JPA with Hibernate

Hibernate – ORM Framework

ORM – Object Relational Mapping (JAVA-DB Connecting)

JPA – Java Persistence API

Persistence – Storing the state of an object into a db entity.

Serialization – Storing the state of the object into a flat file(.txt,.rtf,.doc)

<https://hibernate.org/> -- Official Site of Hibernate

The Latest version of Hibernate is : 5.6 (Stable) 6.0(latest) [on 18 Oct 2021]

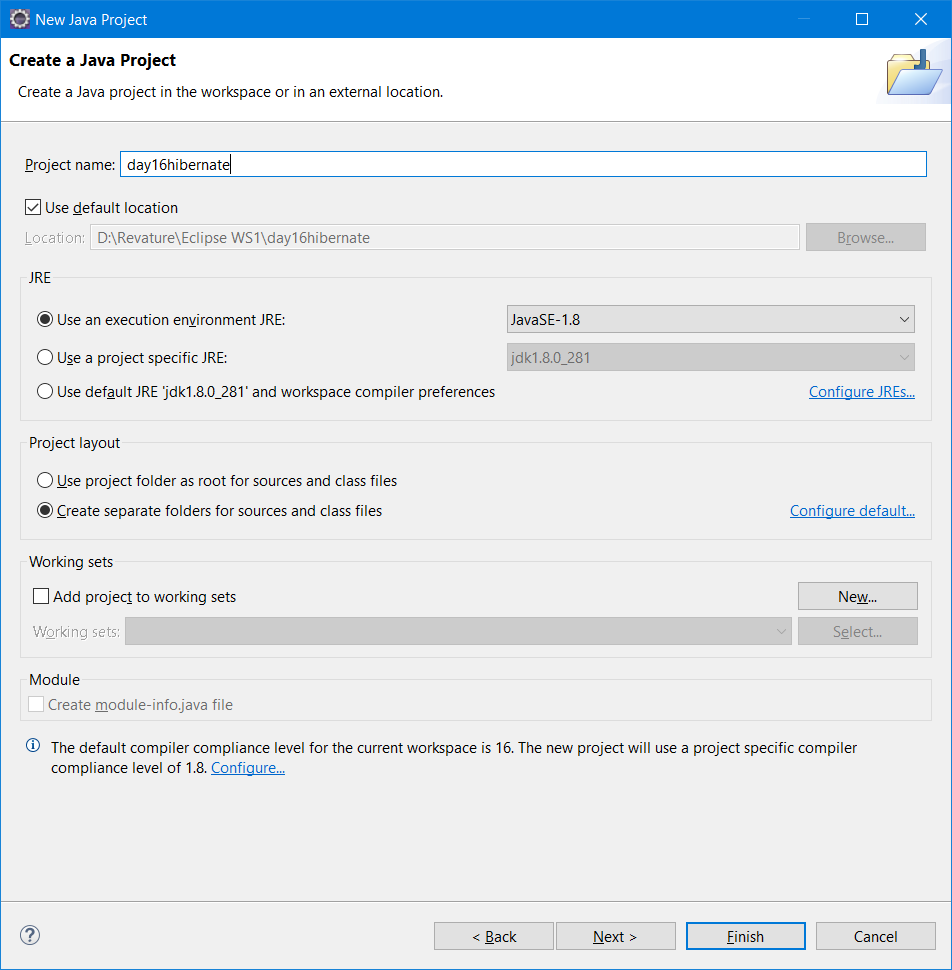
Features

1. Connection Pooling (Hikari Pool)
2. DB Caching
3. HQL (Hibernate Query Lang) [DB independent queries]
4. Scaling

<https://hibernate.org/orm/documentation/getting-started/>

<https://www.javatpoint.com/steps-to-create-first-hibernate-application>

Step 1: Open Eclipse IDE and create a new Java Project [File->New->Project (Java Project)]

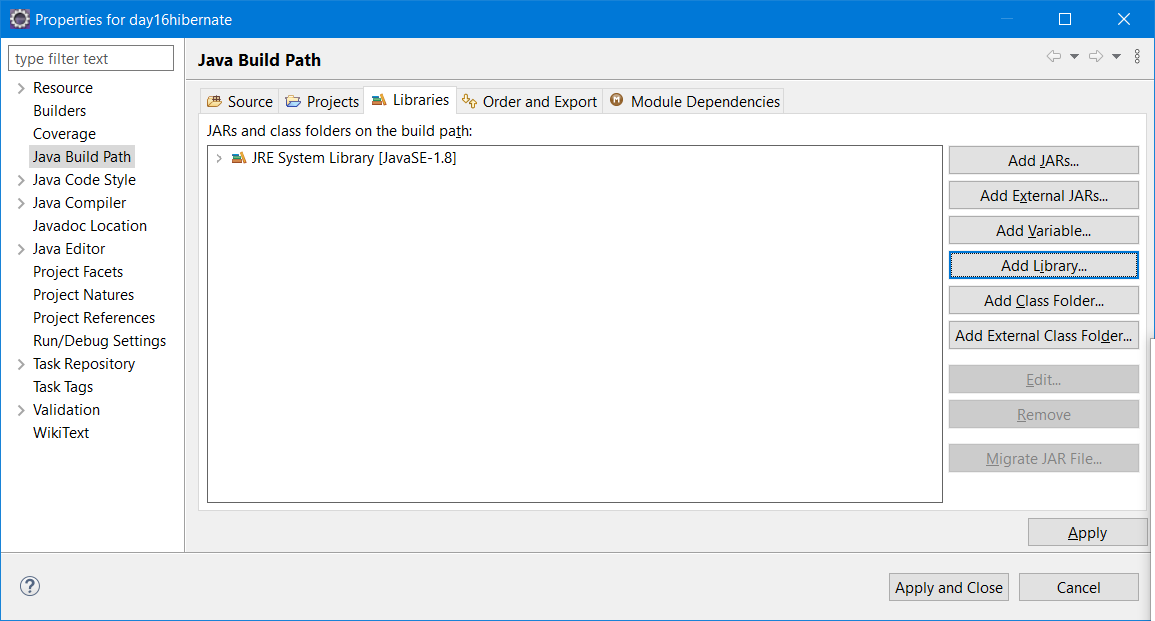


Step 2: Adding the hibernate dependencies to the build path

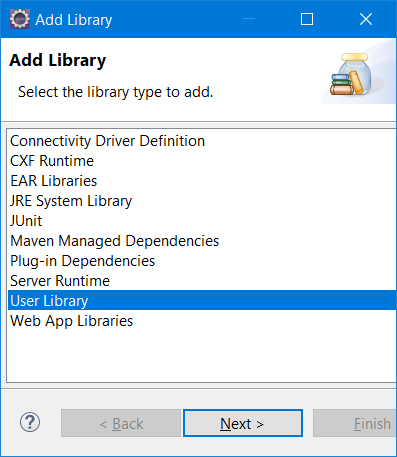
Download the necessary jars using the following link <https://www.javatpoint.com/src/hb/hibernatejar.zip>

Extract the content in a folder.

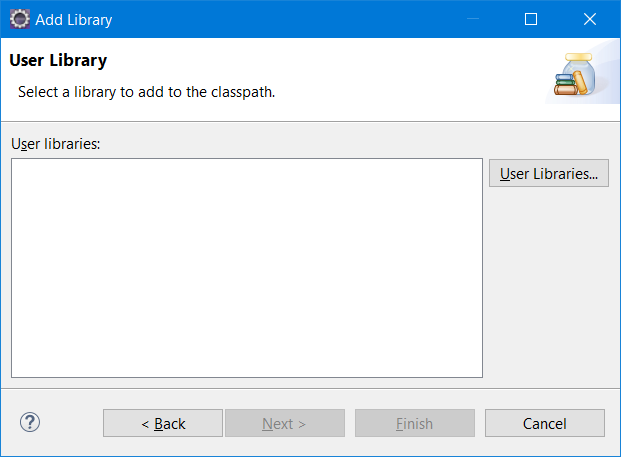
Right click Project ->Build Path -> Configure Build path ----🡪 select libraries tab



Step 3: Create a new “User Library” and add all the hibernate jars in it.

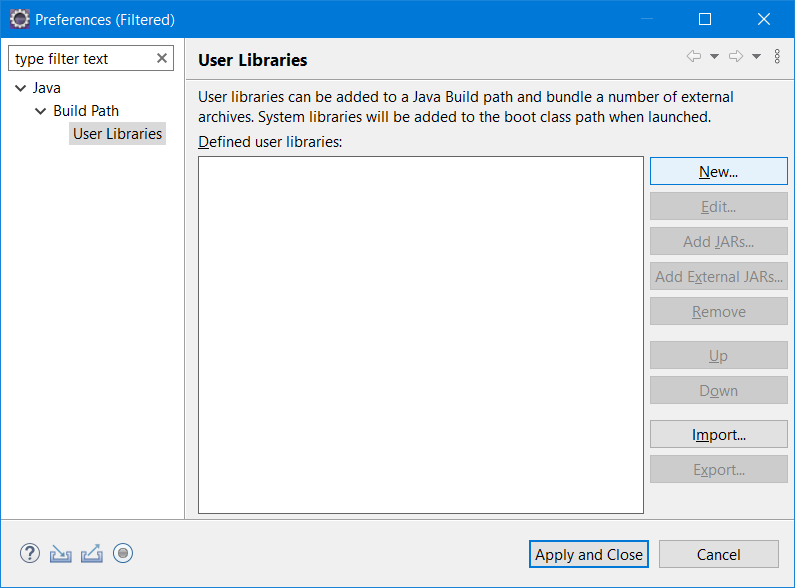
“Add Library” -> 

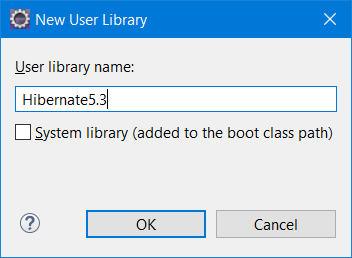
Select “User Library” option as shown above.



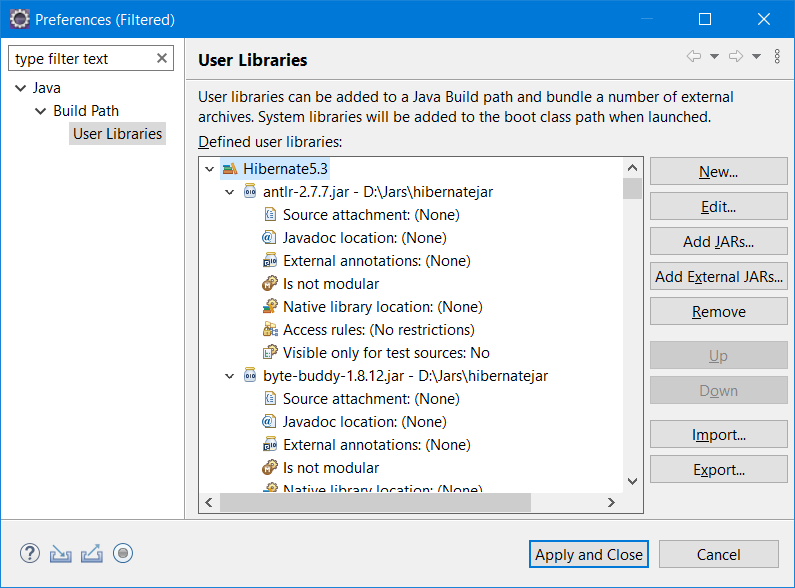
Click on “User Libraries” button.

And click on “New” button to create a new User Library.



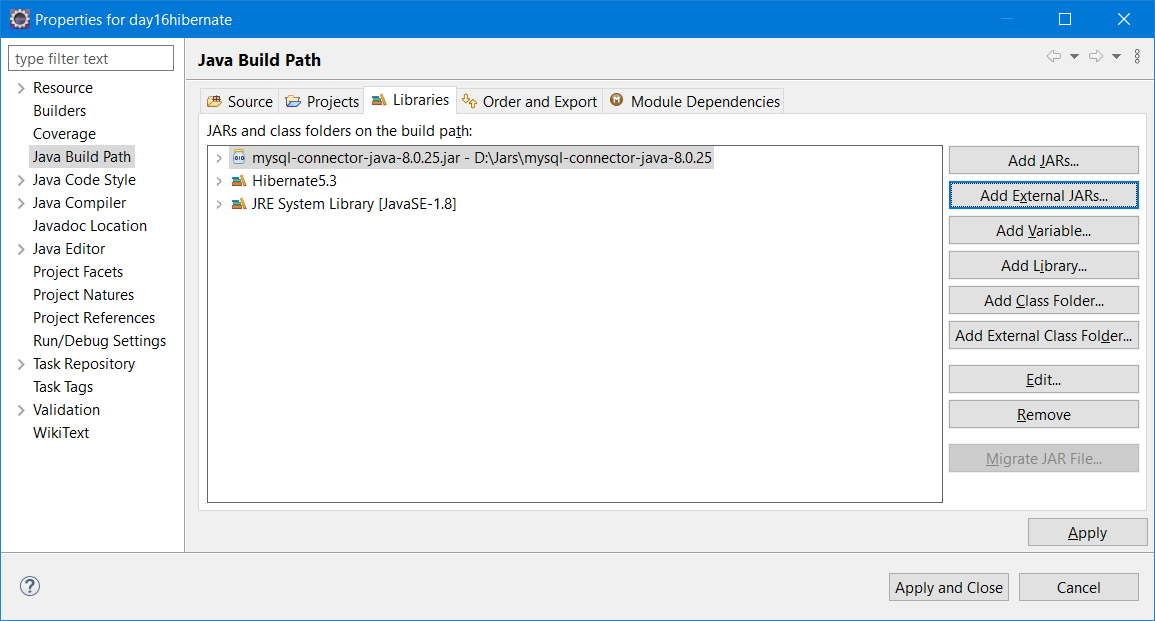


Click “Ok” and add “External Jars” button. Select all the hibernate jars that we downloaded earlier and click “Apply & Close” button.



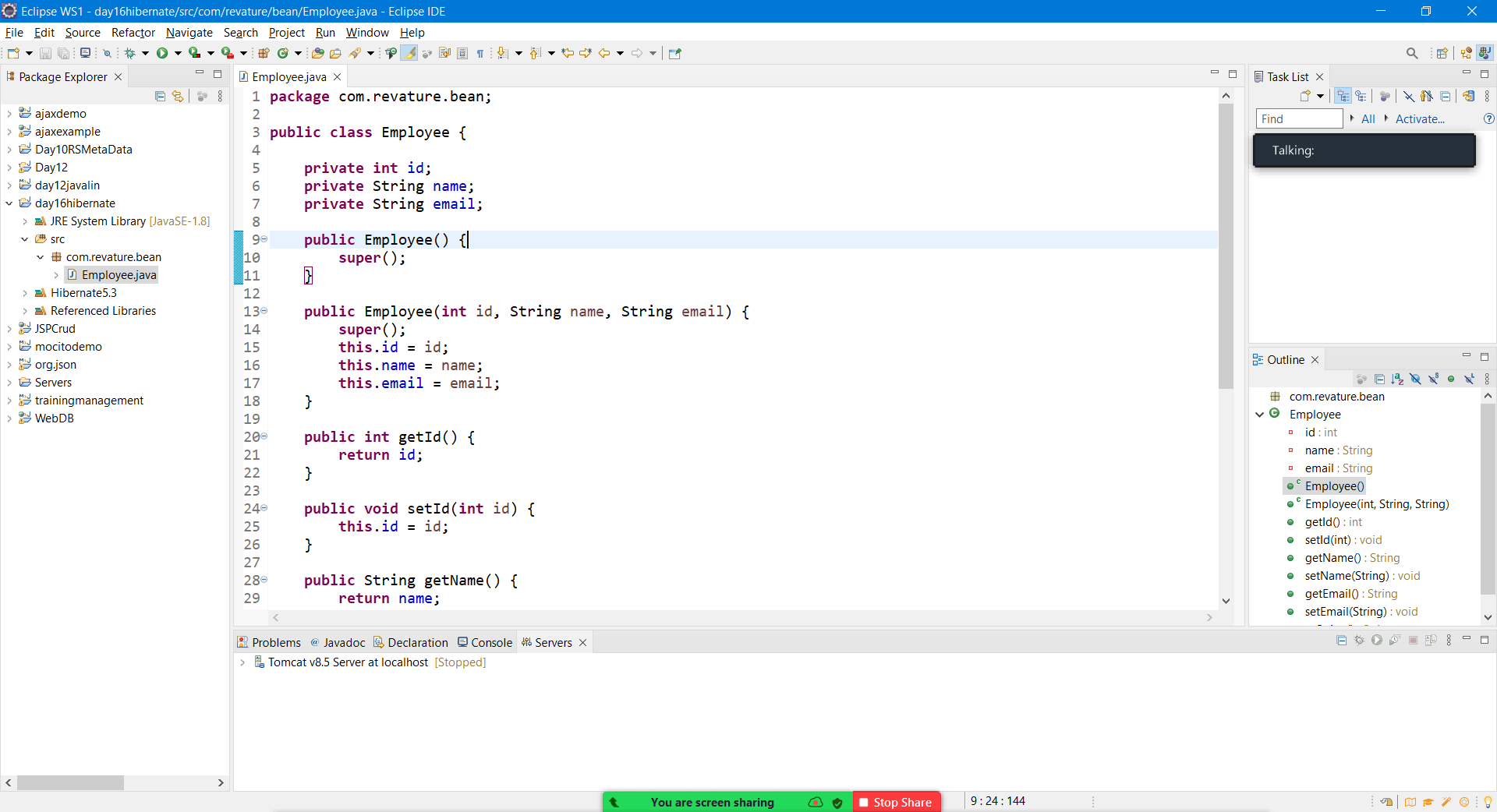
Then click on “Finish” & “Apply & Close”

Step 4: Add the MySQL driver jar to the build path.



Step 5: Create Entity Bean Class

Employee Entity Bean



1. By using the hibernate mapping file (bean.hbm.xml)
2. By Using the annotation @Entity, @Id, @Table, @Column

Step 6: Adding Hibernate Mapping File

Create a new xml file named employee.hbm.xml in “src” folder (Not in any sub folder)

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 5.3//EN" "http://hibernate.sourceforge.net/hibernate-mapping-5.3.dtd">

<hibernate-mapping>

<class name=*"com.revature.bean.Employee"* table=*"employee"*>

<id name=*"id"*>

<generator class=*"assigned"*></generator>

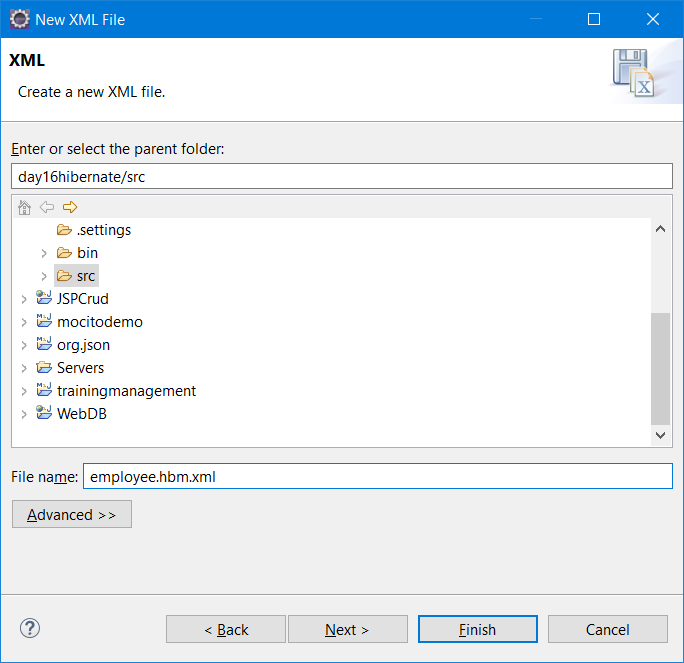
</id>

<property name=*"name"*></property>

<property name=*"email"*></property>

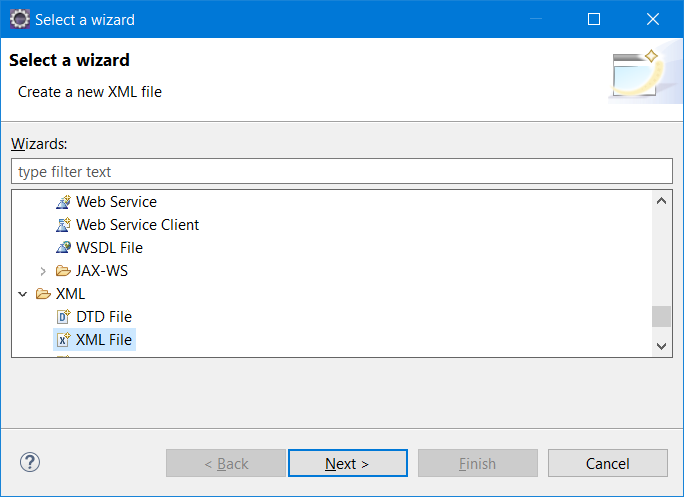
</class>

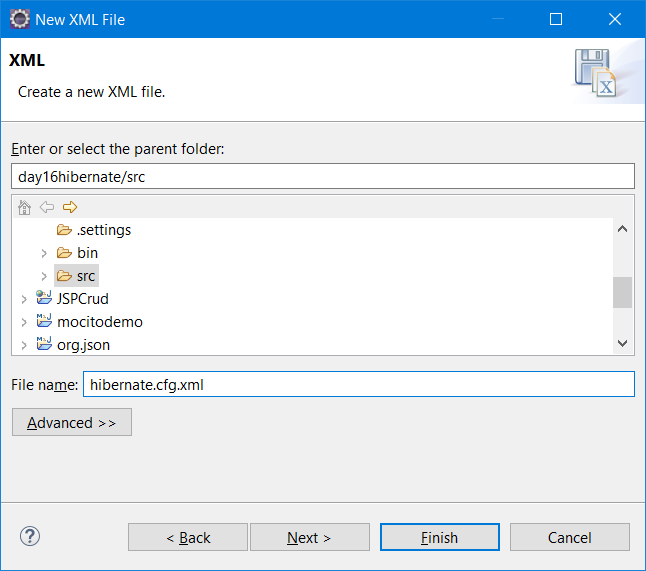
</hibernate-mapping>



Step 6: Adding hibernate.cfg.xml

Src->New->File->XML->xml file





<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 5.3//EN" "http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">

<hibernate-configuration>

<session-factory>

<property name=*"hbm2ddl.auto"*>update</property>

<property name=*"dialect"*>org.hibernate.dialect.MySQLDialect</property>

<property name=*"connection.url"*>jdbc:mysql://localhost:3306/revature</property>

<property name=*"connection.username"*>root</property>

<property name=*"connection.password"*>root</property>

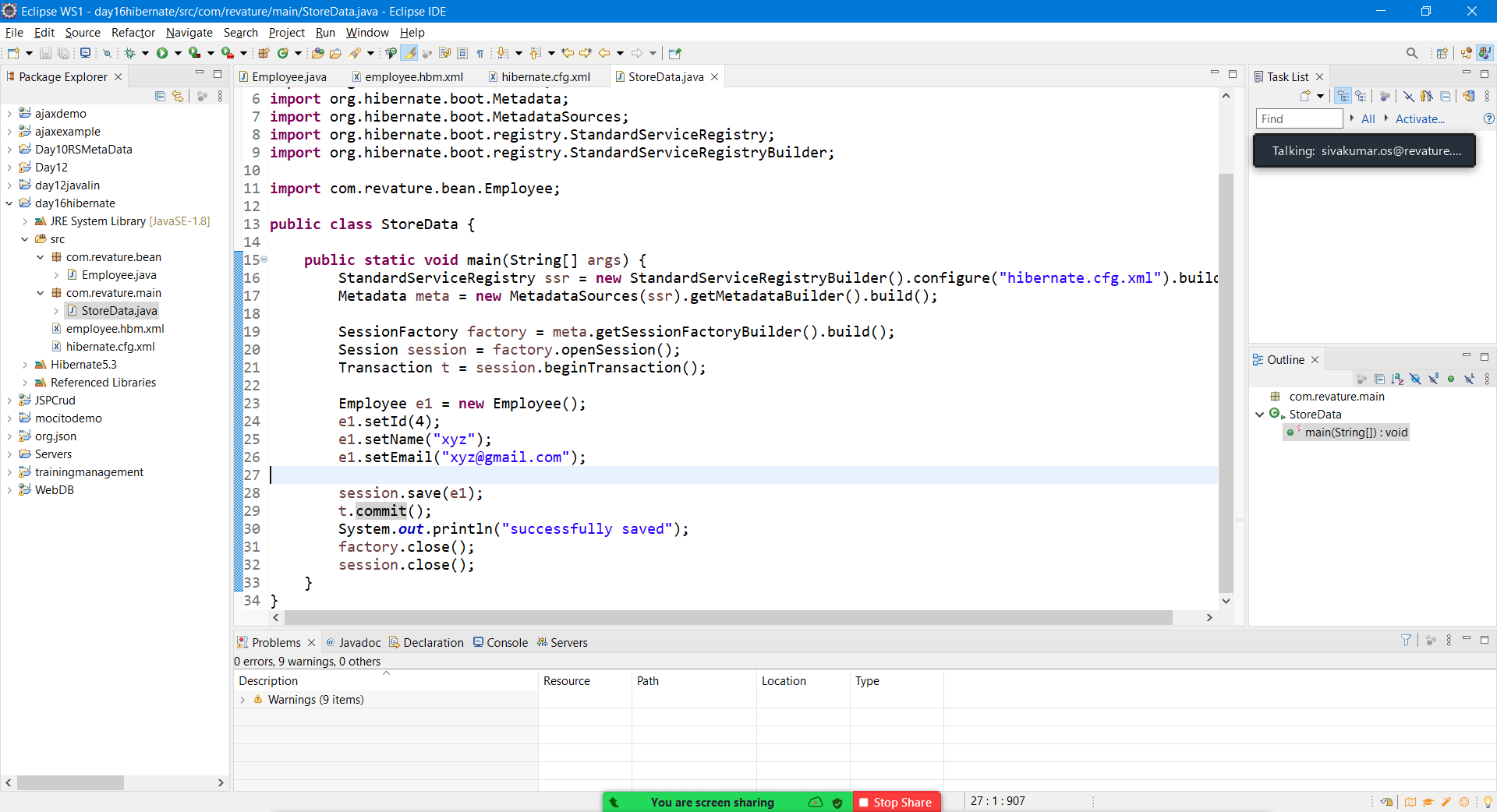
<property name=*"connection.driver\_class"*>com.mysql.cj.jdbc.Driver</property>

<mapping resource=*"employee.hbm.xml"* />

</session-factory>

</hibernate-configuration>

Step 7: Add the Starter class



**package** com.revature.main;

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.Transaction;

**import** org.hibernate.boot.Metadata;

**import** org.hibernate.boot.MetadataSources;

**import** org.hibernate.boot.registry.StandardServiceRegistry;

**import** org.hibernate.boot.registry.StandardServiceRegistryBuilder;

**import** com.revature.bean.Employee;

**public** **class** StoreData {

**public** **static** **void** main(String[] args) {

StandardServiceRegistry ssr = **new** StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();

Metadata meta = **new** MetadataSources(ssr).getMetadataBuilder().build();

SessionFactory factory = meta.getSessionFactoryBuilder().build();

Session session = factory.openSession();

Transaction t = session.beginTransaction();

Employee e1 = **new** Employee();

e1.setId(4);

e1.setName("xyz");

e1.setEmail("xyz@gmail.com");

session.save(e1);

t.commit();

System.***out***.println("successfully saved");

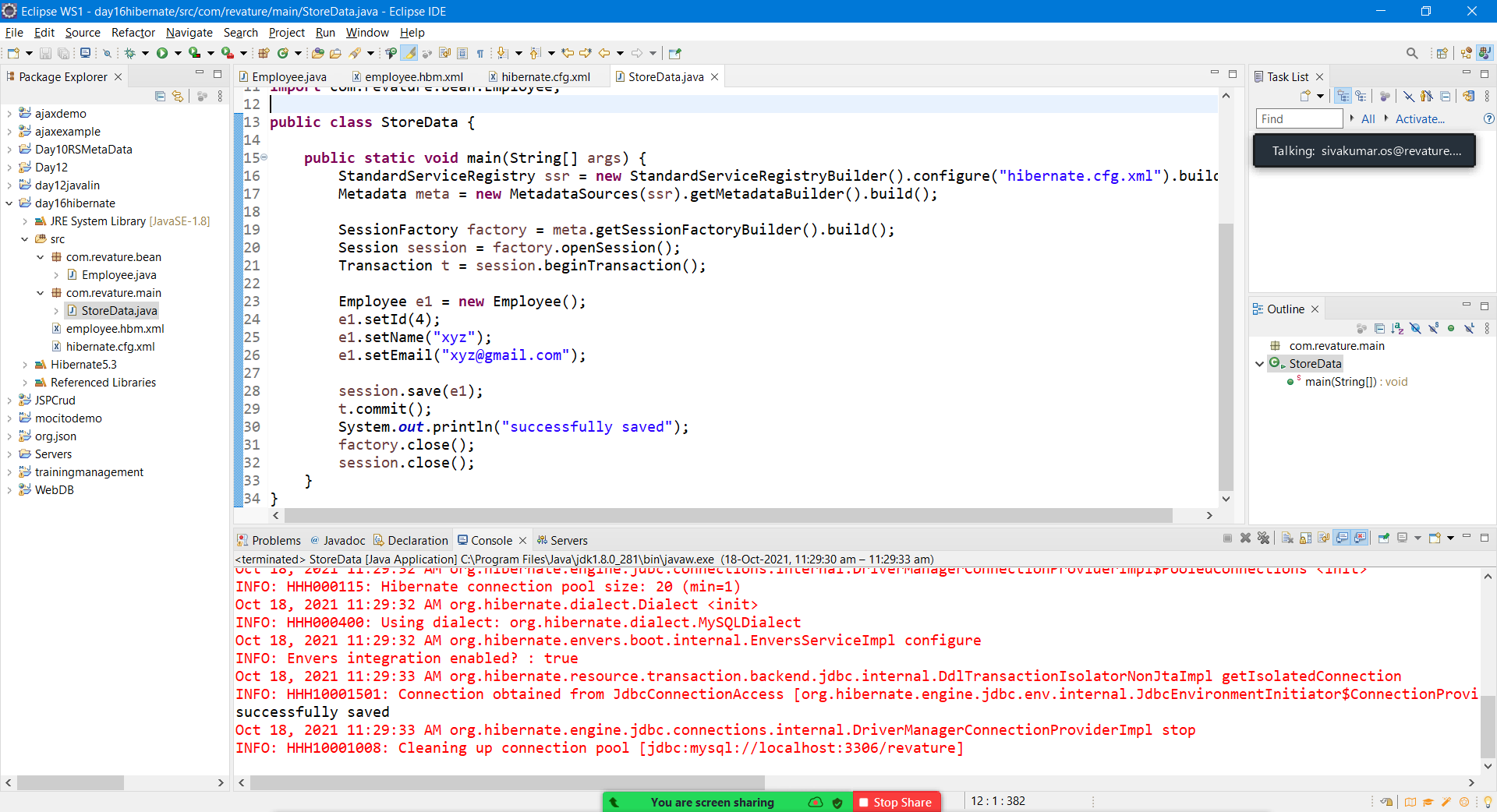
factory.close();

session.close();

}

}

Step 8: Run the Code



JPA – Java Persistance API

JPQL – Java Persistance Query Lang (Obj-Oriented Query Lang & DB Independent)

HQL – Hibernate Query Lang

|  |  |  |
| --- | --- | --- |
|  | JPQL | HQL |
| 1 | EntityManager (I) | Session (I) |
| 2 | EntityManagerFactory(I) | SessionFactory(I) |
| 3 | Transaction(I) | Transaction(I) |

DB Independent

JTA - Java Transaction API

JDBC – Java Database Connectivity

JNDI – Java Naming Directory Interface

Configure[C] – First Created in Hibernate

SessionFactory[I] (Immutable, II Level Cache, Thread-safe)

Session[I] (Not Thread-safe)

Transaction(I)

Query(I)

Criteria(I)

# Hibernate Configuration

The Hibernate configuration file contains the database mapping information that tells Hibernate how it should communicate with the database, and where the mapping for individual classes/tables can be found. These configurations are provided either in an XML file (which must be named hibernate.cfg.xml) or properties file (hibernate.properties).

## hibernate.cfg.xml file

The hibernate.cfg.xml file contains the database and session related configuration. Database configuration includes the JDBC connection URL, Database user credentials, driver class, and hibernate dialect.

This file will get loaded through the **Configuration** class. In the configuration class, we have a configure() method, which loads the hibernate.cfg.xml file and returns **SessionFactory** reference. The Session Factory reference used to get a **session** object.

**Example of hibernate.cfg.xml for a MySQL Database:**

<?xml version='1.0' encoding='UTF-8'?>

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<!-- SQL Dialect -->

<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

<!-- Database Connection Settings -->

<property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>

<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/test\_db?useSSL=false</property>

<property name="hibernate.connection.username">root</property>

<property name="hibernate.connection.password">root</property>

<!-- Connection Pool Size (built-in) -->

<property name="hibernate.connection.pool.size">1</property>

<!-- show all generate SQL query -->

<property name="show\_sql">true</property>

<!-- Drop the existing tables and create new one -->

<property name="hbm2ddl.auto">create</property>

<!-- Mention here all the model classes along with their package name -->

<mapping class="com.revature.hibernate.Student"/>

</session-factory>

</hibernate-configuration>

The hibernate.cfg.xml file has the <hibernate-configuration> element as a root element, which has a <session-factory> element as its child. The <session-factory> element contains a database and mapping information.

The <mapping> element lists a single java classe that is mapped to a table in the database. If a class is mapped using hibernate annotations, then the <mapping> element should use the class attribute; if the class is mapped using an XML file, the resource attribute should be used instead. You should use one <mapping> element for each mapped class. The Hibernate properties used in hibernate.cfg.xml file are listed below:

* **hibernate.dialect** - specifies the type of database used in hibernate so that hibernate generates the appropriate type of SQL statements. The [org.hibernate.dialect](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/dialect/package-summary.html" \t "_blank) package contains the SQL dialect for the databases.
* **hibernate.connection.driver\_class** - used to register or load the JDBC driver class. The name should be fully qualified class name like oracle.jdbc.driver.OracleDriver for oracle , com.mysql.jdbc.Driver for MySQL, etc.
* **hibernate.connection.url** - used to mention the JDBC URL to the database instance.
* **hibernate.connection.username** - username used to connect to the database.
* **hibernate.connection.password** - password for the username being used to connect to the database.
* **hibernate.connection.pool.size** - used to limit the number of connections waiting in the Hibernate database connection pool.
* **show\_sql** - If this property value is true then it enables the logging of all hibernate-generated SQL statements to the console.
* **format\_sql** - If this property value is true then it formats the generated SQL statement to make it more readable.
* **use\_sql\_comments** - If this property value is true then it insert comments inside all generated SQL statements.
* **hibernate.hbm2ddl.auto** - used to create, update or validate the database schema DDL when the SessionFactory Object created. The four possible values for this property are,
  + **create** - creates new database tables based on your class mappings. If a table already exists, then it will drop the existing table and create a new table.
  + **update** – Updates the existing database tables to match the class mappings. If a table doesn't exist, then it creates a new table.
  + **validate** - validates the existing tables against the provided mappings, and doesn't make any changes to the database. If the validation fails, the application will not work properly.
  + **create-drop** - as create, but explicitly drops all existing tables when the SessionFactory is closed (which of course loses all the data in those tables).

## hibernate.properties file

We can use a hibernate.properties file to configure Hibernate instead.

Hibernate searches for the hibernate.cfg.xml file or the hibernate.properties file at startup to find the configuration in our classpath. If both are present, then hibernate gives priority to a hibernate.cfg.xml file over a hibernate.properties file.

A hibernate.properties file equivalent to the above hibernate.cfg.xml file is given below:

hibernate.dialect=org.hibernate.dialect.MySQLDialect

hibernate.connection.driver\_class=com.mysql.jdbc.Driver

hibernate.connection.url=jdbc:mysql://localhost:3306/test\_db?useSSL=false

hibernate.connection.username=root

hibernate.connection.password=root

show\_sql=true

hbm2ddl.auto=create

The difference between a hibernate.cfg.xml and hibernate.properties file is that in an XML file we can directly map classes using the <mapping> element, but there is no way to configure this in a properties file. So, we map the classes using programmatic configuration.

We can specify the mapped class using Configuration object:

Configuration configuration = new Configuration();

configuration.configure().addAnnotatedClass(Student.class);

**XML vs Annotation configuration in Hibernate**

Hibernate classically uses an XML mapping file for the transformation of data from POJO Classes to database tables, and vice versa. An XML file gives the ability to change the configuration without rebuilding the whole project.

In especially large projects though, this can leave you with hundreds of XML files floating around. **Hibernate annotations** provide another way to define mappings directly in your POJO classes, without the use of an XML file.

**Hibernate Annotations**

Hibernate Annotations are used to provide metadata configuration inside the POJO class, so we can understand the table structure and POJO class simultaneously.

Hibernate Annotations are based on the JPA 2 specification. All the JPA annotations are defined in the **javax.persistence** package. Some of the annotation used for table and column mappings are listed below:

Consider the following **Students table** to store the Persistent objects.

create table students (

id INT NOT NULL auto\_increment,

first\_name VARCHAR(20) default NULL,

last\_name VARCHAR(20) default NULL,

email VARCHAR(30) default NULL,

PRIMARY KEY (id)

);

Let us create a **Student** persistent class that is mapped to a student table.

package com.revature.hibernate;

import javax.persistence.\*;

@Entity

@Table(name = "students")

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column

private int id;

@Column(name = "first\_name")

private String firstName;

@Column(name = "last\_name")

private String lastName;

@Column

private String email;

public student(){

}

public Student(String firstName, String lastName, String email) {

this.firstName = firstName;

this.lastName = lastName;

this.email = email;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

@Override

public String toString() {

return "Student [id=" + id + ", firstName=" + firstName + ", lastName=" + lastName + ", email=" + email + "]";

}

}

Hibernate annotations used in the above class are listed below:

* **@Entity** - Used to mark a class as a Mapped class/Persistence class. This class **must** have a no-arg constructor with package visibility so that hibernate can create an instance of the Persistent class by the newInstance() method.
* **@Table** - Used to specify the table details that used to persist the entity in the database. If the name of the database table differs from the name of the class, the name attribute should be used.
* **@Id** - Used to mark the field as a primary key column. Annotating multiple fields with @Id will make them composite keys
* **@GeneratedValue** - Used to instruct the database to generate a value for the field automatically, and to provide a strategy for doing so.
* **@Column** - Maps the field to the table column. The @Column annotation has attributes listed below:
  + **name** - used to specify the name of the column. By default, it's assumed the column name and variable name match. This attribute is required if that is not the case.
  + **length** - used to specify the size of the String value
  + **nullable** - used to mark the column as NOT NULL when the schema is generated.
  + **unique** - used to mark the column as UNIQUE to contain unique values.

Mapping in Hibernate

Inheritance Mapping

1. Table Per Hierarchy
2. Table Per Concrete Class
3. Table Per Sub-Class

|  |  |  |  |
| --- | --- | --- | --- |
|  | Table Per Hierarchy | Table Per Concrete Class | Table Per Sub Class |
| 1 | @Inheritance (strategy=InheritanceType.SINGLE\_TABLE), | @Inheritance( strategy = InheritanceType.TABLE\_PER\_CLASS) | **@Inheritance(strategy=InheritanceType.JOINED)** |
| Base Class | @DiscriminatorColumn(name="type", discriminatorType=DiscriminatorType.STRING) | @AttributeOverrides({  @AttributeOverride(name="id",  column=@Column(name="id")),   @AttributeOverride(name="name",  column=@Column(name="name"))  }) | @PrimaryKeyJoinColumn(name="ID") |
| Sub Class | @DiscriminatorValue(value="employee") |  |  |

Database Relationship

1. One to One ( Primary Key – No Duplicates & No Null) [1..1]
2. One to Many [1..N]
3. Many to One [N..1]
4. Many to Many [M..N]

Collection Mapping

1. List <T> (String)
2. Set <T>
3. Bag <T>
4. Map <K,V>

https://ide.interviewmocha.com/