Java SE & Spring

Module 1: Java SE

16.Java Database Connectivity (JDBC)

JDBC

The Java Database Connectivity (JDBC) API provides universal data access from the Java. Using the JDBC API, you can access virtually any data source, from relational databases to spreadsheets and flat files. The JDBC API is comprised of two packages:

- java.sql
- javax.sql

Both packages are downloaded with Java Platform Standard Edition (Java SE).

Further reading: https://howtodoinjava.com/jdbc-tutorials/

JDBC Drivers

Java database connectivity (JDBC) is JavaSoft specification of a standard application programming interface (API) that allows Java programs to access database management systems. The JDBC API consists of a set of interfaces and classes written in the Java programming language. Using these standard interfaces and classes, programmers can write applications that connect to databases, send queries written in structured query language (SQL), and process the results. JDBC is oriented towards relational databases.

Because JDBC is a standard specification, a Java program that uses the JDBC API can connect to any database management system (DBMS) for which there is a JDBC driver.

Further reading: https://howtodoinjava.com/jdbc-tutorials/

JDBC Drivers

The JDBC API defines the Java interfaces and classes that programmers use to connect to databases and send queries.

A Java program (that uses the JDBC API) loads the specified driver for a particular DBMS before it actually connects to a database. The JDBC's DriverManager class then sends all JDBC API calls to the loaded driver.

There are 4 different types of JDBC drivers:

- Type 1 : JDBC-ODBC bridge driver
- Type 2 : Native-API Driver
- Type 3 : All Java + Middleware translation driver
- Type 4 : Pure Java driver

Further reading: https://howtodoinjava.com/jdbc-tutorials/

Database Types

Generally, databases is comprised of two types. First, Relational Databases (SQL), and Not Relational (NoSQL) databases. The most common types of NoSQL databases are key-value, document, column and graph databases.

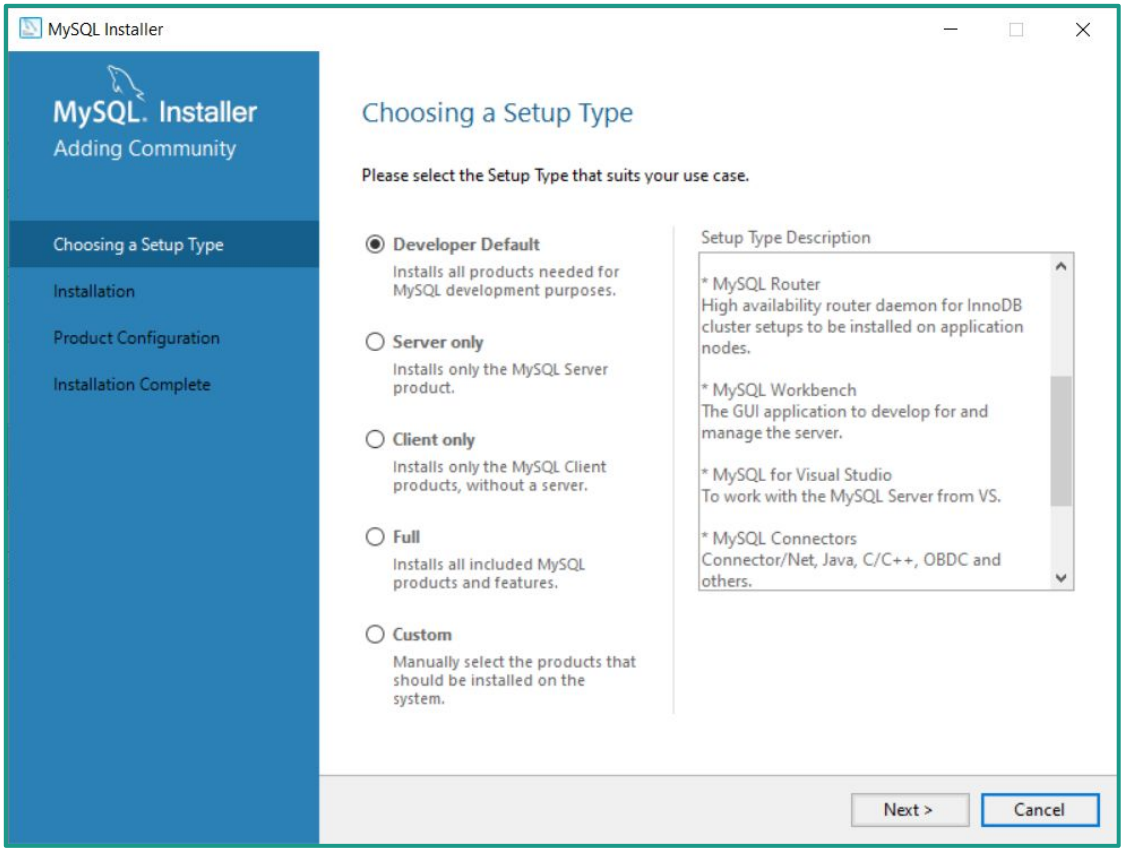
In this course we will talk about MySQL Relational Database. It is open source and has a free community version. You can download it from here.

Installing MySQL

After you download the MySQL installer from the link at the previous slide.

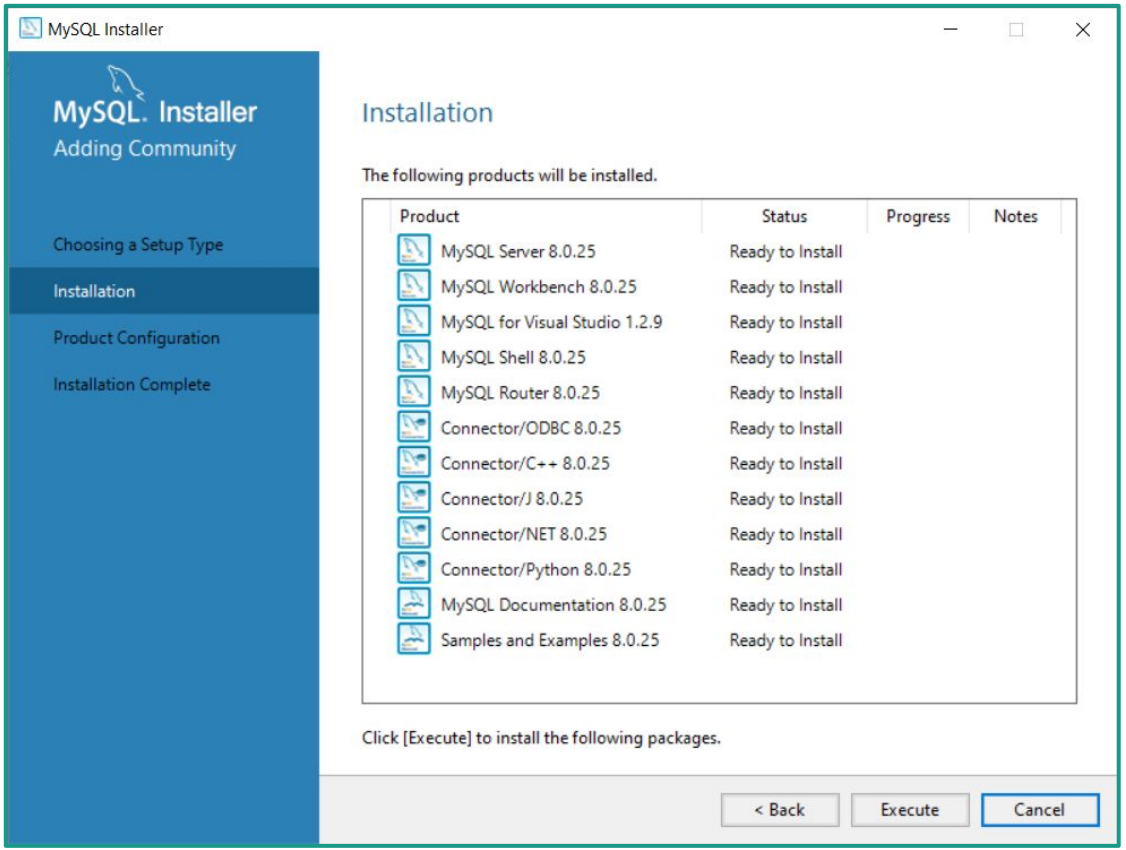
Open the installer by double-clicking➡

Choose “Developer Default”, and click **Next**



Installing MySQL

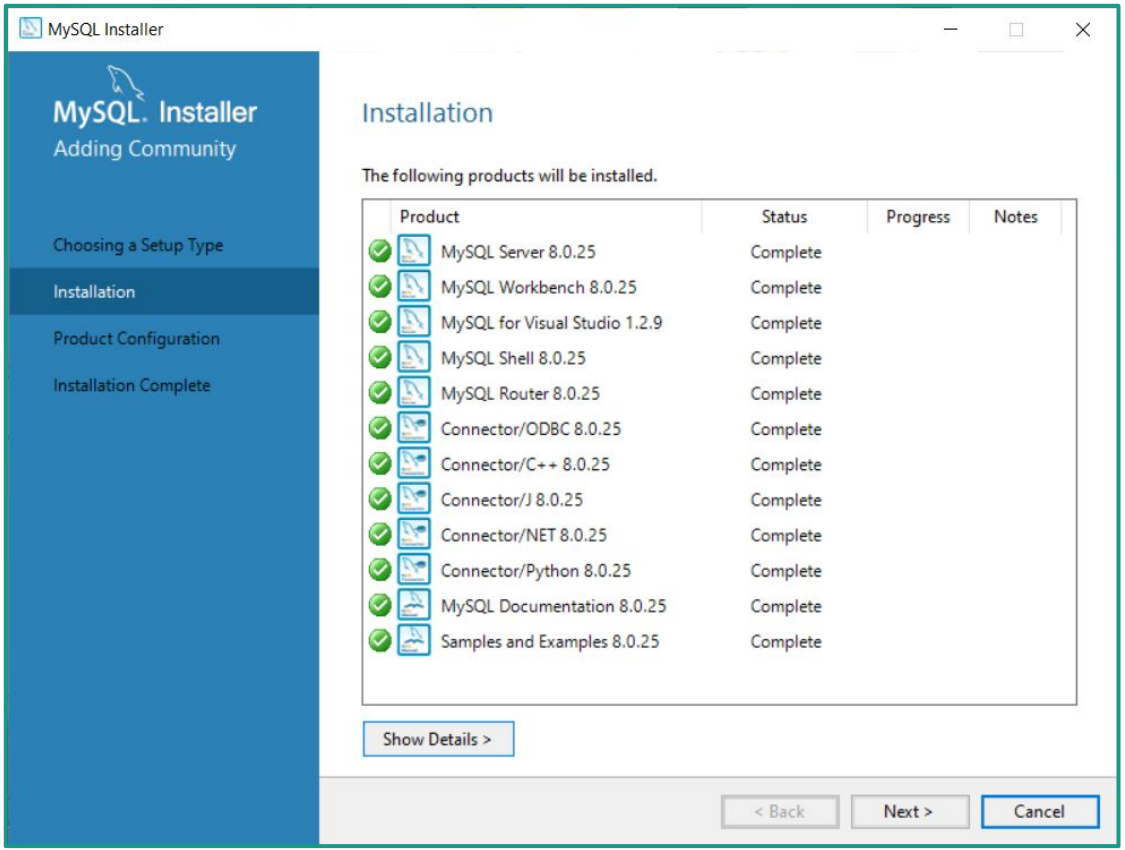
Click the **Execute** button.



Installing MySQL

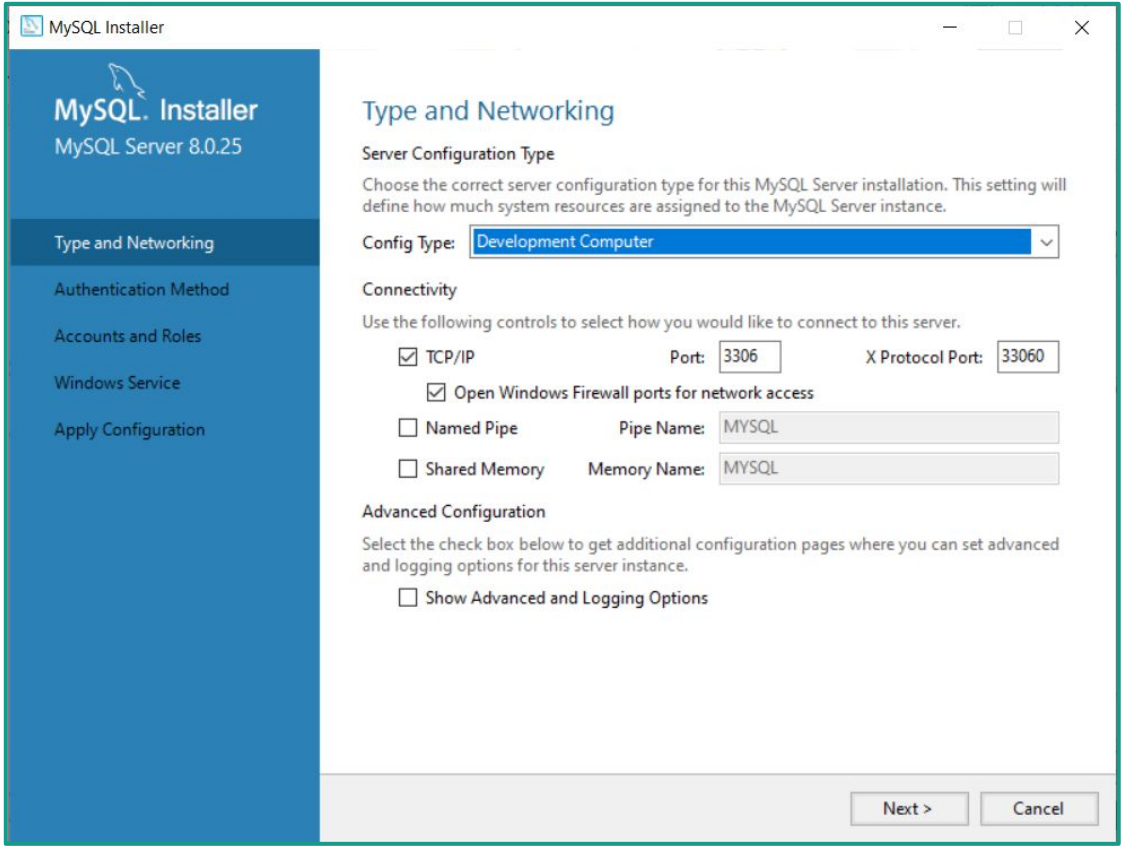
When installation is completed ,

Click **Next** button .



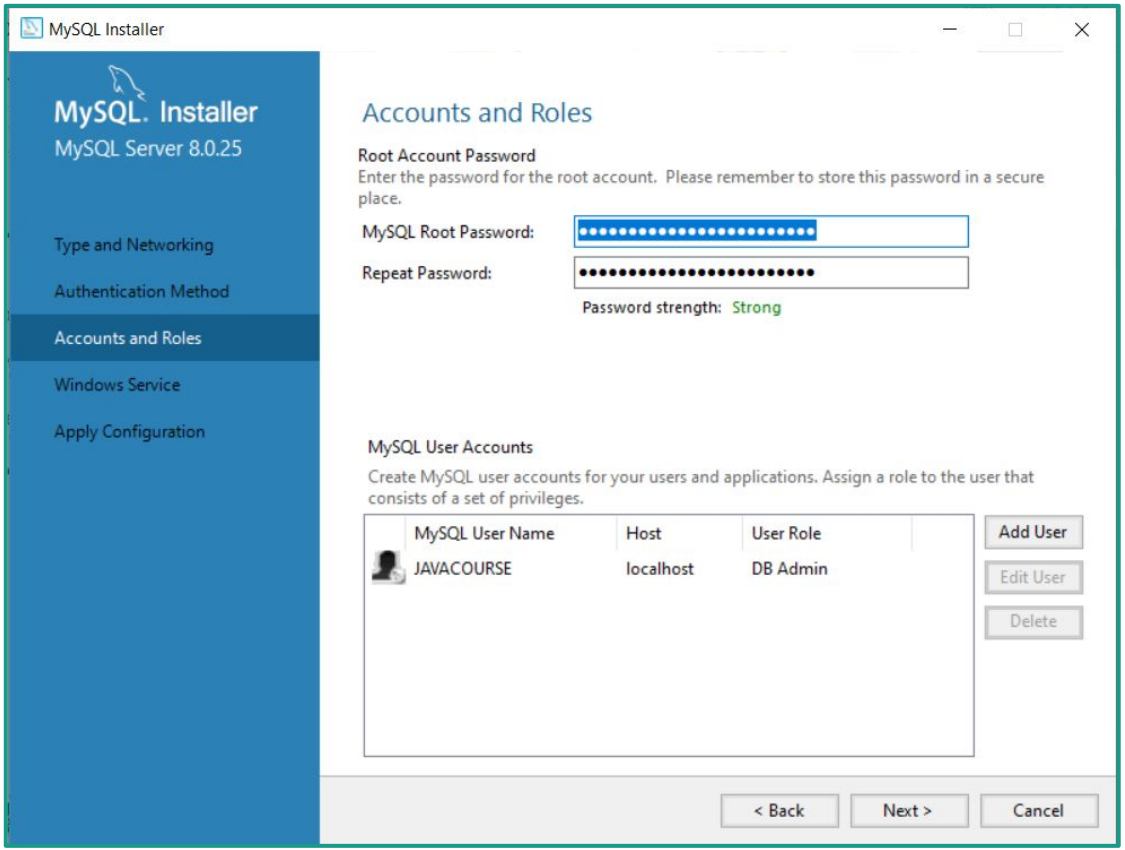
Installing MySQL

In the configuration page, choose “Development Computer” as config type and click **Next**.



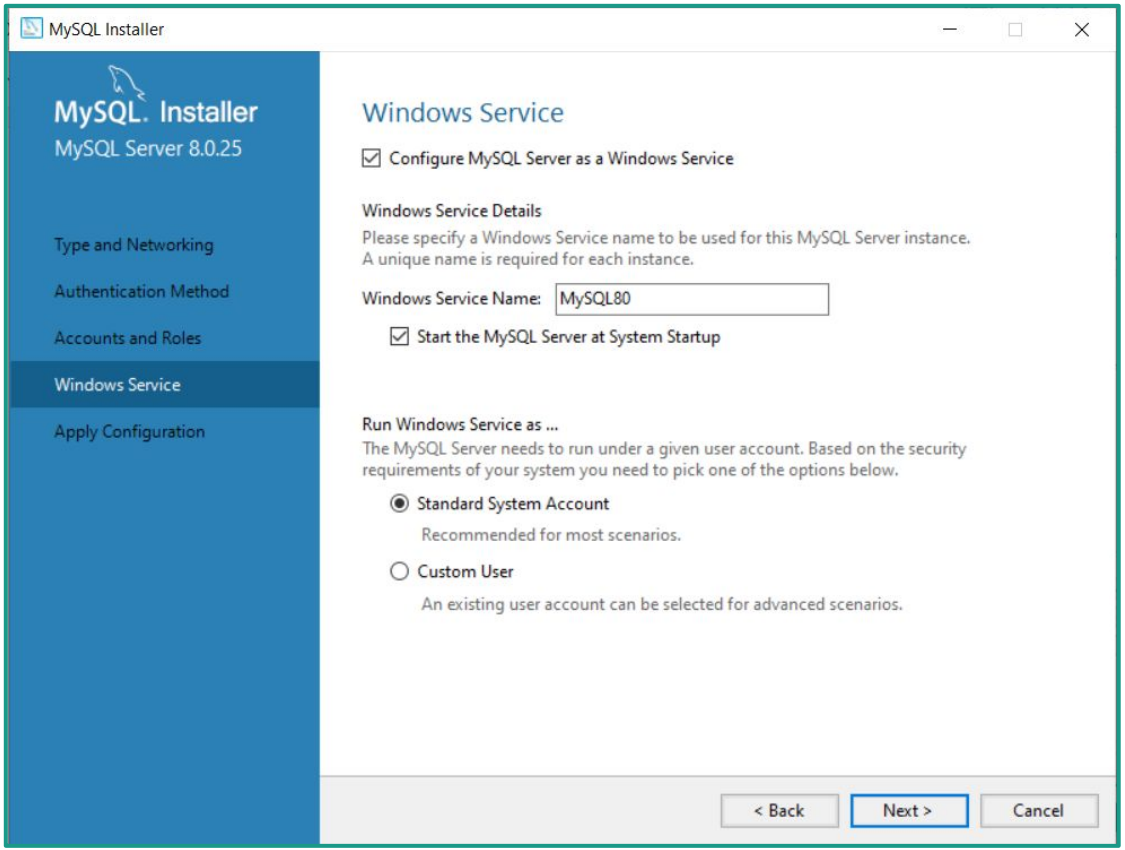
Installing MySQL

Define root user password and create a user with DBA role. (Strong, and hard to guess by others passwords are recommended. 😉)



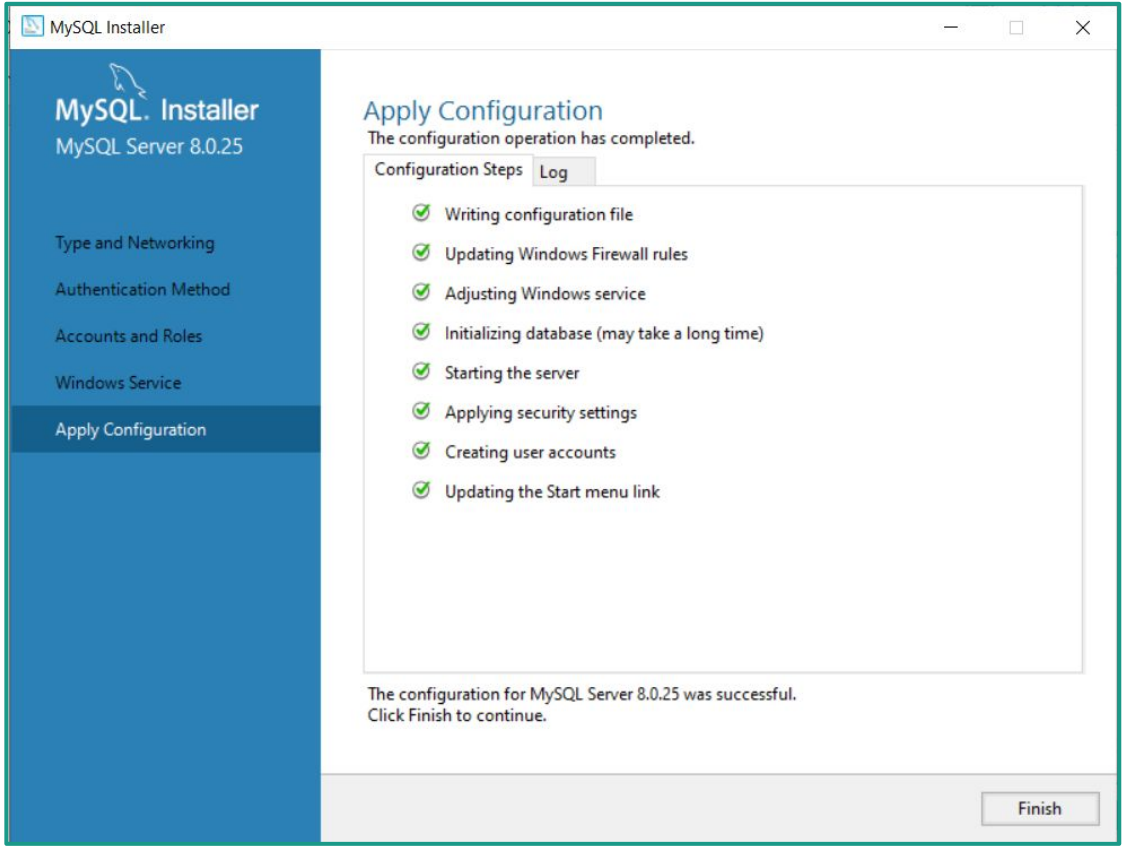
Installing MySQL

Leave the options as it is in this page and click **Next.**



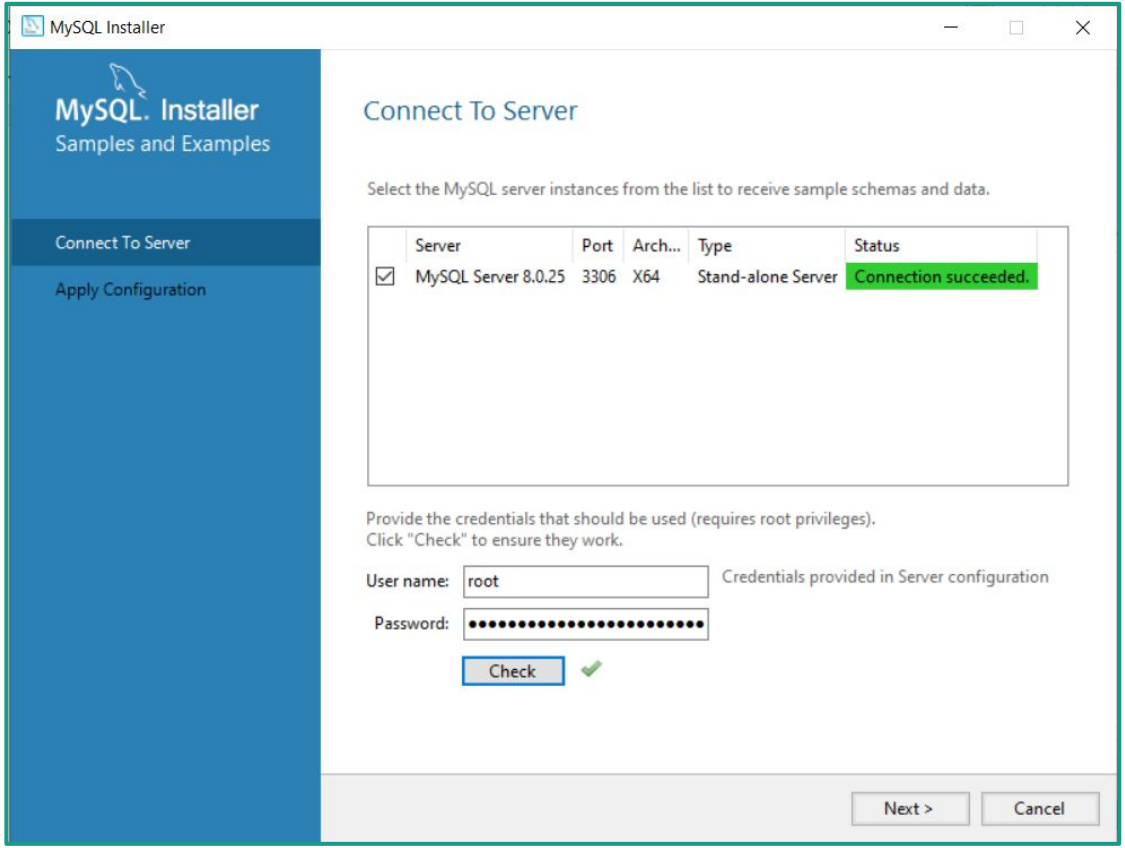
Installing MySQL

Then, click the execute button, after the configuration is finished, you should see this page. Click the **Finish** button to continue remaining steps.



Installing MySQL

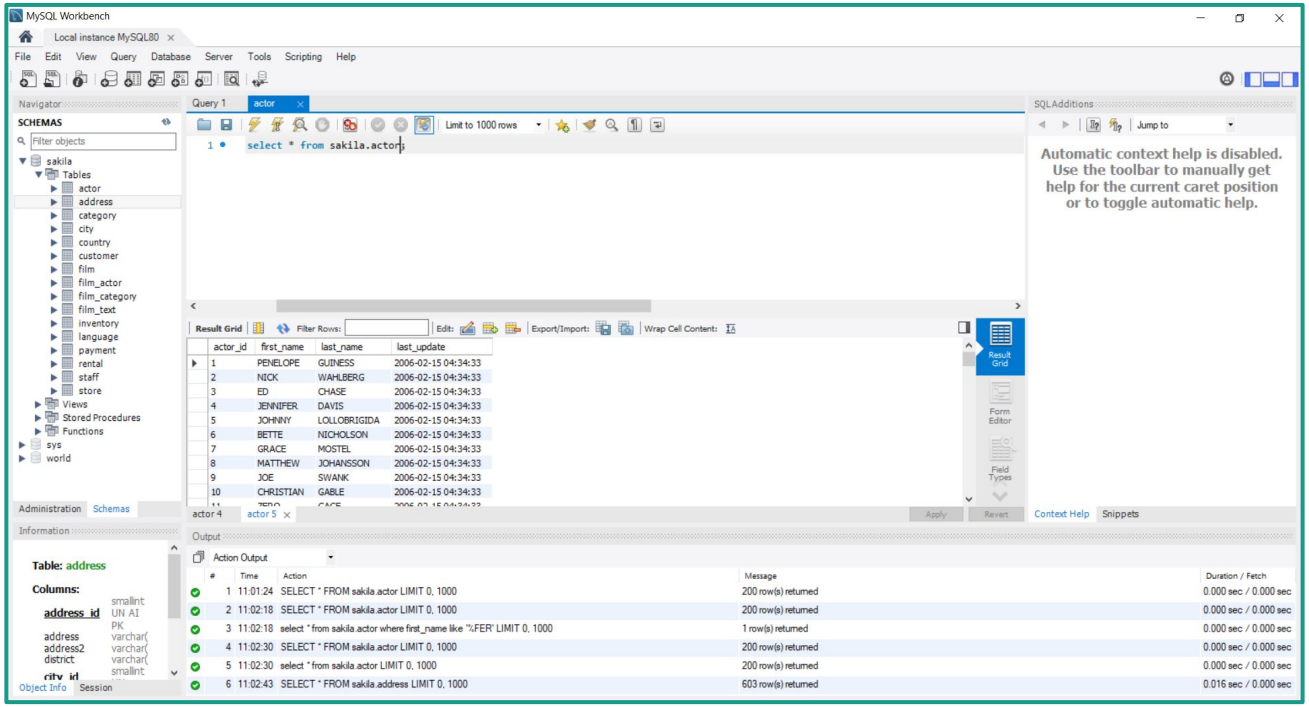
After clicking **Next** button a several time, you will receive to connection test page. Provide the root password you defined in previous steps and click the **Check** button to test you are able to connect to the DB.



Installing MySQL

When installation is finished, MySQL Workbench automatically will be opened. You should see a window like this. ➡

This means we successfully finished the MySQL installation.



Java – JDBC Connection

Handling a connection requires following steps:

1) Load JDBC driver

2) Open DB connection

3) Close DB connection

Load JDBC Driver

First, we need to download MySQL JDBC driver. You can download it here. The easiest way to do this is to use **Class.forName()** on the class that implements the java.sql.Driver interface. With MySQL Connector/J, the name of this class is **com.mysql.jdbc.Driver.** With this method, you could use an external configuration file to supply the driver class name and driver parameters to use when connecting to a database.

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

Load JDBC Driver

The easiest way to do this is to use **Class.forName()** on the class that implements the java.sql.Driver interface. With MySQL Connector/J, the name of this class is **com.mysql.jdbc.Driver.** With this method, you could use an external configuration file to supply the driver class name and driver parameters to use when connecting to a database.

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

Example:

```
try {
      Class.forName("com.mysql.cj.jdbc.Driver");
      System.out.println("Driver is found successfully");
}
catch (ClassNotFoundException e){
    System.out.println("The JDBC driver for MySQL not found!");
    return;
}
```

Open DB Connection

After the driver has been registered with the DriverManager, you can obtain a Connection instance that is connected to a particular database by calling **DriverManager.getConnection()**. This method automatically loads JDBC drivers in the classpath.

Syntax:

```
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/", "root", "password");
```

Further reading: https://docs.oracle.com/javase/tutorial/jdbc/basics/connecting.html

Open DB Connection

Example:

```
Connection mySqlConnection =  null;
try {
     mySqlConnection = DriverManager.getConnection("jdbc:mysql://localhost:3306/"
+ schema, "root", "password");
     System.out.printf("Successfully connected to the database and schema: %s",
schema);
} catch (SQLException e) {
      System.out.println("Unable to connect to the DB at the specified
address!");
      return;
}
```

Further reading: https://docs.oracle.com/javase/tutorial/jdbc/basics/connecting.html

JDBC Query Examples

SQL SELECT, INSERT, UPDATE or DELETE queries are executed to fetch, add, change or remove data stored in relational databases. It requires following steps:

1) Make a database connection

2) Execute the SQL Query

3) Fetch the data from result set

JDBC Select Query Examples

As a main step, it requires creating a Statement object and then using it's **executeQuery()** method.

```
Statement statement = mySqlConnection.createStatement();
ResultSet selectResultSet = statement.executeQuery("SELECT * FROM CITY WHERE
CITY_ID <= 10");

while (selectResultSet.next()){
    System.out.println("--------------------------------------");
    System.out.println("City Id: " + selectResultSet.getString(1));
    System.out.println("City: " + selectResultSet.getString(2));
    System.out.println("Country Id: " + selectResultSet.getString(3));
    System.out.println("Last Update Date: " + selectResultSet.getString(4));
}
```

You can use various **getXXX() (such as getInt(), getDate(), getBoolean() etc.)** methods available in ResultSet. But if you want to make it generic, then use **getString()** method and parse the data as and when needed.

JDBC Insert Example

Example:

```
Connection connection =  null;
Statement stmt = null;
try{
    Class.forName("com.mysql.jdbc.Driver");
    connection = DriverManager
                 .getConnection("jdbc:mysql://localhost:3306/TestSchema", "root",
"password");

stmt = connection.createStatement();
stmt.execute("INSERT INTO EMPLOYEE (ID,FIRST_NAME,LAST_NAME,AGE) "
                                + "VALUES (1,'Ahmet','Yılmaz',5)");
}
catch(SQLException e){
    e.printStackTrace();
}
```

JDBC Update Example

Example:

```
String sql = "UPDATE CITY SET CITY = 'Metro City', LAST_UPDATE = '2021-07-16‟ +
“14:40:58' WHERE CITY_ID = 999”;
         ResultSet rs = stmt.executeQuery(QUERY);
         while(rs.next()){
            //Display values
            System.out.print("ID: " + rs.getInt("id"));
            System.out.print(", Age: " + rs.getInt("age"));
            System.out.print(", First: " + rs.getString("first"));
            System.out.println(", Last: " + rs.getString("last"));
         }

```

JDBC Delete Example

Example:

```
connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/JDBCDemo",
"root", "password");

stmt = connection.createStatement();
stmt.execute("DELETE FROM EMPLOYEE WHERE ID >= 1");
```

JDBC Prepared Statements

In database management systems, a prepared statement or parameterized statement is a feature used to execute the same or similar database statements repeatedly with high efficiency. Typically used with SQL statements such as queries or updates, the prepared statement takes the form of a template into which certain constant values are substituted during each execution.

Advantages of Prepared Statements

- Pre-compilation and DB-side caching of the SQL statement leads to overall faster execution and the ability to reuse the same SQL statement in batches.
- **Automatic prevention of SQL injection attacks by built-in escaping of quotes and other special characters. Note that this requires that you use any of the PreparedStatement setXxx() methods to set the values and not use inline the values in the SQL string by string concatenation.**
- Apart from above two main usage, prepared statements makes it easy to work with complex objects like BLOBs and CLOBs.

Advantages of Prepared Statements

Example:

```
PreparedStatement pstmt = connection.prepareStatement(sql);
pstmt.setInt(1, 87);
pstmt.setString(2, "Ahmet");
pstmt.setString(3, "Yılmaz");
pstmt.setInt(4, 5);
pstmt= preparedStatement.executeQuery();
```

Questions ?

Next:Maven Build Tool