Using Prepared and Callable Statements

Estimated time: **45 minutes**

Overview

In this lab, you will learn:

- **What PreparedStatement and CallableStatement are in JDBC.**
- **How to securely execute parameterized queries using PreparedStatement.**
- **How to call stored procedures using CallableStatement.**
- **The differences between Statement, PreparedStatement, and CallableStatement.**
- **Best practices for error handling in JDBC.**

By the end of this lab, you will be able to use advanced JDBC features for efficient database interactions.

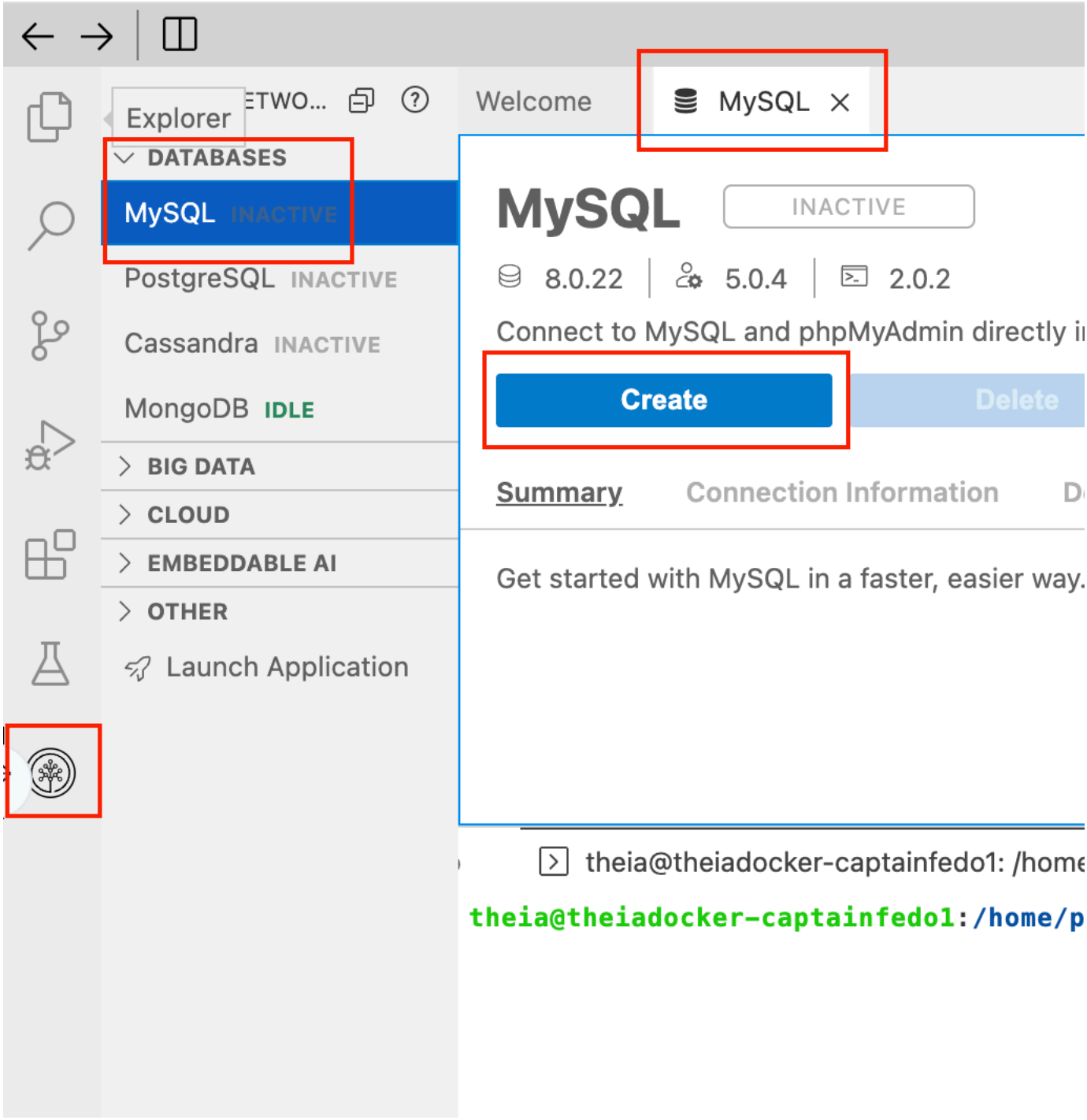
Learning Objectives

After completing this lab, you will be able to:

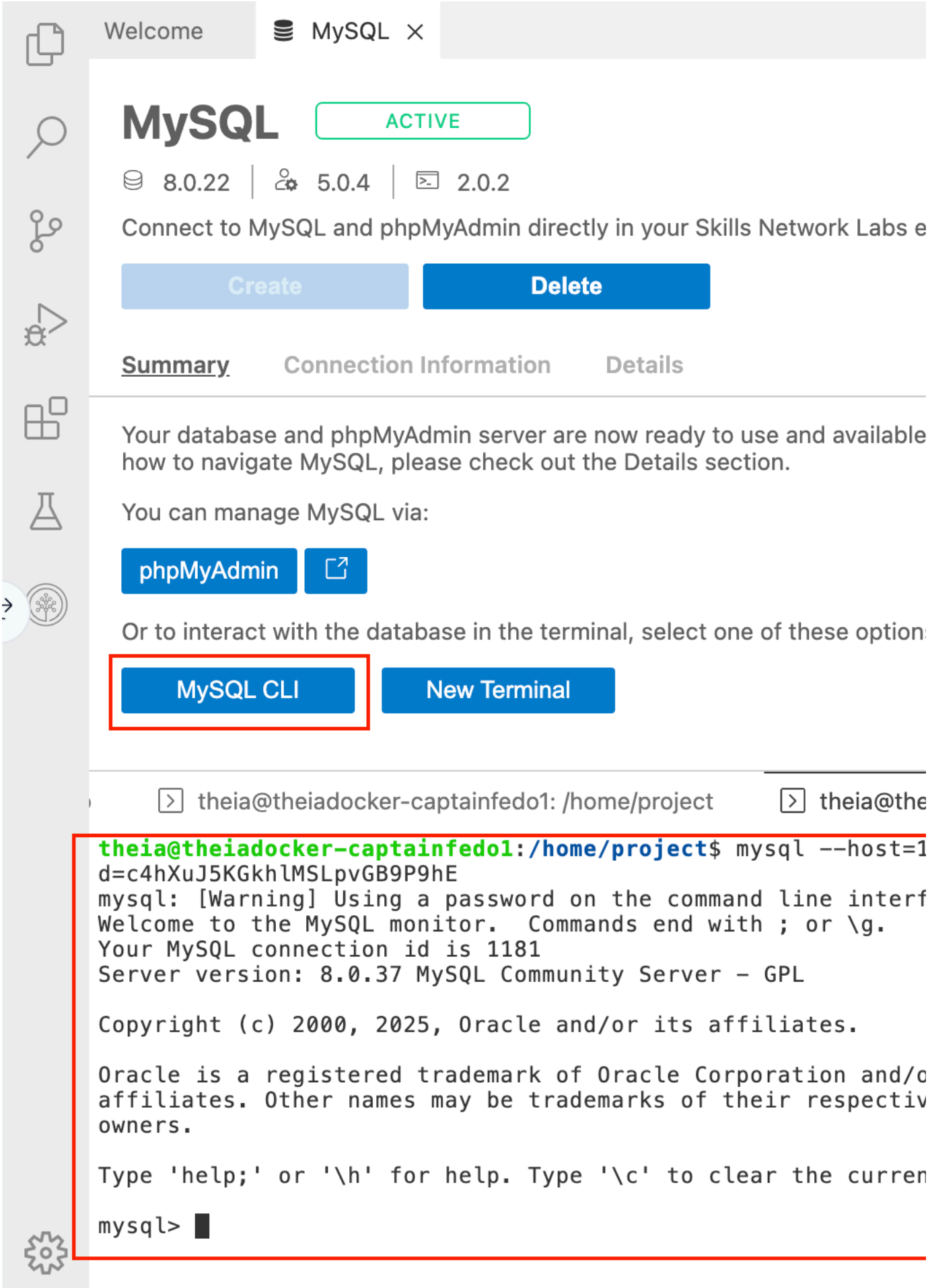
- Use **PreparedStatement** to execute parameterized queries securely.
- Call stored procedures using **CallableStatement**.
- Retrieve **auto-generated keys** from insert operations.
- Handle SQL errors effectively.

Start MySQL Database

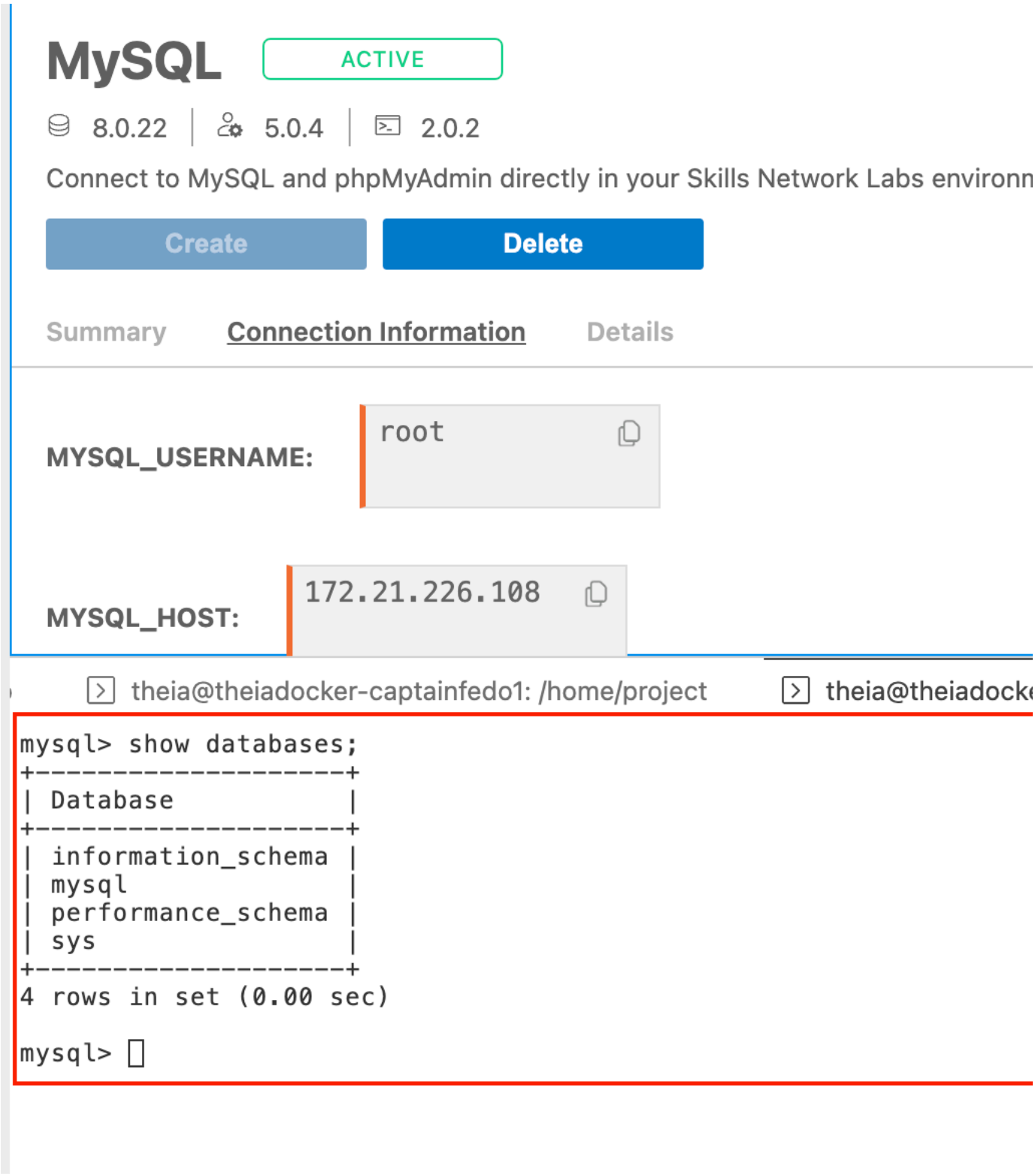
1. The first step is to create the MySQL instance in the IDE environment. Click the toolbox icon at the bottom of the left-hand panel. This will bring up another panel on the left. Select MySQL in the databases category. This will open a new tab. Click the `Create` button.



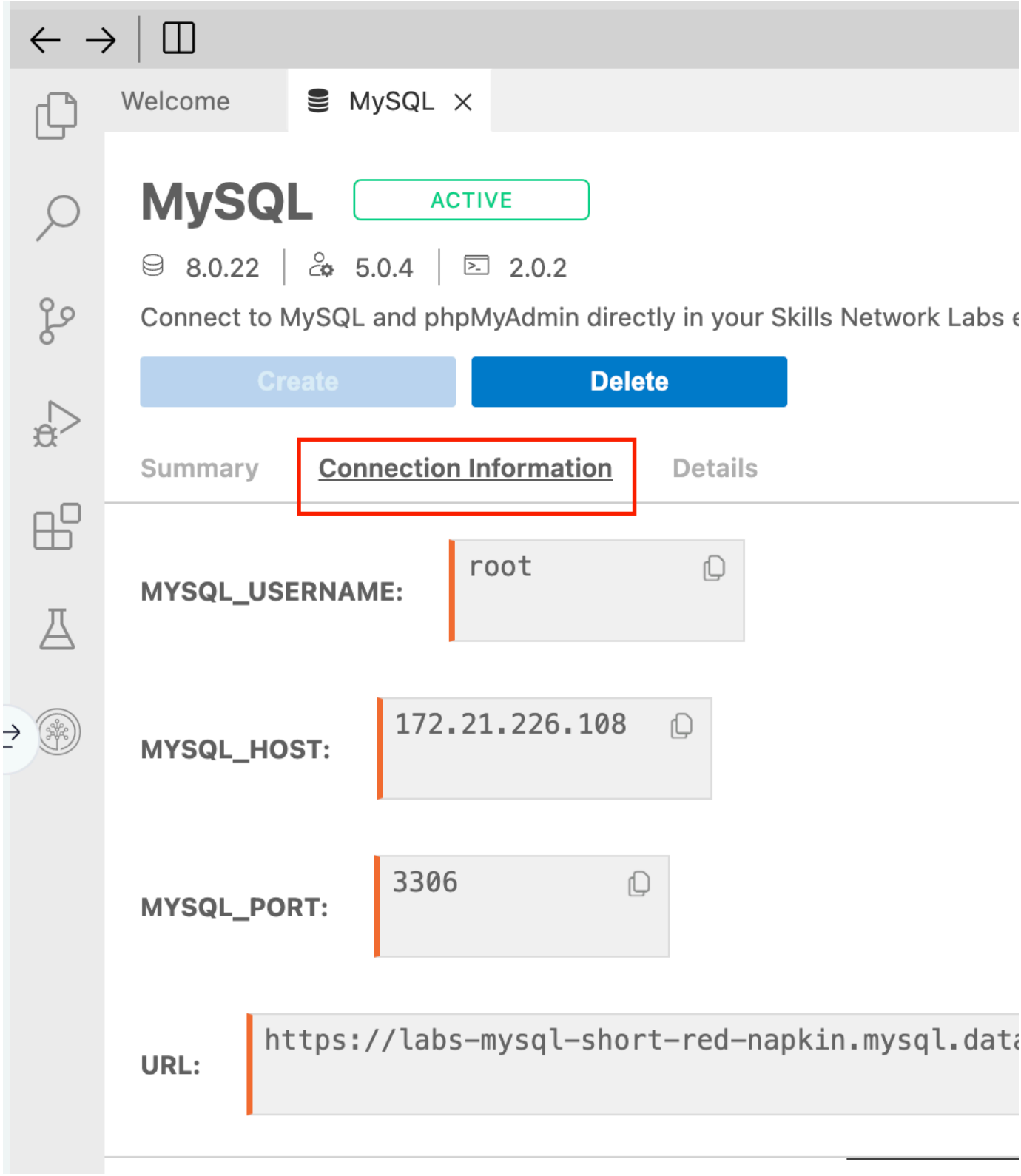
2. The MySQL Database will start in a few minutes. Once it has started, click the `MySQL CLI`button in the `Summary` tab. This will open the MySQL terminal at the bottom of the screen.



3. You can now enter all the **SQL-based** commands in this lab in this terminal. This example shows the command to list all the databases.



4. You can find the username, password, and connection URL information in the `Connection Information` tab if you need to connect to this database outside the IDE environment.



Setting up the Tables

Open the terminal using the `MySQL CLI` button. You will need to create three tables for this lab: `Courses`, `Students`, and `Enrollments` in this order for the dependencies to work.

Creating a database

The first step is to create a database. Use the following command to create a database called MY_DATABASE.

```
CREATE DATABASE MY_DATABASE;
```

Use the `show databases;` command to test whether the command worked. You should see your database listed in the output:

```
+--------------------+
| Database           |
+--------------------+
| MY_DATABASE        |
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.01 sec)
```

Next, use the database for future commands.

```
USE MY_DATABASE;
```

Creating the Courses table

Run the following SQL command in your MySQL CLI to create the Courses table.

```
CREATE TABLE Courses (
    CourseID INT PRIMARY KEY AUTO_INCREMENT,
    CourseName VARCHAR(100) NOT NULL,
    Department VARCHAR(50),
    Credits INT CHECK (Credits > 0) -- Ensuring credit hours are positive
);
```

Now, populate the Courses table with sample courses.

```
INSERT INTO Courses (CourseName, Department, Credits) VALUES
('Introduction to Programming', 'Computer Science', 4),
('Calculus I', 'Mathematics', 3),
('Linear Algebra', 'Mathematics', 3),
('Probability and Statistics', 'Mathematics', 3),
('Differential Equations', 'Mathematics', 4),
('English Literature', 'Humanities', 3),
('Modern History', 'History', 3),
('Physics I', 'Physics', 4),
('Database Systems', 'Computer Science', 4),
('Artificial Intelligence', 'Computer Science', 3);
```

Creating the Students table

Let's create a table called Students with four columns. StudentID, Name, Age, and Major. The StudentID is the primary key. Use the `CREATE TABLE` command to create the table.

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY AUTO_INCREMENT,
    Name VARCHAR(100) NOT NULL,
    Age INT,
    Major VARCHAR(50),
    GPA DECIMAL(3,2)
);
```

Let's populate this table with the following data.

```
INSERT INTO Students (Name, Age, Major, GPA) VALUES
('Alice Johnson', 20, 'Computer Science', 3.8),
('Bob Smith', 22, 'Mathematics', 3.5),
('Charlie Brown', 19, 'History', 3.2),
('David Lee', 21, 'Computer Science', 3.9),
('Eve Wilson', 23, 'English', 3.4),
('Frank Miller', 20, 'Mathematics', 3.6),
('Grace Davis', 22, 'History', 3.1),
('Henry Garcia', 19, 'Computer Science', 3.7),
('Ivy Rodriguez', 21, 'English', 3.3),
('Jack Martinez', 23, 'Mathematics', 3.6),
('Karen White', 20, 'Computer Science', 3.8),
('Liam Green', 22, 'Mathematics', 3.5),
('Mia Taylor', 19, 'History', 3.3),
('Noah Anderson', 21, 'English', 3.4),
('Olivia Thomas', 23, 'Computer Science', 3.9),
('Peter Jackson', 20, 'Mathematics', 3.6),
('Quinn Moore', 22, 'History', 3.2),
('Ryan Martin', 19, 'English', 3.1),
('Sophia Thompson', 21, 'Computer Science', 3.7),
('Tyler Garcia', 23, 'Mathematics', 3.4),
('Ursula Perez', 20, 'Computer Science', 3.5);
```

Creating the Enrollments table

Now, let's create an Enrollments table. This table links students to the courses they are enrolled in.

```
CREATE TABLE Enrollments (
    EnrollmentID INT PRIMARY KEY AUTO_INCREMENT,
    StudentID INT,
    CourseID INT,
    EnrollmentDate DATE NOT NULL,
    FOREIGN KEY (StudentID) REFERENCES Students(StudentID),
    FOREIGN KEY (CourseID) REFERENCES Courses(CourseID)
);
```

Next, let's add some data to this table.

```
INSERT INTO Enrollments (StudentID, CourseID, EnrollmentDate) VALUES
(1, 1, '2024-09-05'),  -- Alice Johnson in Intro to Programming
(1, 2, '2024-09-05'),  -- Alice Johnson in Calculus I
(3, 3, '2024-09-05'),  -- Charlie Brown in Linear Algebra
(5, 1, '2024-09-05'),  -- Eve Wilson in Intro to Programming
(7, 2, '2024-09-05'),  -- Grace Davis in Calculus I
(9, 3, '2024-09-05'),  -- Henry Garcia in Linear Algebra
(11, 1, '2024-09-05'), -- Karen White in Intro to Programming
(13, 2, '2024-09-05'), -- Mia Taylor in Calculus I
(15, 3, '2024-09-05'), -- Noah Anderson in Linear Algebra
(17, 1, '2024-09-05'), -- Ryan Martin in Intro to Programming
(19, 2, '2024-09-05'), -- Sophia Thompson in Calculus I
(21, 3, '2024-09-05'); -- Tyler Garcia in Linear Algebra
```

Connecting MySQL with Java using VS Code

To connect **MySQL** with **Java** using **VS Code**, follow the steps below:

Step 1: Open Terminal in the IDE

Navigate to the project directory using the regular non SQL terminal. If you don't have one open, use the `Terminal` -> `New Terminal` to create a new terminal.

```
cd /home/project
```

Step 2: Create the Maven project

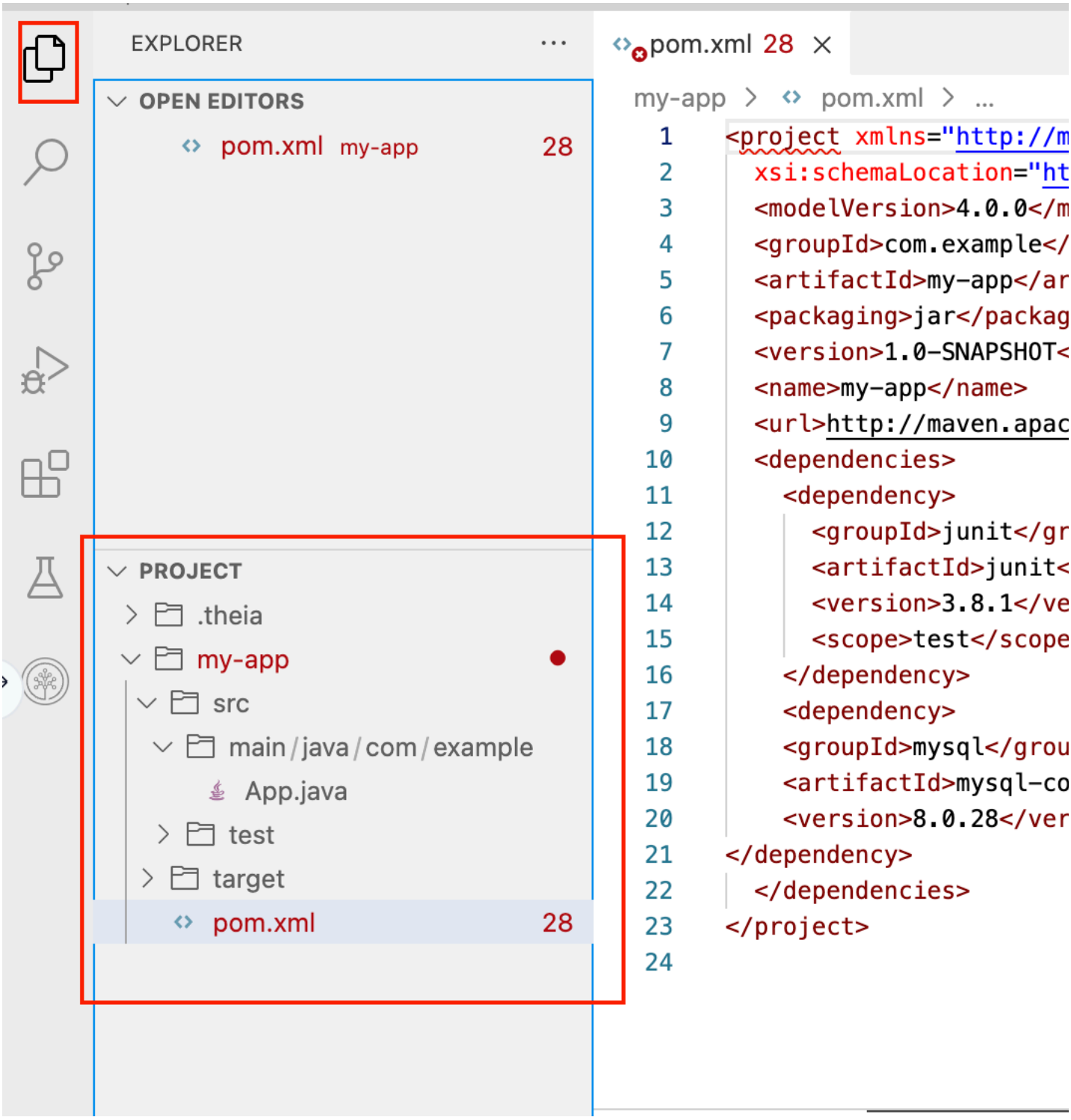
```
mvn archetype:generate \
    -DgroupId=com.example \
    -DartifactId=my-app \
    -Dversion=1.0-SNAPSHOT \
    -Dpackage=com.example \
    -DinteractiveMode=false
```

This Maven command generates a new project structure.

- It uses the `archetype:generate` goal to create a project based on a predefined template.
- The -D flags specify key project details:
  - `-DgroupId` (typically a reverse domain name)
  - `-DartifactId` (the project's name)
  - `-Dversion`
  - `-Dpackage`
- The `-DinteractiveMode=false` flag ensures the command runs non-interactively, using the provided parameters without prompting for further input.

Essentially, it's a quick way to bootstrap a new Java project with a standard directory layout and pom.xml configuration.

This should create a new directory called `my-app` in the project directory. You can see the directory structure in the `Explore` panel of the IDE. Click the **Refresh Explorer** icon if you cannot see it yet.



Step 3: Add MySQL Connector Dependency

Open the `pom.xml` file in your project folder.

Open **pom.xml** in IDE

You will see some errors in `pom.xml` file. This can be fixed by changing `http` to `https` for the `project` tag. Change from:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
...
```

to the following:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
...
```

Add the following MySQL connector dependency under the `<dependencies>` tag:

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.28</version>
</dependency>
```

So it might look something like:

```
    <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.28</version>
    </dependency>
  </dependencies>
```

- After modifying your pom.xml, it should look like this:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>my-app</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.28</version>
    </dependency>
  </dependencies>
</project>
```

Step 4: Install the dependencies

Make sure you are in the `my-app` directory in the terminal and run the `mvn install` command to install the dependencies.

```
cd /home/project/my-app && mvn install
```

The output should show a success message:

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  2.060 s
[INFO] Finished at: 2025-03-21T02:07:45-04:00
[INFO] ------------------------------------------------------------------------
```

Step 5: Navigate to the `App.java` File

Here's the path: **home/project/my-app/src/main/java/com/example/App.java**

Alternatively, click the link below to open the file.

Open **App.java** in IDE

Step 6: Establish Database Connectivity

To establish a connection to MySQL, replace the existing code with the following code inside the `App.java`file. The code has a `main` function that calls other functions from within it.

- `insertStudent(conn)`: insert a student in the database
- `updateStudent(conn)`: updates a student in the database
- `getStudentsByMajor(conn)`: gets all students by a given major
- `callGetStudentsByMajor(conn)`: calls a stored procedure GetStudentsByMajor
- `callGetAverageGPA(conn)`: calls a stored procedure GetAverageGPA
- `insertStudentWithGeneratedKey(conn)`: inserts student and retrieve generated key
- `simulateSQLError(conn)`: simulates an SQL error

Currently, all methods are empty. You will fill out each method in this lab.

```
package com.example;
import java.sql.*;
public class App {
    public static void main(String[] args) {
        String url = "jdbc:mysql://{MYSQL_HOST}:3306/MY_DATABASE"; // Change to your database
        String user = "{MYSQL_USERNAME}"; // Your database username
        String password = "{MYSQL_PASSWORD}"; // Your database password
        try (Connection conn = DriverManager.getConnection(url, user, password)) {
            System.out.println("Connection successful!");
            insertStudent(conn);
            updateStudent(conn);
            getStudentsByMajor(conn);
            callGetStudentsByMajor(conn);
            callGetAverageGPA(conn);
            insertStudentWithGeneratedKey(conn);
            simulateSQLError(conn);
            System.out.println("------------------------");
            System.out.println("Finish!");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    public static void insertStudent(Connection conn) throws SQLException {
        System.out.println("------------------------");
        System.out.println("Insert data into the Students table");
    }
    public static void updateStudent(Connection conn) throws SQLException {
        System.out.println("------------------------");
        System.out.println("Update Students with GPA of 3.9 and name of Alice");
    }
    public static void getStudentsByMajor(Connection conn) throws SQLException {
        System.out.println("------------------------");
        System.out.println("Get all students with major of Computer Science");
    }
    public static void callGetStudentsByMajor(Connection conn) throws SQLException {
        System.out.println("------------------------");
        System.out.println("Calling stored procedure GetStudentsByMajor");
    }
    public static void callGetAverageGPA(Connection conn) throws SQLException {
        System.out.println("------------------------");
        System.out.println("Calling stored procedure GetAverageGPA");
    }
    public static void insertStudentWithGeneratedKey(Connection conn) throws SQLException {
        System.out.println("------------------------");
        System.out.println("Insert student and retrieve generated key");
```

```
    }
    public static void simulateSQLError(Connection conn) {
        System.out.println("------------------------");
        System.out.println("Simulating SQL Error");
    }
}
```

To establish the connection, you need to replace the `{MYSQL_HOST}`, `{DATABASE_NAME}`, `{MYSQL_USERNAME}`, and `{MYSQL_PASSWORD}` with the database details. You'll find all the details, except the Database Name, in the MySQL connection information as explained at the start of this lab.

- Replace `{MYSQL_HOST}` with the MYSQL_HOST value in the MySQL connection information
- Replace the `{DATABASE_NAME}`with your Database name. In this case it is `MY_DATABASE`.
- Replace `{MYSQL_USERNAME}`, and `{MYSQL_PASSWORD}` as per your MySQL database credentials.

For example, for my environment, this looks like:

```
String url = "jdbc:mysql://172.21.128.49:3306/MY_DATABASE"; // Change to your database
String user = "root"; // Your database username
String password = "WUrb0ISQ3hroWlAqHWSUsfiM"; // Your database password
```

After setting up the project, you can proceed with running your JDBC queries inside the `try` block.

Run the Application

Use the following command to run the program.

```
cd /home/project/my-app && mvn clean install && mvn exec:java -Dexec.mainClass="com.example.App"
```

You should see the logs say "connection successful!" and outputs each method being called.

```
...
...
[INFO] --- exec:3.5.0:java (default-cli) @ my-app ---
Connection successful!
------------------------
Insert data into the Students table
------------------------
Update Students with GPA of 3.9 and name of Alice
------------------------
Get all students with major of Computer Science
------------------------
Calling stored procedure GetStudentsByMajor
------------------------
Calling stored procedure GetAverageGPA
------------------------
Insert student and retrieve generated key
------------------------
Simulating SQL Error
------------------------
...
...
```

Understanding PreparedStatement

Why Use PreparedStatement?

- **Prevents SQL Injection** by treating parameters as data.
- **Improves Performance** by precompiling queries.
- **Reusable** for executing the same query with different values.

Insert Data Using PreparedStatement

Let's fill out the `insertStudent` method using a preparedStatement to insert a student into the table. Ideally, the values will not be hardcoded, but passed into the method.

Copy the code below into the `App.java` file and replace the existing `insertStudent` method.

```
public static void insertStudent(Connection conn) throws SQLException {
        System.out.println("------------------------");
        System.out.println("Insert data into the Students table");
        try (PreparedStatement preparedStatement = conn.prepareStatement(
                "INSERT INTO Students (Name, Major, GPA) VALUES (?, ?, ?)")) {
            preparedStatement.setString(1, "Alice Johnson");
            preparedStatement.setString(2, "Computer Science");
            preparedStatement.setDouble(3, 3.8);
            int rowsAffected = preparedStatement.executeUpdate();
            System.out.println("Rows inserted: " + rowsAffected);
        }
    }
```

Explanation

- **? Placeholders** allow dynamic query execution.
- **setString() and setDouble()** bind actual values.
- **executeUpdate()** runs the SQL statement.

Run the program again.
Remember to run the program from the regular terminal and not the MySQL CLI terminal.

Use the following command to run the program.

```
cd /home/project/my-app && mvn clean install && mvn exec:java -Dexec.mainClass="com.example.App"
```

You should see a message such as the one below, stating `Row inserted: 1`:

```
[INFO] --- exec:3.5.0:java (default-cli) @ my-app ---
Connection successful!
------------------------
Insert data into the Students table
Rows inserted: 1
------------------------
```

Updating and Retrieving Data

Update Data Using PreparedStatement

We are hardcoding the major and GPA in this lab, but ideally, you would pass this information to the method from the `main` function.

Copy the code below into the `App.java` file and replace the existing `updateStudent` method.

```
public static void updateStudent(Connection conn) throws SQLException {
```

```
        System.out.println("------------------------");
        System.out.println("Update Students with GPA of 3.9 and name of Alice");
        try (PreparedStatement preparedStatement = conn.prepareStatement(
                "UPDATE Students SET GPA = ? WHERE Name = ?")) {
            preparedStatement.setDouble(1, 3.9);
            preparedStatement.setString(2, "Alice Johnson");
            int rowsAffected = preparedStatement.executeUpdate();
            System.out.println("Rows updated: " + rowsAffected);
        }
    }
```

Run the program again.
Remember to run the program from the regular terminal and not the MySQL CLI terminal.

Use the following command to run the program.

```
cd /home/project/my-app && mvn clean install && mvn exec:java -Dexec.mainClass="com.example.App"
```

You should see something like the following. The output might look different depending on how many times you have run the code. The output here shows that two rows were already inserted and this run inserted one more and then updated three rows at once:

```
Connection successful!
------------------------
Insert data into the Students table
Rows inserted: 1
------------------------
Update Students with GPA of 3.9 and name of Alice
Rows updated: 3
------------------------
```

Retrieve Data Using PreparedStatement

Let's look at how to get data using a PreparedStatement.

Copy the code below into the `App.java` file and replace the existing `getStudentsByMajor` method.

```
public static void getStudentsByMajor(Connection conn) throws SQLException {
        System.out.println("------------------------");
        System.out.println("Get all students with major of Computer Science");
        try (PreparedStatement preparedStatement = conn.prepareStatement(
                "SELECT * FROM Students WHERE Major = ?")) {
            preparedStatement.setString(1, "Computer Science");
            ResultSet resultSet = preparedStatement.executeQuery();
            while (resultSet.next()) {
                System.out.println("Name: " + resultSet.getString("Name"));
                System.out.println("GPA: " + resultSet.getDouble("GPA"));
            }
        }
    }
```

Run the program again.
Remember to run the program from the regular terminal and not the MySQL CLI terminal.

Use the following command to run the program.

```
cd /home/project/my-app && mvn clean install && mvn exec:java -Dexec.mainClass="com.example.App"
```

The output should look something like this:

```
[INFO] --- exec:3.5.0:java (default-cli) @ my-app ---
Connection successful!
------------------------
Insert data into the Students table
Rows inserted: 1
------------------------
Update Students with GPA of 3.9 and name of Alice
Rows updated: 4
------------------------
Get all students with major of Computer Science
Name: Alice Johnson
GPA: 3.9
Name: David Lee
GPA: 3.9
Name: Henry Garcia
GPA: 3.7
Name: Karen White
GPA: 3.8
Name: Olivia Thomas
GPA: 3.9
Name: Sophia Thompson
GPA: 3.7
Name: Ursula Perez
GPA: 3.5
Name: Alice Johnson
GPA: 3.9
Name: Alice Johnson
GPA: 3.9
Name: Alice Johnson
GPA: 3.9
------------------------
```

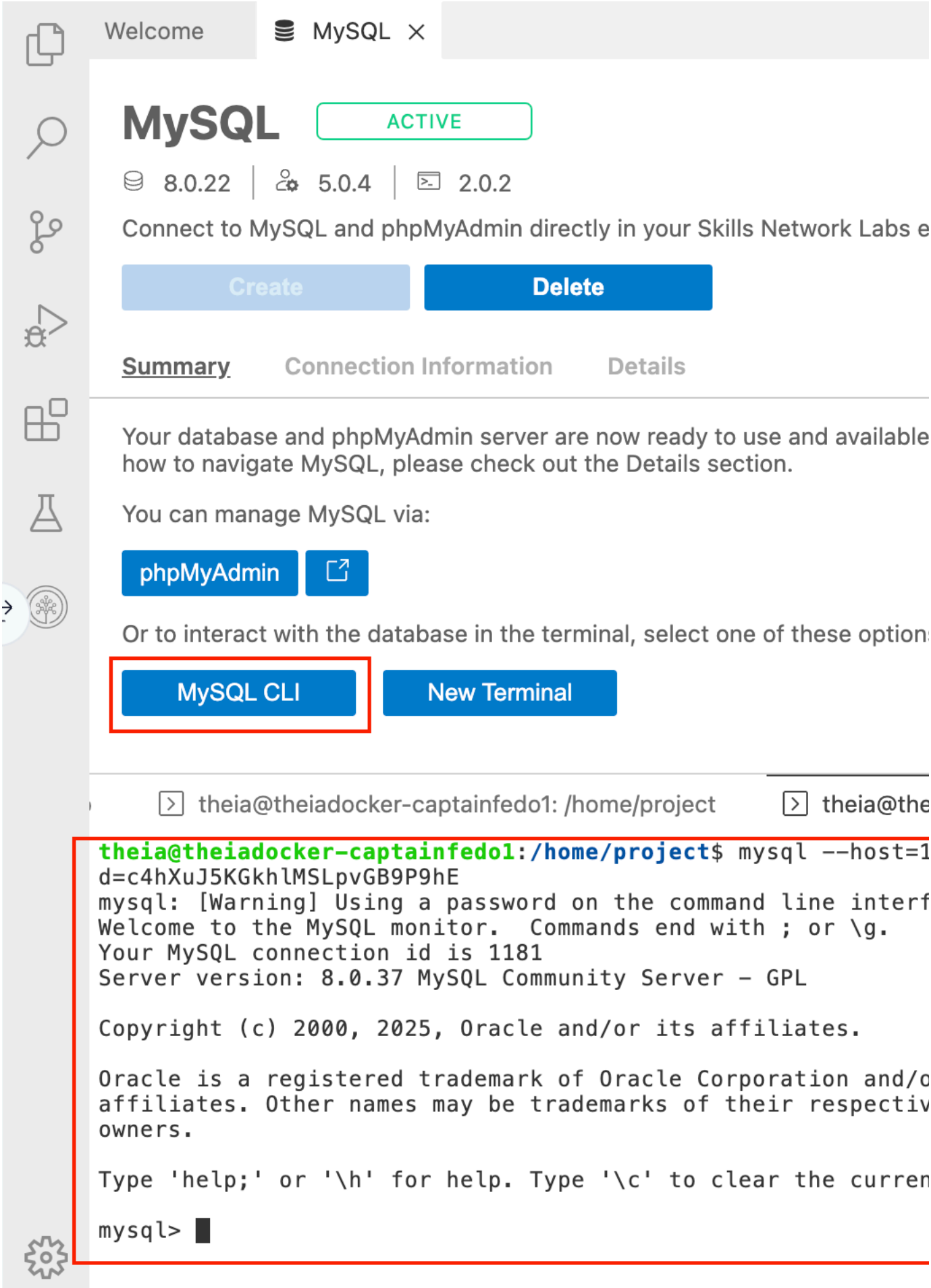
Using CallableStatement for Stored Procedures

What is CallableStatement?

- **Executes stored procedures** from a database.
- **Improves performance** by encapsulating SQL logic.

Example: Calling a Stored Procedure with an Input Parameter

1. First, open the `MySQL CLI` from the `MySql` tab.



2. Next, run the following command to ensure you are using the correct `MY_DATABASE` database for future commands.

```
USE MY_DATABASE;
```

3. Next, copy and paste the following code into the MySQL CLI to create the first stored procedure.

Stored Procedure

```
DELIMITER $$
CREATE PROCEDURE GetStudentsByMajor(IN majorName VARCHAR(255))
BEGIN
    SELECT Name, GPA FROM Students WHERE Major = majorName;
END$$
DELIMITER ;
```

4. You can use the following command to ensure the procedure was created.

```
SHOW PROCEDURE STATUS
WHERE Db = DATABASE() AND Type = 'PROCEDURE';
```

```
mysql> SHOW PROCEDURE STATUS
    -> WHERE Db = DATABASE() AND Type = 'PROCEDURE'
    -> ;
+-------------+--------------------+-----------+---------+----------
----------------+----------------------+--------------------+
| Db          | Name               | Type      | Definer | Modified
ter_set_client | collation_connection | Database Collation |
+-------------+--------------------+-----------+---------+----------
----------------+----------------------+--------------------+
| MY_DATABASE | GetStudentsByMajor | PROCEDURE | root@%  | 2025-03-2
               | latin1_swedish_ci    | utf8mb4_0900_ai_ci |
+-------------+--------------------+-----------+---------+----------
----------------+----------------------+--------------------+
1 row in set (0.00 sec)
```

Finally, we can fill out the `callGetStudentsByMajor` method to call this procedure.

5. Switch back to the `App.java` file, and copy the code below into the `App.java` file and replace the existing `callGetStudentsByMajor` method.

Java Code

```
public static void callGetStudentsByMajor(Connection conn) throws SQLException {
        System.out.println("------------------------");
        System.out.println("Calling stored procedure GetStudentsByMajor");
        try (CallableStatement callableStatement = conn.prepareCall("{CALL GetStudentsByMajor(?)}")) {
            callableStatement.setString(1, "Computer Science");
            ResultSet resultSet = callableStatement.executeQuery();
            while (resultSet.next()) {
                System.out.println("Name: " + resultSet.getString("Name"));
                System.out.println("GPA: " + resultSet.getDouble("GPA"));
            }
        }
    }
```

6. Run the program again, using the following command:

Remember to run the program from the standard theia@theiadocker terminal and not the MySQL CLI terminal.

```
cd /home/project/my-app && mvn clean install && mvn exec:java -Dexec.mainClass="com.example.App"
```

You should see the following added to the output. Your actual output might differ depending on how many times you have already run this code.

```
...
...
------------------------
Calling stored procedure GetStudentsByMajor
Name: Alice Johnson
GPA: 3.9
Name: David Lee
GPA: 3.9
Name: Henry Garcia
GPA: 3.7
Name: Karen White
GPA: 3.8
Name: Olivia Thomas
GPA: 3.9
Name: Sophia Thompson
GPA: 3.7
Name: Ursula Perez
GPA: 3.5
Name: Alice Johnson
GPA: 3.9
Name: Alice Johnson
GPA: 3.9
Name: Alice Johnson
GPA: 3.9
Name: Alice Johnson
GPA: 3.9
------------------------
...
...
```

Example: Calling a Stored Procedure with an Output Parameter

1. Now switch back to the **MySQL CLI terminal** and run the following command to ensure you are using the correct `MY_DATABASE` database for future commands.

```
USE MY_DATABASE;
```

2. Copy and paste the following code into the MySQL CLI to create the second stored procedure.

Stored Procedure

```
DELIMITER $$
CREATE PROCEDURE GetAverageGPA(OUT avgGPA DOUBLE)
BEGIN
    SELECT AVG(GPA) INTO avgGPA FROM Students;
END$$
DELIMITER ;
```

3. Run the `SHOW PROCEDURE` command again as below, to ensure the procedure was created.

```
SHOW PROCEDURE STATUS
WHERE Db = DATABASE() AND Type = 'PROCEDURE';
```

You should now see two stored procedures.

```
mysql> SHOW PROCEDURE STATUS
    -> WHERE Db = DATABASE() AND Type = 'PROCEDURE';
+-------------+--------------------+-----------+---------+----------
-----+----------------------+----------------------+----------------
| Db          | Name               | Type      | Definer | Modified
ment | character_set_client | collation_connection | Database Collat
+-------------+--------------------+-----------+---------+----------
-----+----------------------+----------------------+----------------
| MY_DATABASE | GetAverageGPA      | PROCEDURE | root@%  | 2025-03-3
     | latin1               | latin1_swedish_ci    | utf8mb4_0900_ai
| MY_DATABASE | GetStudentsByMajor | PROCEDURE | root@%  | 2025-03-3
     | latin1               | latin1_swedish_ci    | utf8mb4_0900_ai
+-------------+--------------------+-----------+---------+----------
-----+----------------------+----------------------+----------------
2 rows in set (0.00 sec)
```

Finally, you'll fill out the `callGetAverageGPA` method that calls this second stored procedure.

4. Switch back to the `App.java` file, and copy the code below into the `App.java` file and replace the existing `callGetAverageGPA` method.

Java Code

```
public static void callGetAverageGPA(Connection conn) throws SQLException {
        System.out.println("------------------------");
        System.out.println("Calling stored procedure GetAverageGPA");
        try (CallableStatement callableStatement = conn.prepareCall("{CALL GetAverageGPA(?)}")) {
            callableStatement.registerOutParameter(1, java.sql.Types.DOUBLE);
            callableStatement.execute();
            double avgGPA = callableStatement.getDouble(1);
            System.out.println("Average GPA: " + avgGPA);
        }
    }
```

Explanation

- **registerOutParameter()** binds the output parameter.
- **execute()** runs the stored procedure.
- **getDouble(1)** retrieves the returned value.

Run the program again.
Remember to run the program from the regular terminal and not the MySQL CLI terminal.

Use the following command to run the program.

```
cd /home/project/my-app && mvn clean install && mvn exec:java -Dexec.mainClass="com.example.App"
```

You should see a new output:

```
...
...
------------------------
Calling stored procedure GetAverageGPA
Average GPA: 3.58076923
------------------------
...
...
```

Retrieving Auto-Generated Keys

When inserting data into a database table, it's common for the primary key to be automatically generated by the database, usually using an AUTO_INCREMENT column. In such cases, you might want to retrieve this generated key for further operations, like referencing it in another table. This is particularly useful in scenarios where you're managing relational data or implementing logging and auditing features.

The following code demonstrates how to insert a new student record into a Students table using a PreparedStatement and retrieve the auto-generated primary key using Statement.RETURN_GENERATED_KEYS.

Copy the code below into the `App.java` file and replace the existing `insertStudentWithGeneratedKey` method.

Insert Data and Retrieve Generated Key

```
public static void insertStudentWithGeneratedKey(Connection conn) throws SQLException {
        System.out.println("------------------------");
        System.out.println("Insert student and retrieve generated key");
        try (PreparedStatement preparedStatement = conn.prepareStatement(
                "INSERT INTO Students (Name, Major, GPA) VALUES (?, ?, ?)",
                Statement.RETURN_GENERATED_KEYS)) {
            preparedStatement.setString(1, "John Doe");
            preparedStatement.setString(2, "Mathematics");
            preparedStatement.setDouble(3, 3.6);
            preparedStatement.executeUpdate();
            ResultSet generatedKeys = preparedStatement.getGeneratedKeys();
            if (generatedKeys.next()) {
                int newStudentID = generatedKeys.getInt(1);
                System.out.println("Generated Student ID: " + newStudentID);
            }
        }
    }
```

Explanation

- Statement.RETURN_GENERATED_KEYS signals to the database that the generated primary keys should be returned after the execution of the INSERT statement.
- preparedStatement.setString() and preparedStatement.setDouble() are used to safely insert values into the query, preventing SQL injection.
- preparedStatement.executeUpdate() runs the INSERT statement and returns the number of affected rows.
- preparedStatement.getGeneratedKeys() provides access to the primary key that was automatically generated by the database.
- generatedKeys.next() ensures there is a result available, and generatedKeys.getInt(1) retrieves the primary key (typically the first column).

Run the program again.
Remember to run the program from the regular terminal and not the MySQL CLI terminal.

Use the following command to run the program.

```
cd /home/project/my-app && mvn clean install && mvn exec:java -Dexec.mainClass="com.example.App"
```

You should see the key retrieved logged in the output. The actual key value might be different for your code.

```
...
...
------------------------
Insert student and retrieve generated key
Generated Student ID: 28
------------------------
...
...
```

Handling SQL Exceptions

The following code demonstrates how to handle SQL errors using a try-catch block in JDBC. It intentionally attempts to query a non-existent table to simulate an SQL error, catch the resulting exception, and print a detailed error message.

Copy the code below into the `App.java` file and replace the existing `simulateSQLError` method.

Handling Errors in JDBC

```
public static void simulateSQLError(Connection conn) {
        System.out.println("------------------------");
        System.out.println("Simulating SQL Error");
        try (PreparedStatement preparedStatement = conn.prepareStatement(
                "SELECT * FROM NonExistentTable")) {
            preparedStatement.executeQuery();
        } catch (SQLException e) {
            System.err.println("SQL Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

Explanation

- **Catching exceptions** prevents application crashes.
- **SQLException handling** ensures debugging information is available.

Run the program again.
Remember to run the program from the regular terminal and not the MySQL CLI terminal.

Use the following command to run the program.

```
cd /home/project/my-app && mvn clean install && mvn exec:java -Dexec.mainClass="com.example.App"
```

The output should show the simulated error.

```
...
...
------------------------
Simulating SQL Error
SQL Error: Table 'MY_DATABASE.NonExistentTable' doesn't exist
java.sql.SQLSyntaxErrorException: Table 'MY_DATABASE.NonExistentTable' doesn't exist
        at com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.java:120)
        at com.mysql.cj.jdbc.exceptions.SQLExceptionsMapping.translateException(SQLExceptionsMapping.java:122)
        at com.mysql.cj.jdbc.ClientPreparedStatement.executeInternal(ClientPreparedStatement.java:953)
        at com.mysql.cj.jdbc.ClientPreparedStatement.executeQuery(ClientPreparedStatement.java:1009)
        at com.example.App.simulateSQLError(App.java:130)
        at com.example.App.main(App.java:20)
```

```
        at org.codehaus.mojo.exec.ExecJavaMojo.doMain(ExecJavaMojo.java:375)
        at org.codehaus.mojo.exec.ExecJavaMojo.doExec(ExecJavaMojo.java:364)
        at org.codehaus.mojo.exec.ExecJavaMojo.lambda$execute$0(ExecJavaMojo.java:286)
        at java.base/java.lang.Thread.run(Thread.java:1583)
------------------------
Finish!
...
...
```

Conclusion and Next Steps

Congratulations! You have successfully learned how to use PreparedStatement and CallableStatement in JDBC, mastering advanced techniques for secure and efficient database interactions. You've demonstrated the ability to prevent SQL injection, improve query performance, and encapsulate complex database logic through stored procedures.

This lab provided a practical understanding of parameterized queries, stored procedure execution, auto-generated key retrieval, and robust error handling. You are now equipped to build more secure and efficient Java applications that interact with relational databases.

Next Steps:

1. Practice Parameterized Queries with Various Data Types: Experiment with using PreparedStatement to execute queries with different data types, such as dates, timestamps, and binary data. Practice binding parameters using the appropriate setXXX() methods.
2. Develop Stored Procedures with Multiple Input/Output Parameters: Create and call stored procedures that involve multiple input and output parameters. This will deepen your understanding of how to use CallableStatement to interact with complex database-side logic.
3. Implement Robust Error Handling for Stored Procedure Calls: Practice handling SQLException when calling stored procedures, specifically focusing on handling different types of errors that can occur during stored procedure execution.

Author(s)

Upkar Lidder