Exercise - JPA Entity Mappings -1

These exercises will investigate a great deal of the most useful Object-Entity mappings available in JPA. You should however, be aware of that the amount possible annotations are enormous. For this exercise we will create the tables from the Entity classes, so we know exactly what we do. At the end however, you should create a new project and its entity classes from the tables generated by this exercise. This will show you a great deal of those annotations we haven't used (because we are using default values)

1) Create a new plain (Maven) java application, and add the packages: **entity**, **test** and **enums**

In the **test** package, add a class `Tester` including a main method. Create a test method in this class from which we will do all interactions with the Entity Manager.

2) Create a local MySQL database

3a) Create an **Entity** class, Book with only one field (apart from the id) *title*. Use `GenerationType.IDENTITY` to let the database generate id's automatically (using AUTO_INCREMENT).

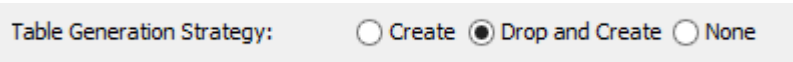
3b) Include the MySQL JDBC Driver to the project.

Right Click dependencies → Add Deependency → Type "mysql-connector-java" in the Query Text Field → Select the newest version of the Driver).

Open the pom-file to see how the dependency was added to the file.

3c) in your test method, create an `EntityManager` (use the slides if you need hints)

For all of these exercises, you should (initially) select the Drop and Create strategy in your persistence.xml file:



3d) In the test package, add a new class SchemaBuilder and provide the class with a `main(..)` method.

Add this line to the method: `Persistence.generateSchema("NAME_OF YOUR_PU", null);`

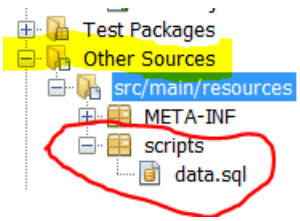
You can recreate your database whenever you like, by running this method.

Auto generation of Id's and MySQL's AUTO_INCREMENT

4) Add a few books to the project and verify that we can find them using the entity manager.

Verify that a matching table has been created, and populated with data.

Now, add a new folder **scripts** to "src/main/resources" as sketched to the right, and add a file **data.sql** to the folder.



Use plain SQL to insert a number of Books in the file as sketched below[1]:

```
INSERT INTO BOOK (ID, TITLE) VALUES (null, 'book-3');
SET @book1 = LAST_INSERT_ID();
INSERT INTO BOOK (ID, TITLE) VALUES (null, 'book-4');
SET @book2 = LAST_INSERT_ID();
```

Add the following line to your persistence.xml file:

```
<property name="javax.persistence.sql-load-script-source" value="scripts/data.sql"/>
```

Re-run the project and observe/explain the id's found in the Book table.

5)

Add a new entity class Customer, initially only with a *firstName* and *lastName* property and using the GenerationType.Identity strategy.

Add the necessary SQL to the addData.sql script to add a few customers.

Re-run the project and verify that you can find the new customers.

Enums

5) Add the following enum to the project:

```
public enum CustomerType {
  GOLD,
  SILVER,
  IRON,
  RUSTY
}
```

- Provide the Customer class with a `CustomerType` field, + a getter and setter.
- In your test code add a `CustomerType` to your test Customers.
- Run and explain the column and values in the Customer table.
- Add this annotation `@Enumerated(EnumType.STRING)` on top of the CustomerType field
- Run and explain the column and values in the Customer table.

Collections of basic types

6) Provide the customer with a list of hobbies as sketched below:

```
Private List<String> hobbies = new ArrayList();
```

And the methods: `addHobby(String s)` and `String getHobbies()` (return a comma separated list with all hobbies)

Test and verify how the list is stored by the Customer table

Do you like what you see?

If not, add the following annotation to the hobbies List (do it anyway ;-)

[1] In this example we don't use the @bookn value, but you could use it later in the script, as the value for a foreign key

```
@ElementCollection()
```

Regenerate (run the project) the tables and observe the result. I assume you agree, this looks at lot better ;-)

Maps of Basic Types

7) Add a map to your Customer class as sketched below:

```
private Map<String,String> phones = new HashMap();
```

Add a method: `addPhone(String phoneNo, String description){..}`

Add a method: `getPhoneDescription(String phoneNo){..}`

Add a few phone numbers to your customer in the Tester class, and execute (which should regenerate the tables).

Bloooob, do you like what you see?

If not, add the following annotations to the map:

```
@ElementCollection(fetch = FetchType.LAZY)
@MapKeyColumn(name = "PHONE")
@Column(name="Description")
```

Execute and observe the generated columns and values. Make sure you understand the purpose of each of the annotations