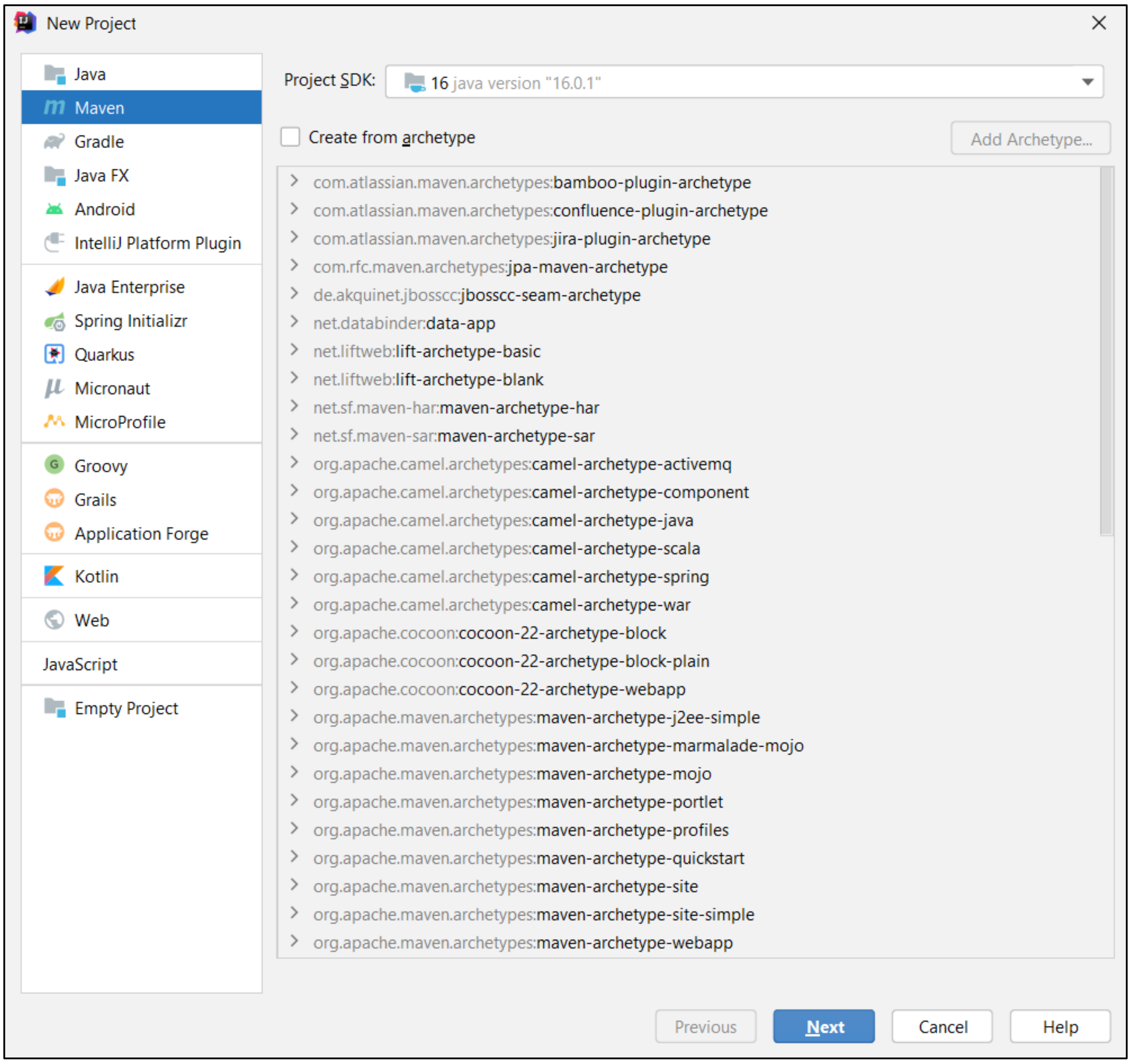
Lab: Workshop – MiniORM Part 1

This document defines the lab assignments for the "Spring Data" course at Software University.

By following the guides of this document you will be able to create your custom ORM with basic functionality (insert, update and retrieve a single or set of objects). It will have options to work with already created tables in a database or create new tables if such is not present yet.

1. Setup

Create a new **Maven** project in **IntelliJ**.



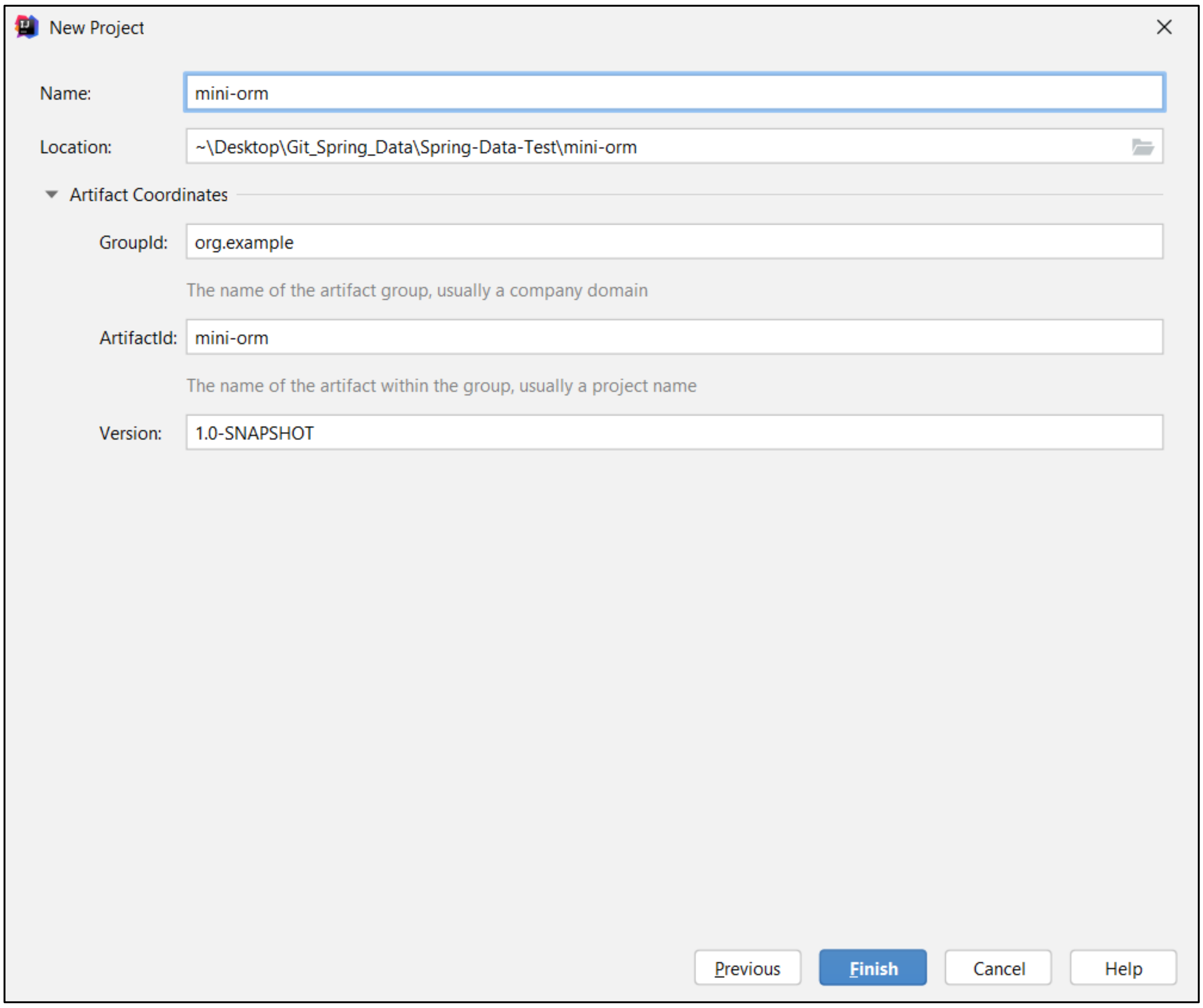
Project SDK: 16 java version "16.0.1"

Create from archetype

com.atlassian.maven.archetypes:bamboo-plugin-archetype
com.atlassian.maven.archetypes:confluence-plugin-archetype
com.atlassian.maven.archetypes:jira-plugin-archetype
com.rfc.maven.archetypes:jpa-maven-archetype
de.akquinet.jbosscc:jbosscc-seam-archetype
net.databinder:data-app
net.liftweb:lift-archetype-basic
net.liftweb:lift-archetype-blank
net.sf.maven-har:maven-archetype-har
net.sf.maven-sar:maven-archetype-sar
org.apache.camel.archetypes:camel-archetype-activemq
org.apache.camel.archetypes:camel-archetype-component
org.apache.camel.archetypes:camel-archetype-java
org.apache.camel.archetypes:camel-archetype-scala
org.apache.camel.archetypes:camel-archetype-spring
org.apache.camel.archetypes:camel-archetype-war
org.apache.cocoon:cocoon-22-archetype-block
org.apache.cocoon:cocoon-22-archetype-block-plain
org.apache.cocoon:cocoon-22-archetype-webapp
org.apache.maven.archetypes:maven-archetype-j2ee-simple
org.apache.maven.archetypes:maven-archetype-marmalade-mojo
org.apache.maven.archetypes:maven-archetype-mojo
org.apache.maven.archetypes:maven-archetype-portlet
org.apache.maven.archetypes:maven-archetype-profiles
org.apache.maven.archetypes:maven-archetype-quickstart
org.apache.maven.archetypes:maven-archetype-site
org.apache.maven.archetypes:maven-archetype-site-simple
org.apache.maven.archetypes:maven-archetype-webapp

Maven is a software development tool that helps you manage your projects's build, setting it up project and its dependencies.

Fill in the fields as shown in the screenshot below and click "**Next**".


New Project

Name: mini-orm

Location: ~\Desktop\Git_Spring_Data\Spring-Data-Test\mini-orm

Artifact Coordinates

GroupId: org.example

The name of the artifact group, usually a company domain

ArtifactId: mini-orm

The name of the artifact within the group, usually a project name

Version: 1.0-SNAPSHOT

Select desired project location and click "**Next**" again.

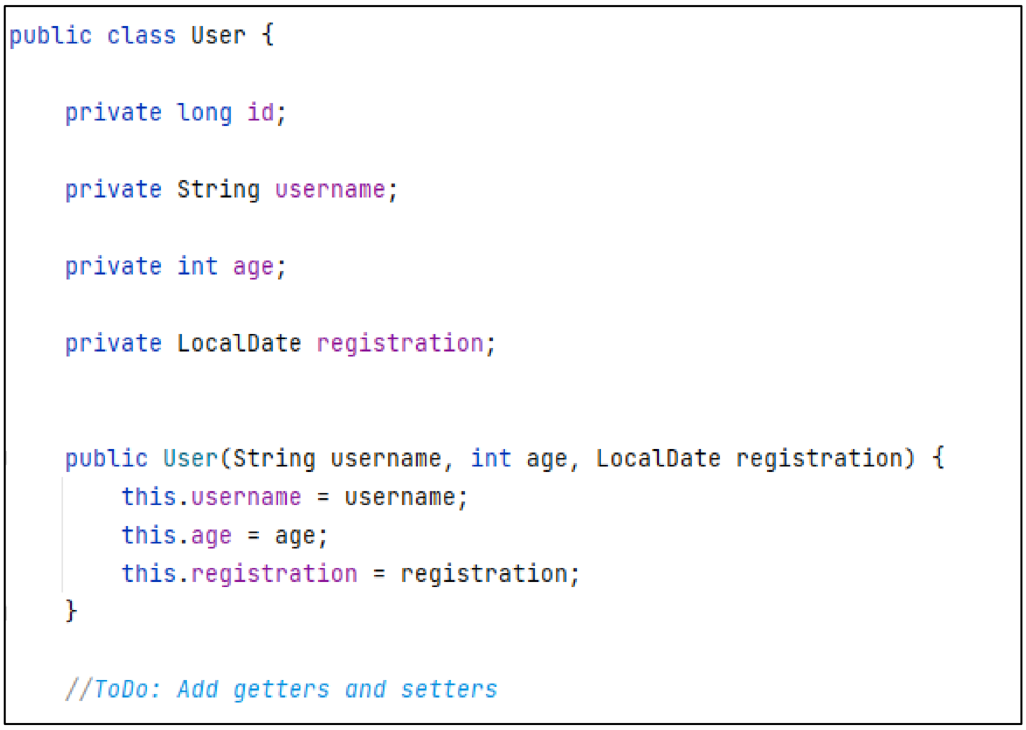
Initially, no database driver is imported. Do that using Maven by adding a dependency in the "**pom.xml**" file.

```
<dependencies>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.25</version>
    </dependency>
</dependencies>
```

2. Create Entities

In the means of ORMs, database objects are mapped to object-oriented implementations called "**entities**". They are objects that analogically to database tables, e.g. users, hold fields containing the user's main characteristics – **`id`**, **`username`**, **age**, etc.

In the "java" folder create a package called **entities** where will every one of our entities be. Now let's **create a class User** with fields and properties (**`id`, `username`, `password`, `age`, `registrationDate`**). Create a **constructor** that **sets all fields except id – it's auto incremented on the database level**. The order of the parameters in the constructor must be **the same as the sequence of columns in the table in the database.** Add Getters and Setters for all fields.



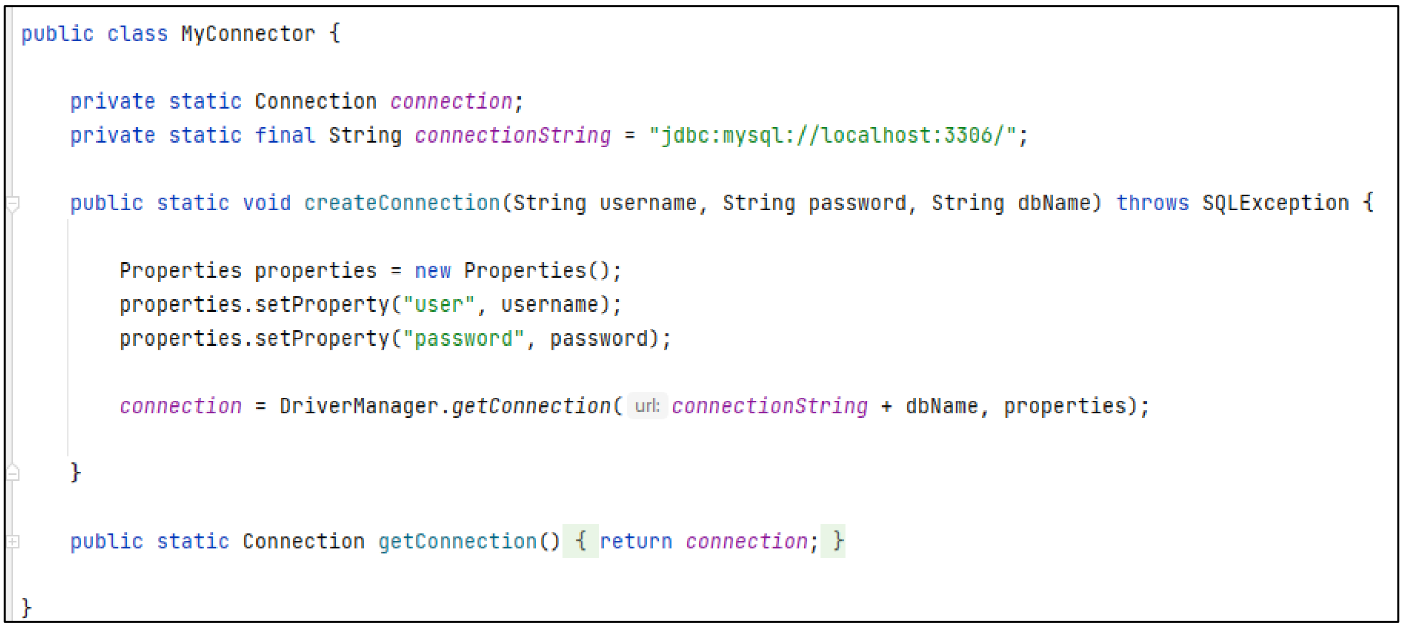
```java
public class User {

    private long id;

    private String username;

    private int age;

    private LocalDate registration;


    public User(String username, int age, LocalDate registration) {
        this.username = username;
        this.age = age;
        this.registration = registration;
    }

    //ToDo: Add getters and setters
```

3. Create Database Connection

Unlike all previous tasks, it's time to start separating our logic into classes.

Create a new package "**orm**" and **`class` MyConnector** in it that generates a connection with our database. To achieve this, we would require the following parameters:

- **`Username`** – database username
- **`Password`** – database password
- **`Database Name`** – the current database for the project. We need to create one manually.



```java
public class MyConnector {

    private static Connection connection;
    private static final String connectionString = "jdbc:mysql://localhost:3306/";

    public static void createConnection(String username, String password, String dbName) throws SQLException {

        Properties properties = new Properties();
        properties.setProperty("user", username);
        properties.setProperty("password", password);

        connection = DriverManager.getConnection( url: connectionString + dbName, properties);

    }

    public static Connection getConnection() { return connection; }

}
```

4. Create Database Context

It's time to create an **`interface`** that will define the operations we can perform with the database. Name your interface **`DbContext`** and defined the following methods in it.

- **`boolean persist(E entity)`** – it will insert or update the entity depending if it is attached to the context
- **`Iterable<E> find(Class<E> table)`** – returns collection of all entity objects of type **E**
- **`Iterable<E> find(Class<E> table, String where)`** – returns collection of all entity objects of type **T** matching the criteria given in "**`where`**"
- **`E findFirst(Class<E> table)`** – returns the first entity object of type **E**
- **`E findFirst(Class<E> table, String where)`** – returns the first entity object of type **E** matching the criteria given in "**`where`**"

```
public interface DbContext<E> {
    boolean persist(E entity);

    Iterable<E> find(Class<E> table);

    Iterable<E> find(Class<E> table, String where);

    E findFirst(Class<E> table);

    E findFirst(Class<E> table, String where);
}
```

5. Create Entity Manager

Enough with the preparation it's time to write the core of our Mini ORM. That would be the **`EntityManager`** class that will be responsible for inserting, updating and retrieving entity objects. That class would implement methods of the **`DbContext interface`**. It will require a **`Connection`** object that would be initialized with a given connection string.

```
public class EntityManager<E> implements DbContext<E>{
    private Connection connection;

    public EntityManager(Connection connection){
        this.connection = connection;
    }

    //Implement methods
```

6. Persist Object in the Database

The logic behind the **persist** method is pretty simple. First, the method should **check** if the given **object** to be persisted **is not contained** in the database and if not **add it**, otherwise **update its properties with the new values**. The method returns whether the object was **successfully persisted** in the database or not.

But how can we check if the user that we are trying to persist is new to the database or have to update it? We can do that by checking the value of its **`id`** field and if it's not empty that means we're trying to insert it. But the method works with a generic type – **E** and we don't have direct access to its getter methods – for example **`getId`**.

In order to minimize concrete and work with other entities in the future(not only **`User`**) we have to access the field some other way. One thing that can help us is **`Annotations`**.

Create 3 annotations **`Entity`**, **`Column`** and **`Id.`**

```java
@Retention(RetentionPolicy.RUNTIME)
public @interface Id {

}
```

```java
@Retention(RetentionPolicy.RUNTIME)
public @interface Entity {
    String name();
}
```

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Column {
    String name();
}
```

Annotate entities and their corresponding fields. In the **`Entity`** annotation specify the name of the database table you want to be mapped – **users** and in **`Column`** – the corresponding table column name to the java field.

We are going to need a new additional method **`getId(Class entity)`** in the **`EntityManager`** class. It will return the **`id`** field via reflection:

```java
private Field getId(Class<?> entity) {
    return Arrays.stream(entity.getDeclaredFields())
            .filter(x -> x.isAnnotationPresent(Id.class))
            .findFirst()
            .orElseThrow(() -> new UnsupportedOperationException("Entity does not have primary key"));
}
```

And **`persist`** Method:

```java
public boolean persist(E entity) throws IllegalAccessException, SQLException {
    Field primaryKey = getId(entity.getClass());
    primaryKey.setAccessible(true);
    Object value = primaryKey.get(entity);

    if (value == null || (long) value <= 0) {
        return doInsert(entity, primaryKey);
    }

    return doUpdate(entity, primaryKey);
}
```

If the returned value is null we need to do an insert, otherwise update. So far the **`persist`** method should look like this:

We need to implement 2 more methods:

- **`private boolean doInsert(E entity, Field primary)`**
- **`private boolean doUpdate(E entity, Field primary)`**

Both methods would prepare query statements and execute them.

The difference between them is when you insert a new entity you should **set its `id`**. The **`Id`** is generated on the database level. Both methods return whether the entity successfully persisted.

Here are some tips for the Insert method:

- Get the table name you will be inserted into
- Start joining the components of your query – **INSERT** clause, table name + fields, **VALUES** and the corresponding values for the insertion
  HINT: Iterate over the entity's fields
- Prepare and execute the statement via the connection

```
private boolean doInsert(E entity, Field primary)
        throws SQLException, IllegalAccessException {
    String tableName = this.getTableName(entity.getClass());
    String query = "INSERT INTO " + this.getTableName(entity.getClass()) + " (";

    //Get fields and values

    return connection.prepareStatement(query).execute();
}
```

And some tips for the update method:

- Get the table name you will be updated into.
- Start joining the components of your query – **UPDATE** clause, table name, **SET**, **WHERE** and the given predicate.
  HINT: Iterate over the entity's fields and add "**`id = {entity's id value}`**" to the **WHERE** clause
- Prepare and execute the statement via the connection.

```
private boolean doUpdate(E entity, Field primary)
        throws IllegalAccessException, SQLException {

    String query = "UPDATE " + this.getTableName(entity.getClass()) + " SET ";

    //Get fields and values
    //Add WHERE clause

    return connection.prepareStatement(query).execute();
}
```

7. Fetching Results

Finally, when we have persisted our entities (objects) in the database let's implement functionality to **get them out of the database and persist them in the operating memory**. We would implement just several methods. That would be **all Find methods from the `DbContext`**.
Here are some tips on how to implement **`public E findFirst(Class<E> table, String where)`** the other ones are similar and they would be on you.

```java
@Override
public E findFirst(Class<E> table, String where) throws SQLException, NoSuchMethodException,
        InstantiationException, IllegalAccessException {
    Statement statement = connection.createStatement();
    String tableName = getTableNameFromEntity(table);

    String query = String.format("SELECT * FROM %s %s LIMIT 1;",
            tableName, where != null ? " WHERE " + where : "");

    ResultSet resultSet = statement.executeQuery(query);
    E entity = table.getDeclaredConstructor().newInstance();

    resultSet.next();
    fillEntity(table, resultSet, entity);

    return entity;
}
```

Here you can see that we used a new method **`fillEntity`**. That method receives a **`ResultSet`** object, **retrieves information from the current row** of the reader and fills in the data. Create two methods:

```java
private void fillEntity(Class<E> table, ResultSet resultSet, E entity) throws SQLException, IllegalAccessException {
    Field[] declaredFields = Arrays.stream(table.getDeclaredFields())
            .toArray(Field[]::new);

    for (Field field : declaredFields) {
        field.setAccessible(true);
        fillField(field, resultSet, entity);
    }
}
```

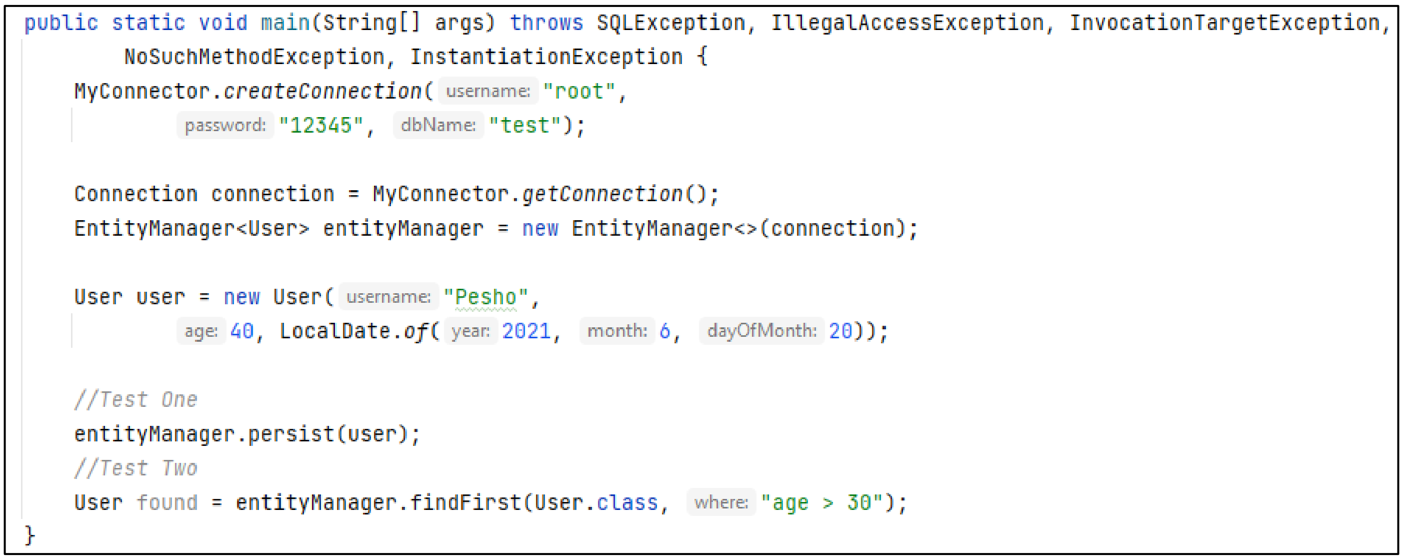
And **`fillField`** method:

```java
private void fillField(Field field, ResultSet resultSet, E entity) throws SQLException, IllegalAccessException {
    field.setAccessible(true);

    if (field.getType() == int.class || field.getType() == long.class) {
        field.set(entity, resultSet.getInt(field.getName()));
    } else if (field.getType() == LocalDate.class) {
        field.set(entity, LocalDate.parse(resultSet.getString(field.getName())));
    } else {
        field.set(entity, resultSet.getString(field.getName()));
    }
}
```

Both methods cooperate closely. **`FillEntity`** calls for **`fillField`** and passes the entity which fields have to be filled, the current field, the **`ResultSet`** object from which we will get information and the field's **`Column`** annotation name, which will give us access to the value in the **`ResultSet`**.

8. Test Framework

If you came to this point you are done with building the **first part** of our MiniORM. Now let's test it to make sure it works as expected. Create several users and persist them in the database. Then update some of the properties of the users (e.g., change password or increase age or some other change). Remember you need to use the persist method to commit changes of the object to the database. Make sure the data is always updated in the database. Here is some example of usage:



```
public static void main(String[] args) throws SQLException, IllegalAccessException, InvocationTargetException,
        NoSuchMethodException, InstantiationException {
    MyConnector.createConnection( username: "root",
            password: "12345",  dbName: "test");

    Connection connection = MyConnector.getConnection();
    EntityManager<User> entityManager = new EntityManager<>(connection);

    User user = new User( username: "Pesho",
            age: 40, LocalDate.of( year: 2021,  month: 6,  dayOfMonth: 20));

    //Test One
    entityManager.persist(user);
    //Test Two
    User found = entityManager.findFirst(User.class,  where: "age > 30");
}
```

9. Fetch Users

Insert several users in the database and **print the usernames and passwords** of those who are **registered after the 2020** year and are **at least 18 years old**.