirectional ?

Many To Many (bidirectional)

Finally let's implement a ManyToMany relationship between the Customer and Address class. That is: a customer can have many addresses, and an address can "have" many Customers.

Before you do this, refresh your knowledge from the 1-2 semesters and answer the following questions.

- How can we implement ManyToMany relationships in an OO-language (like Java)?
- How can we implement ManyToMany relationships in a Relational Database?

5) One more time, remove the generated code in both classes

Right click the addresses list and select create *bidirectional Many to Many Relationship* (observe; both sides can be the owing side)

- Observe the generated code and make sure you understand every line generated in BOTH classes.
- Run the project and investigate the generated tables. Explain ALL generated tables.
- Create a "test" method and insert a number of Customers and Addresses. Make sure to test both the scenario where a customer can have more than one address, and an Address can belong to more than one customer.

6)

Create a "façade" class *CustomerFacade*, providing the following methods:

```
1. Customer getCustomer(int id);
2. List<Customer> getCustomers();
3. Customer addCustomer(Customer cust);
4. Customer deleteCustomer(int id);
5. Customer editCustomer(Customer cust);
```

If not already done, provide the Customer Class with the following methods:

- `List<Address> getAddresses();`
- `Void addAddress(Address address);`

Provide the `Address` class with the similar methods (for Customers)

Hints:

Structure of Façade Class) →Use this template for the Factory class.

```
public class CustomerFacade {
  EntityManagerFactory emf;

  public CustomerFacade(EntityManagerFactory emf) {
    this.emf = emf;
  }
  EntityManager getEntityManager(){
```

```
        return emf.createEntityManager();
    }
}
```

Structure of each method that uses the EntityManager → Use the pattern given below:

```
EntityManager em = getEntityManager();
try{
  // Use the entity manager
}finally{
  em.close();
}
```

2) → Google JPA + SELECT

3) →In order to persist both a Customer and his Addresses you can:

- Use the cascade property of the @ManyToMany annotation
- Persist the customer, and persist the addresses in the Customers address list.