Lab: Database Apps Introduction
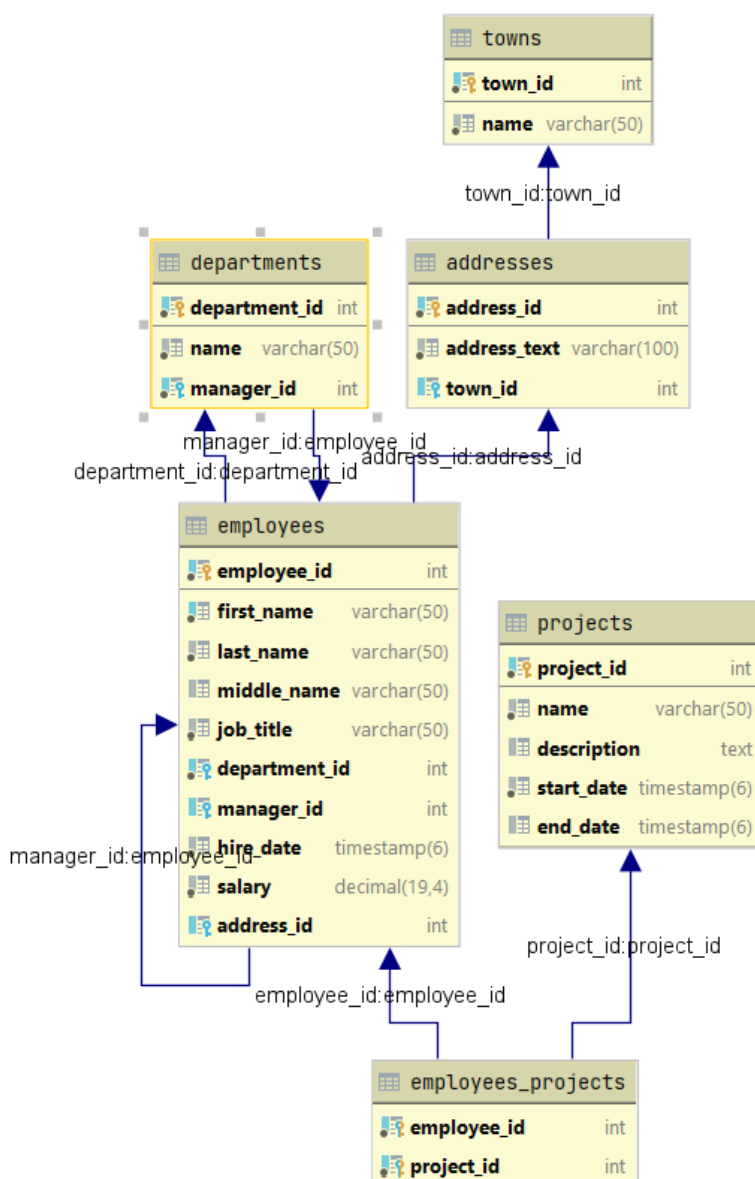
This document defines the lab assignments for the "Spring Data" course at Software University.

Part 0: Inspect SoftUni DB

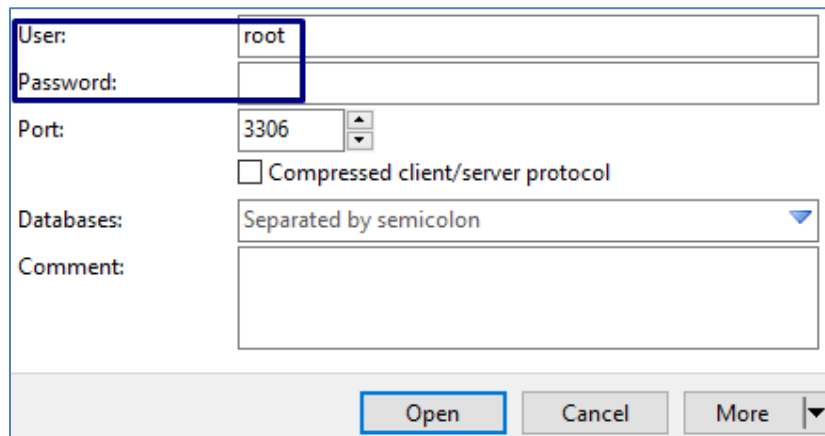## Part 1: Accessing Database via Simple Java Application – Demo

You are given a simple java program that opens a connection to a local database server and retrieves the following data – first name and last name from the "**soft_uni**" database. The data is filtered by salary criteria, which is given by the user at the input. The exercise shows briefly the usage of the **java.sql** package and the **MySQL Connector/J** driver.

Example

Input	Output
70000.00	Ken Sanchez James Hamilton Brian Welcker
80000.00	Ken Sanchez James Hamilton

1. Connection and Connection Properties

Up to this point we have used the HeidiSQL or Workbench GUI to connect to our local instance of MySQL server.

From now on we will achieve this with **Java code** using the **JDBC API**.

The following code shows how a connection is created:

```java
Properties props = new Properties();
props.setProperty("user", user);
props.setProperty("password", password);

Connection connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/soft_uni", props);
```

Lets look closely on what is happening there. A **Properties** class, which holds the username and the password of the user, is being instantiated. The username and the password are given beforehand by the input. They are passed as an argument to the **DriverManager** static class's method **getConnection** via the **props** instance.

The **getConnection** method is a static method, which establishes a connection, for which the **DriverManager** selects the appropriate driver – the **MySQL Connector/J** driver in our case.

The **getConnection** method requires a second argument – **a connection string**. A **connection string** is a string, which holds information about the data source and consists of a database name and other parameters required to establish an initial connection. In our case these parameters are:

- host and port of our local server – **localhost:3306**
- database name – can be diverse, but by the means of our assignment it is **soft_uni**
- user – by default **root**, or any other **specified by the user**
- password – user password for connection if there is one

Remember that the user and the password are already included in the **Properties** instance **props**.

2. Preparing and Executing Statements

SQL execution is done via the **Statement Interface**. To prevent SQL Injection, **PreparedStatement** is used:

```
PreparedStatement stmt =
    connection.prepareStatement("SELECT first_name, last_name FROM employees
WHERE salary > ?");
String salary = sc.nextLine();
stmt.setDouble(1, Double.parseDouble(salary));
```

The user is asked to set the salary criteria, by which we will filter the results.

Then we use the **connection** to prepare a **PreparedStatement** and pass an unfinished SQL query, to which we must append the value for the salary criteria. The question mark is a placeholder for that value. The **setDouble** method accepts two parameters – index of the parameter to be replaced and the value.

If we had done plain concatenation using only **Statement** and appended the value for the **salary** to the query, we would have made our program vulnerable to SQL Injection, because we wouldn't be checking the value. It could be another SQL query, which in the worst case would harm our database. Therefore, we use **PreparedStatement**, **setDouble** and **Double.parseDouble** methods, to insure that the value is as expected – a **double** number.

3. Iterating over the Result

Finally, if data in the database matches the given criteria, we will receive a **ResultSet** of table rows, otherwise it wil be empty.

```
ResultSet rs = stmt.executeQuery();
```

The **ResultSet** object is a table of data. It is not updatable and it can be read only from first to last. Thus it keeps a cursor pointing to the rows of the table. Field values can be accessed via **getter** methods, such as **getString**, which accepts column name:

```
while(rs.next()) {
    System.out.println(rs.getString("first_name") + " " + rs.getString("last_name"));
}
```

Part 2: Writing your own data retrieval application

Now it's time for you to write a similar program. Follow the steps to write a java application that retrieves information about the users, their games and duration. We are going to use the "**diablo**" database from the previous course.

Example

Input	Output
nakov	User: nakov Svetlin Nakov has played 6 games

destroyer	No such user exists

1. Establish a connection to the database

Get the user's **username** and **password** and change the connection string according to the name of the database.

```
Properties props = new Properties();
props.setProperty("user", user);
props.setProperty("password", password);

Connection connection = DriverManager
        .getConnection(DB "jdbc:mysql://localhost:3306/diablo", props);
```

2. Username and statement

Ask the user for a username, which you will use to retrieve the desired info. Write a proper SQL statement to get the result.

```
PreparedStatement stmt =
        connection.prepareStatement(DB "SELECT user_id,user_name,first_name, last_name, count(ug.id) " +
            "AS `games_played` FROM users AS u " +
            "INNER JOIN users_games ug ON ug.'id' " +
            "ON u.id = ug.user_id " +
            "WHERE u.user_name = ? " +
            "GROUP BY ug.user_id");
```

3. Output

Consider that the input may be invalid – **user with given username might not exist** and you will receive an empty **ResultSet**. If so, print "**No such user exists**", otherwise print the **user_name**, **first** and **last names** and the total **count** of games a user has played.

```
if(!rs.next()){
    System.out.println("No such user");
}else{
    System.out.println("user: " + username);
    System.out.println(rs.getString(DB "first_name") + " " + rs.getString(DB "last_name")
        + " has played " + rs.getString(DB "games_played") + " games");
}
connection.close();
```