**Guided Lab 305.4.1A**

**Hibernate Project Demonstration**

**Lab Overview:**

In this lab, we will walk you through creating our first CRUD application with Hibernate. When developing hibernate applications, we need to provide two sets of configuration in the hibernate.cfg.xml file.

* The first set of configuration contains database specific properties that will be used to create Database connections and Session objects.
* The second set of configurations contains mappings between model classes and database tables.

**Learning Objective:**

At the end of this lab, you will be able to develop **Hibernate** applications.

**Instructions:**

## **Manually Create an SQL Database**

**Use the following statement to create a database named usersdb.**

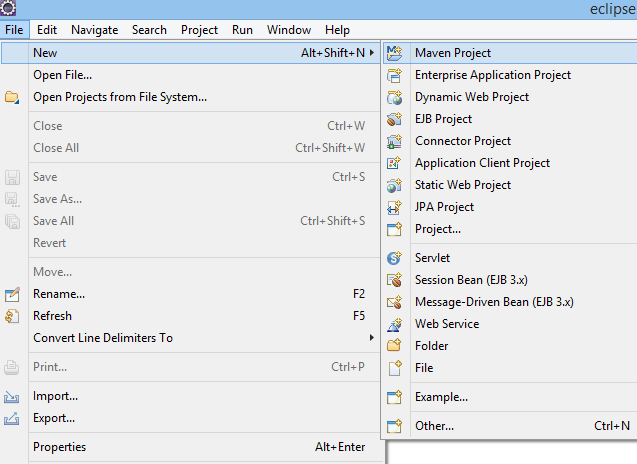
| **create database usersdb;** |
| --- |

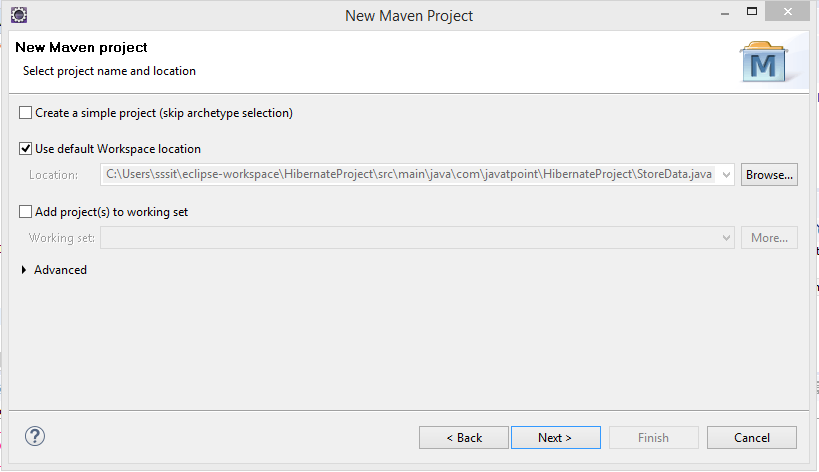
## 

## **Setup a Java Maven Project**

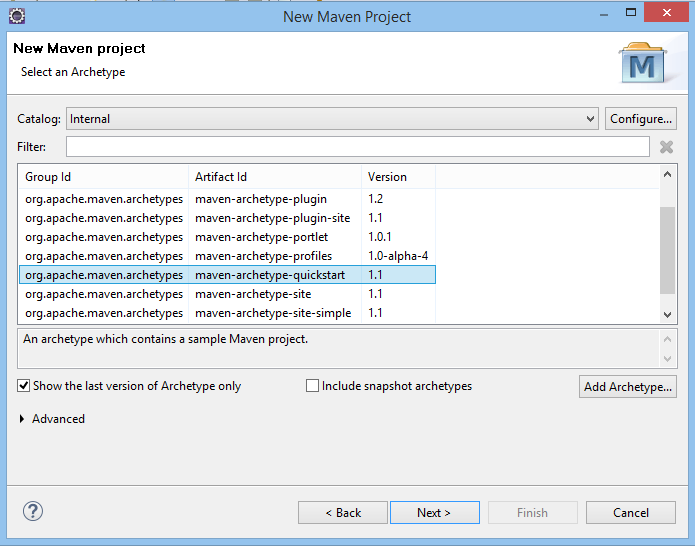
* We are going to create a Maven-based Hibernate application using annotations in the IDE.
* To create the Hibernate application in the IDE, we need to follow the below steps:

### 1) Create the Maven Project

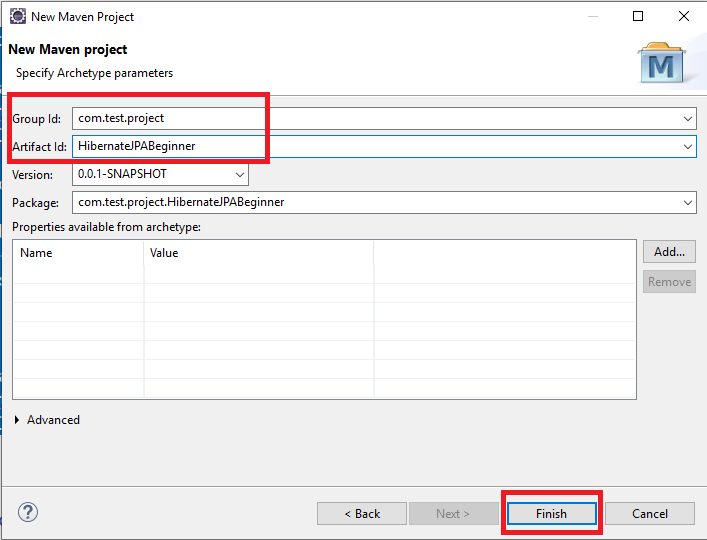
* To create the Maven project, left-click on **File → New → Maven Project,** as shown in the screenshot below: ****
* The new Maven project will open in Eclipse. Click **Next,** as shown in the screenshot below:

****

* **Now, select the catalog type: Internal.**
* **Artifact ID: *maven archetype - quickstart of the 1.1 version.***
* **Click on the *Next* button as shown in the screenshot below:**

****

* **Now, specify the name of the *Group Id* and the *Artifact Id.*** 
  1. **The Group Id contains a package name (e.g., com.test.project).**
  2. **The Artifact Id contains a project name (e.g., HibernateJPABeginner).** 
     + **Then click Finish, as shown in the screenshot below:**

****

### 

### 

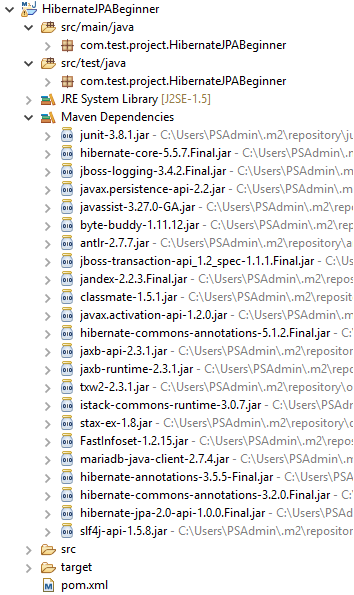
### 2) Add Jar dependencies, and configuration in the pom.xml file.

**Next, we need to add a couple of jar dependencies for Hibernate, JPA, and MySQL Connector Java in the pom.xml file of our Maven Project.**

**Open the pom.xml file and insert the following XML code under the <dependencies> </dependencies> tag just before the </project> tag:**

| | **<dependencies>**  **<dependency>  <groupId>org.hibernate</groupId>  <artifactId>hibernate-core</artifactId>  <version>5.5.7.Final</version> </dependency> *<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->***  **<dependency>**  **<groupId>mysql</groupId>**  **<artifactId>mysql-connector-java</artifactId>**  **<version>8.0.25</version>**  **</dependency>**  **</dependencies>** | | --- | |
| --- | --- |

**Here, we added two dependencies for the project: hibernate-core and MySQL-connection. Maven automatically downloads the required JAR files, which are shown under the *Maven Dependencies* node in the project:**

****

### 3). Create the Persistence Class (Model Class or Pojo).

* **Create a package “com.test.hib.model” under the *main->src->Java***
* **Create an entity class named “User.java” under the above package.**
* **Then we will use annotations to map this table to the corresponding table in the database.**
* **Add the following initial code in the User.java class:**

| **public class User {  private Integer id;  private String fullname;  private String email;  private String password;**  **private int age;**  **private double salary;**  **private String city;**  **public User(String fullname, String email, String password, int age, double salary, String city) {  this.fullname = fullname;  this.email = email;  this.password = password;  this.age = age;  this.salary = salary;  this.city = city;  }  public User() {   }**  **public Integer getId() {  return id;  }    public void setId(Integer id) {  this.id = id;  }    public String getFullname() {  return fullname;  }    public void setFullname(String fullname) {  this.fullname = fullname;  }    public String getEmail() {  return email;  }    public void setEmail(String email) {  this.email = email;  }    public String getPassword() {  return password;  }    public void setPassword(String password) {  this.password = password;  }**  **public int getAge() {  return age;  }   public void setAge(int age) {  this.age = age;  }   public double getSalary() {  return salary;  }   public void setSalary(double salary) {  this.salary = salary;  }   public String getCity() {  return city;  }   public void setCity(String city) {  this.city = city;  }**  **}** |
| --- |

**You can see in the code above that this is just a POJO (Plain Old Java Object) class with some class variables, getter and setter methods, and its constructors.**

**Now, let’s use some annotations provided by JPA to map this model class to the user table in the database.**

| **package com.test.hib.model;**  **import javax.persistence.\*;**  **@Entity**  **@Table(name = "USER")**  **public class User {**  **@Column(name = "USER\_ID")**  **@Id**  **@GeneratedValue(strategy = GenerationType.IDENTITY)**  **private Integer id;**  **private String fullname;**  **private String email;**  **private String password;**  **private int age;**  **private double salary;**  **private String city;**    **public Integer getId() {**  **return id;**  **}**  **public void setId(Integer id) {**  **this.id = id;**  **}**  **public String getFullname() {**  **return fullname;**  **}**  **public void setFullname(String fullname) {**  **this.fullname = fullname;**  **}**  **public String getEmail() {**  **return email;**  **}**  **public void setEmail(String email) {**  **this.email = email;**  **}**  **public String getPassword() {**  **return password;**  **}**  **public void setPassword(String password) {**  **this.password = password;**  **}**  **public int getAge() {**  **return age;**  **}**  **public void setAge(int age) {**  **this.age = age;**  **}**  **public double getSalary() {**  **return salary;**  **}**  **public void setSalary(double salary) {**  **this.salary = salary;**  **}**  **public String getCity() {**  **return city;**  **}**  **public void setCity(String city) {**  **this.city = city;**  **}**  **}** |
| --- |

* **@Entity annotation specifies that the class is an entity.**
* **@Table maps the entity with the table. If no @Table is defined, the default value is used: the class name of the entity.**
* **@Id declares the identifier property of the entity.**
* **@Column maps the entity's field to the table's column. If @Column is omitted, the default value is used: the field name of the entity.**

### 

### 4) Create the Configuration File

* **For Eclipse IDE**: To create the configuration file, right-click on **src/main/java** → **New → Other - search files from search panel → click on File → specify the file nam**e **“**hibernate.cfg.xml**”** → **Finish**. Write the following code.
* **For Intellij IDE**: Create a configuration file named hibernate.cfg.xml under the resources folder, and write the following code.

**hibernate.cfg.xml**

| <?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE hibernate-configuration PUBLIC  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd"> <hibernate-configuration> <session-factory>  <!-- Drop and re-create the database on startup -->  <property name="hibernate.hbm2ddl.auto"> update </property>  <!-- Database connection settings -->  <property name="connection.driver\_class">com.mysql.cj.jdbc.Driver</property>  <property name="connection.url">jdbc:mysql://localhost:3306/usersdb</property>  <property name="connection.username">root</property>  <property name="connection.password">password</property>  <!-- MySQL DB dialect -->  <property name="dialect">org.hibernate.dialect.MySQL5Dialect</property>  <!-- print all executed SQL on console -->  <property name="hibernate.show\_sql" >true </property>  <property name="hibernate.format\_sql" >true </property>    <!-- Mapping entity file -->  <mapping class="com.test.hib.model.User"/>  </session-factory> </hibernate-configuration> |
| --- |

*Hibernate has many* [*configuration properties*](https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/session-configuration.html)*. Apart from the standard connection properties, it is worth mentioning the dialect property, which allows us to specify the name of the SQL dialect for the database.*

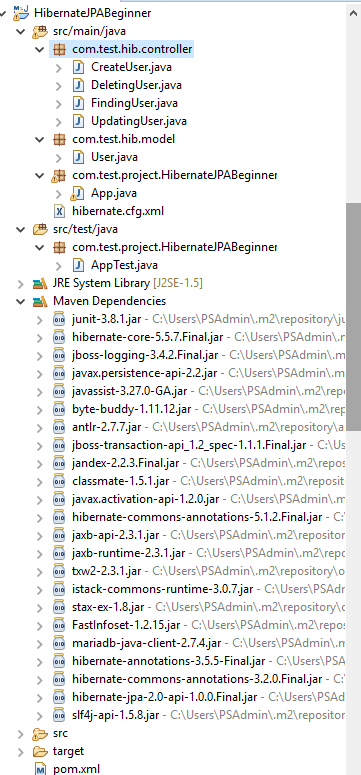
[Click here for reference document for Hibernate XML Config → *https://www.onlinetutorialspoint.com/hibernate/hbm2ddl-auto-example-hibernate-xml-config.html*](https://www.onlinetutorialspoint.com/hibernate/hbm2ddl-auto-example-hibernate-xml-config.html)

### 5) Develop CRUD Operations

We will use this application to save a few user records, and then we will apply CRUD operations to those records.

Create a package called **com.test.hib.controller.** Then create the below classes in that package:

* CreateUserTable.java
* CreateUser.java
* FindingUser.java
* UpdatingUser.java
* DeletingUser.java



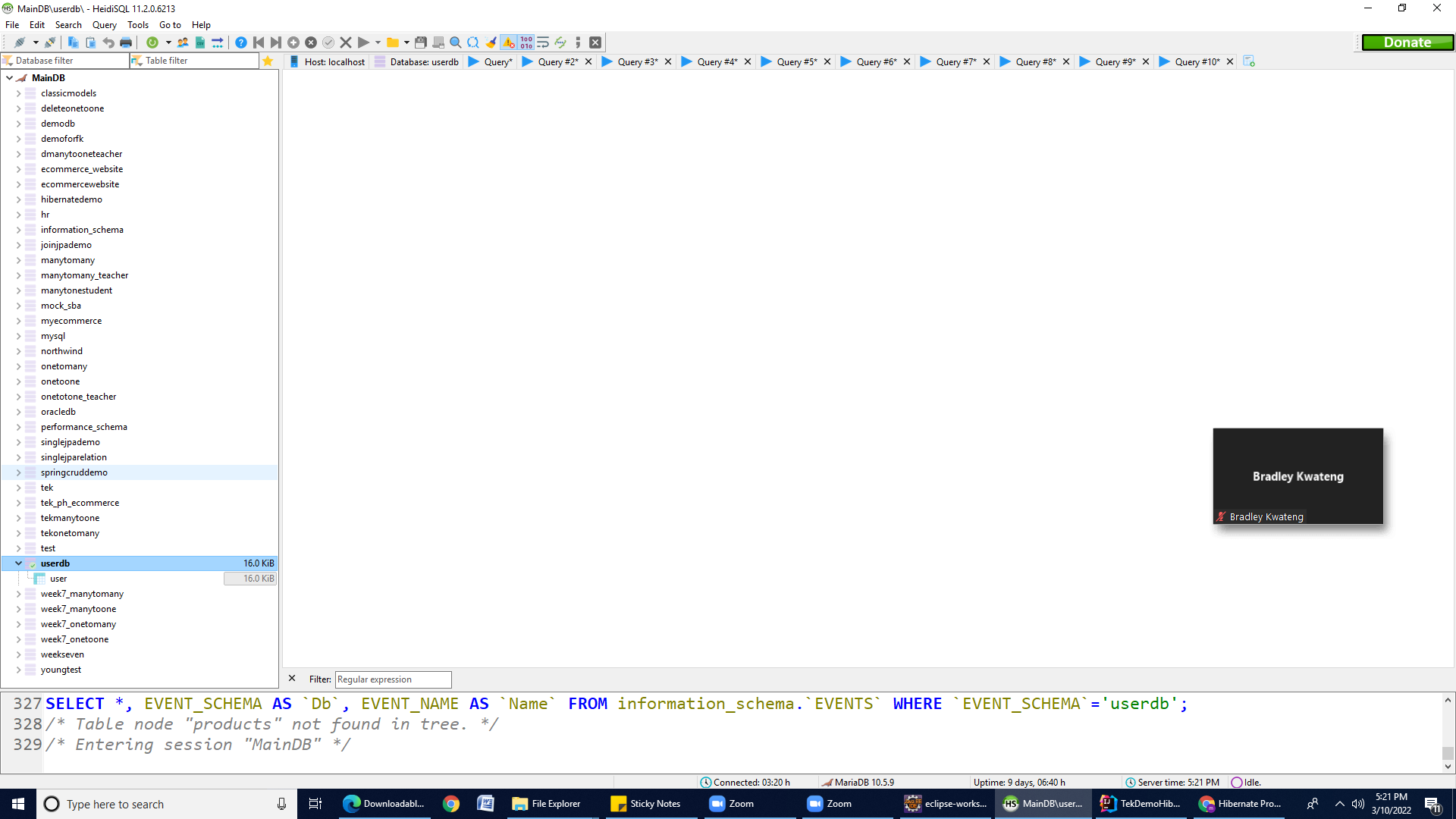
## **5.1) Create a User Table using Hibernate.**

Add the following code to the “CreateUserTable” class:

| import org.hibernate.Session; import org.hibernate.SessionFactory; import org.hibernate.Transaction; import org.hibernate.cfg.Configuration; public class CreateUserTable {  public static void main(String[] args) {  SessionFactory factory = new Configuration().configure().buildSessionFactory();  Session session = factory.openSession();  Transaction t = session.beginTransaction();  User uone = new User();  t.commit();  System.out.println("successfully created user table");  factory.close();  session.close();  } } |
| --- |

## **"Run an Application"**

Run the CreateUserTable class. The **user** table should be generated/created in the **userdb** Database:



## 

## 

## **5.2) Adding User Records to the Database**

Now, let’s write some code to create user entity instances using JPA. To do so,

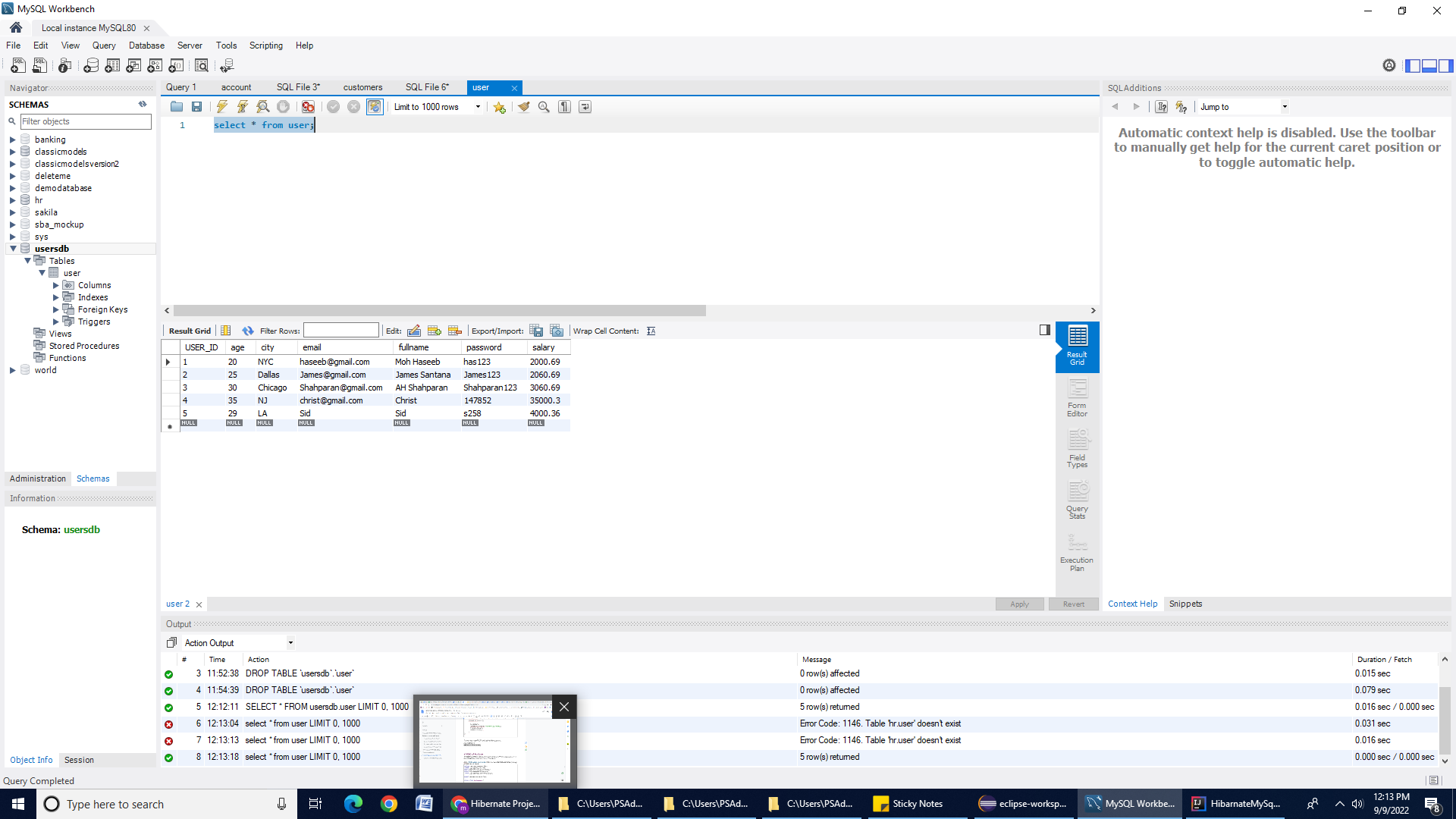
Open the **CreateUser.java** class, which contains the main method(). Add the following Java code:

| **package com.test.hib.controller; import org.hibernate.Session; import org.hibernate.SessionFactory; import org.hibernate.Transaction; import org.hibernate.cfg.Configuration;  public class CreateUser {  public static void main(String[] args) {  // TODO Auto-generated method stub SessionFactory factory = new Configuration().configure().buildSessionFactory();  Session session = factory.openSession();    Transaction t = session.beginTransaction();   User uone = new User();  uone.setEmail("haseeb@gmail.com");  uone.setFullname("Moh Haseeb");  uone.setPassword("has123");  uone.setSalary(2000.69);  uone.setAge(20);  uone.setCity("NYC");    User uTwo = new User();  uTwo.setEmail("James@gmail.com");  uTwo.setFullname("James Santana");  uTwo.setPassword("James123");  uTwo.setSalary(2060.69);  uTwo.setAge(25);  uTwo.setCity("Dallas");    User uThree = new User();  uThree.setEmail("Shahparan@gmail.com");  uThree.setFullname("AH Shahparan");  uThree.setPassword("Shahparan123");  uThree.setSalary(3060.69);  uThree.setAge(30);  uThree.setCity("Chicago");    /\*========= We can pass value/data by using constructor =========\*/  User ufour = new User( "Christ", "christ@gmail.com", "147852", 35, 35000.3,"NJ");  User ufive = new User("Sid", "Sid", "s258", 29, 4000.36, "LA");  *//Integer userid = null;*  session.persist(uone);   session.persist(uTwo);   session.persist(uThree);   session.persist(ufour);   session.persist(ufive);     t.commit();  System.out.println("successfully saved");   factory.close();   session.close();   } }** |
| --- |

For the output: Open MySQL and type the following queries.

| use userdb ;  select \* from user; |
| --- |

**Result:**



### 

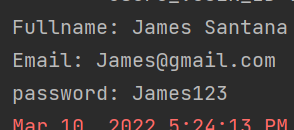
### 6) READ all the Users

The **session.get(Class, id)** returns an object of the specified class that maps a row in the database table. If no row is found, it returns null.

Open the **FindingUser.java class** under the **com.test.hib.controller** package and add the code below:

| **package com.test.hib.controller; import org.hibernate.Session; import org.hibernate.SessionFactory; import org.hibernate.Transaction; import org.hibernate.cfg.Configuration; import com.test.hib.model.User;  public class FindingUser {  public static void main(String[] args) { SessionFactory factory = new Configuration().configure().buildSessionFactory();    Session session = factory.openSession();  Transaction tx = session.beginTransaction();    int USER\_ID = 2;  User u = session.load(User.class, USER\_ID);  System.out.println("Fullname: " + u.getFullname());  System.out.println("Email: " + u.getEmail());  System.out.println("password: " + u.getPassword());**  **//Close resources  tx.commit();  factory.close();  session.close();  } }** |
| --- |

Run the **FindingUser.java** class. You will see the result on your console as shown below:



### 7) Updating User record

**Session.update() and session.merge()** are both used for updating the rows of data in the database.

**Session.update()**: If you are sure that the session does not contain an already persistent instance with the same identifier, use **update** to save the data in hibernate.

**Session.merge():** If you want to save your modifications at any time without knowing about the state of a session, use merge() in hibernate.

Open the **UpdatingUser.java class** under the **com.test.hib.controller** package and add the code below:

| **package com.test.hib.controller; import org.hibernate.Session; import org.hibernate.SessionFactory; import org.hibernate.Transaction; import org.hibernate.cfg.Configuration; import com.test.hib.model.User;  public class UpdatingUser {  public static void main(String[] args) {  SessionFactory factory = new Configuration().configure().buildSessionFactory();  Session session = factory.openSession();  Transaction tx = session.beginTransaction();  User u = new User();**  **u.setId(3);  u.setEmail("mhaseeb@perscholas");  u.setFullname("M haseeb");  u.setPassword("123456");  session.merge(u);  session.getTransaction().commit();  session.close();  } }** |
| --- |

Run the **UpdatingUser.java** class. Go back to the **userdb** database under the **User** table. You can find **USER\_ID**: 3 must be updated.

### 8) Performing DELETE operation:

We will call the **session.remove(Object)** method to remove a mapped object from the database.

Open the **DeletingUser.java class** under the **com.test.hib.controller** package and add the code below:

| **package com.test.hib.controller; import org.hibernate.Session; import org.hibernate.SessionFactory; import org.hibernate.Transaction; import org.hibernate.cfg.Configuration; import com.test.hib.model.User;  public class DeletingUser {  public static void main(String[] args) { SessionFactory factory = new Configuration().configure().buildSessionFactory();  Session session = factory.openSession();  Transaction tx = session.beginTransaction();  User u = new User();  u.setId(3);  session.remove(u);  tx.commit();  session.close();  factory.close();  }  }** |
| --- |

Run the **DeletingUser.java** class. Then go back to the **userdb** database under the **User** table. The **USER\_ID**:**3** record must be deleted.

**Reference:**

<https://www.objectdb.com/api/java/jpa/Query>

<https://stackoverflow.com/questions/37066024/what-is-the-mariadb-dialect-class-name-for-hibernate>

<https://docs.jboss.org/hibernate/core/3.3/reference/en-US/html/objectstate.html>

**Submission Instructions:**

Include the following deliverables in your submission -

* + Submit your source code or screenshot using the Start Assignment button in the top-right corner of the Assignment page in Canvas.

## 