Lab: Workshop – MiniORM Part 2

This document defines the lab assignments for the ["Spring Data" course @ Software University](https://softuni.bg).

Since you have come to this point it is considered that you have fully finished Part 1 of our Mini ORM. By following the guidelines of this document, you will add more functionality to the framework like create, delete and so on.

One of the things our framework cannot do at this moment is to create a table. In big applications, schema manipulation is done with a default configuration of the framework, **depending on our development**. What does that mean?

When we have our app **running** and users actively **using** it, we rarely (almost never) want to lose or drop our data. Thus, why do complex ORMs like Hibernate or Entity, look for **changes in the models** (e.g. **`User`**) constantly and if set **update the structure** of the tables.

1. Create Table

Creation of the tables will be done by the **`EntityManager`** object that we have already implemented. It should be able to parse the fields in our models to table columns – set appropriate data type, name (from **`Column`** annotation) and constraints.

Begin by creating such method **`doCreate(Class<E> entityClass)`**:

```java
@Override
public void doCreate(Class<E> entityClass) throws SQLException {
    String tableName = getTableNameFromEntity(entityClass);
    String query = String.format("CREATE TABLE %s ( id INT PRIMARY KEY AUTO_INCREMENT, %s);",
            tableName, getAllFieldsAndDataTypes(entityClass));

    PreparedStatement statement = connection
            .prepareStatement(query);

    statement.execute();
}
```

The method has several simple steps that have to perform:

- Get the table name that should be created.
- Start building a query.
- **Scan the model's declared fields** and **map** them to annotated names and **MySQL data types.**

```java
private String getAllFieldsAndDataTypes(Class<E> entityClass) {
    StringBuilder result = new StringBuilder();
    Field[] fields = Arrays.stream(entityClass.getDeclaredFields())
            .filter(field -> field.isAnnotationPresent(Column.class))
            .toArray(Field[]::new);

    Arrays.stream(fields).forEach(field -> {
      result.append(getNameAndDateTypeOfField(field));
    });
    return  result.toString();
 }
```

Data types mapping is very similar to the **fillField** method.

Hints

For example, java **int** or **Integer** type is mapped to the "**INT**" SQL type, String to "**VARCHAR(…)**" and so on.

Consider when you should check the need for table creation. Sometimes we might need a simple alter and just add fields to already existing tables. If we drop and create a new table we might lose important information about already existing users, no matter if the new fields are left blank after altering.

2. Test Create

Create a new custom entity and try to persist it with the **EntityManager.**

3. Alter Table

In most cases when our application is running, we need to just **ALTER** the tables and add new model fields.

A new private method is needed:

```
@Override
public void doAlter(E entity) throws SQLException {
    String tableName = getTableNameFromEntity(entity);
    String query = String.format("ALTER TABLE %s ADD COLUMN %s;",
            tableName, getNewFields(entity.getClass()));

     connection.prepareStatement(query).executeUpdate();
}
```

And a helper one to let us know if the field that we will be trying to insert doesn't exist already:

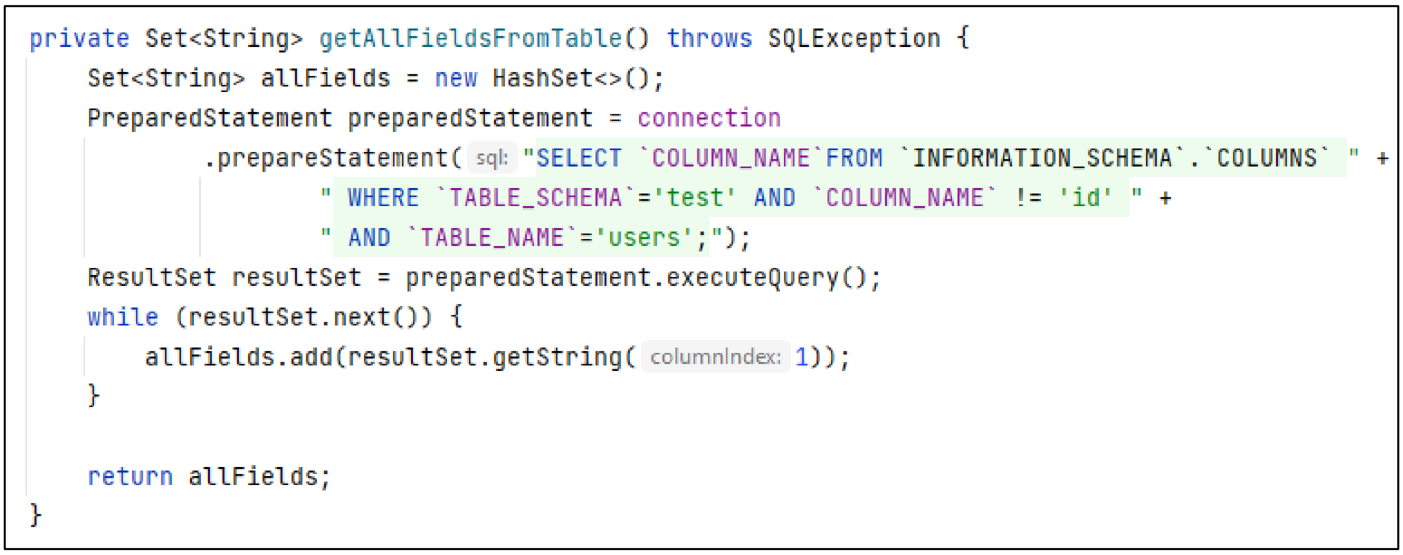
```
private String getNewFields(Class<?> entityClass) throws SQLException {
    StringBuilder result = new StringBuilder();
    Set<String> fields = getAllFieldsFromTable();

    Arrays.stream(entityClass
            .getDeclaredFields())
            .filter(field -> field.isAnnotationPresent(Column.class))
            .forEach(field -> {
                String fieldName = field.getName();
                if (!fields.contains(fieldName)) {
                    result.append(getNameAndDateTypeOfField(field));
                }
            });
    return result.toString();
}
```

Information about the columns of a table can be retrieved from the **information_schema** of the database server. It is an object where each MySQL instance stores information about all the other databases that the MySQL server maintains. It is also referred to as the data dictionary and system catalog.

Hints

- Get information about existing tables and their columns from **`information_schema`.**
- Use the **`ResultSet`** class to check if retrieved fields by **COLUMN_NAME.**



```
private Set<String> getAllFieldsFromTable() throws SQLException {
    Set<String> allFields = new HashSet<>();
    PreparedStatement preparedStatement = connection
            .prepareStatement( sql: "SELECT `COLUMN_NAME`FROM `INFORMATION_SCHEMA`.`COLUMNS` " +
                    " WHERE `TABLE_SCHEMA`='test' AND `COLUMN_NAME` != 'id' " +
                    " AND `TABLE_NAME`='users';");
    ResultSet resultSet = preparedStatement.executeQuery();
    while (resultSet.next()) {
        allFields.add(resultSet.getString( columnIndex: 1));
    }

    return allFields;
}
```

4. Test Alter

Add new fields to already existing entity class (you can use the one made from 3.) and try to persist a new object.

5. Delete

Try implementing the last CRUD operation – the delete.

Hints

Create a method that receives a database column and deletes criteria as parameters, very similar to the find methods and delete records matching given criteria.