N.HARI PRASAD(LAB-9)

Q1. Create an application in Maven project using hibernate, with CRUD operations

Employee pom.xml:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>Hari12345</groupId>
<artifactId>hari12345</artifactId>
<version>0.0.1-SNAPSHOT</version>
<dependencies>

<!-- Hibernate 4.3.6 Final -->
<dependency>
<groupId>org.hibernate</groupId>
<artifactId>hibernate-core</artifactId>
<version>4.3.6.Final</version>
</dependency>
<!-- Mysql Connector -->
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<version>8.0.12</version>
</dependency>
<dependency>
<groupId>hari1111</groupId>
<artifactId>hari1111</artifactId>
<version>0.0.1-SNAPSHOT</version>
</dependency>
<dependency>
<groupId>RelationshipDemo</groupId>
<artifactId>RelationshipDemo</artifactId>
<version>0.0.1-SNAPSHOT</version>
</dependency>
</dependencies>
</project>
```

Employee Entity:

```
package Employee.Manage;
import javax.persistence.*;
@Entity
@Table(name = "EMPLOYEE")
public class Employee {
@Id @GeneratedValue
@Column(name = "id")
private int id;
@Column(name = "first_name")
private String firstName;
@Column(name = "last_name")
private String lastName;
@Column(name = "salary")
private int salary;
public Employee() {}
public int getId() {
return id;
}
public void setId( int id ) {
this.id = id;
}
public String getFirstName() {
return firstName;
}
public void setFirstName( String first_name ) {
this.firstName = first_name;
}
public String getLastName() {
return lastName;
}
public void setLastName( String last_name ) {
this.lastName = last_name;
}
public int getSalary() {
return salary;
}
public void setSalary( int salary ) {
this.salary = salary;
}
}
```

Employee hibernate configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-
configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<property name="hibernate.dialect">
org.hibernate.dialect.MySQLDialect
</property>
<property name="hibernate.connection.driver_class">
com.mysql.jdbc.Driver
</property>
<!-- Assume test123 is the database name -->
<property name="hibernate.connection.url">
jdbc:mysql://localhost/test123
</property>
<property name="hibernate.connection.username">
root
</property>
<property name="hibernate.connection.password">
root
</property>
</session-factory>
</hibernate-configuration>
```

Employee Manage Program:

```
package Employee.Manage;

import java.util.List;

import java.util.Date;

import java.util.Iterator;


import org.hibernate.HibernateException;

import org.hibernate.Session;

import org.hibernate.Transaction;

import org.hibernate.cfg.AnnotationConfiguration;
```

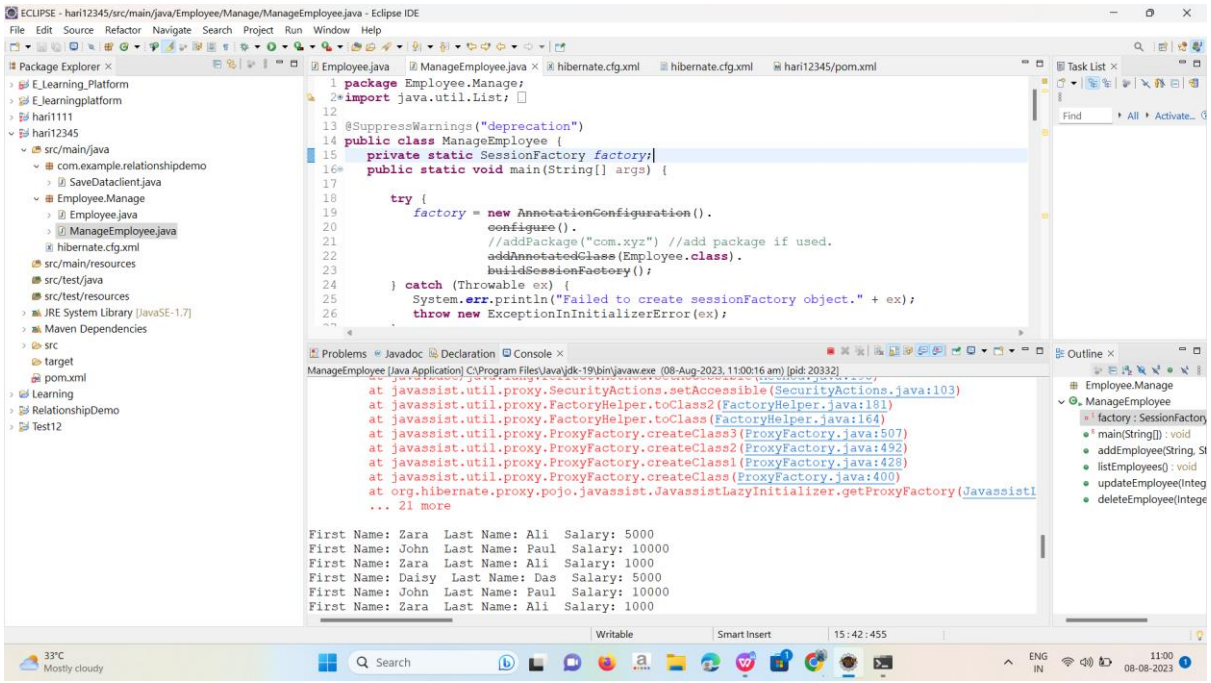
```
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

@SuppressWarnings("deprecation")
public class ManageEmployee {
  private static SessionFactory factory;
  public static void main(String[] args) {

    try {
      factory = new AnnotationConfiguration().
            configure().
            //addPackage("com.xyz") //add package if used.
            addAnnotatedClass(Employee.class).
            buildSessionFactory();
    } catch (Throwable ex) {
      System.err.println("Failed to create sessionFactory object." + ex);
      throw new ExceptionInInitializerError(ex);
    }

    ManageEmployee ME = new ManageEmployee();

    /* Add few employee records in database */
    Integer empID1 = ME.addEmployee("HARI", "PRASAD", 1000);
    Integer empID2 = ME.addEmployee("SRINU", "NEELISETTI", 5000);
    Integer empID3 = ME.addEmployee("SWEETY", "SWATHI", 10000);

    /* List down all the employees */
```

```
  ME.listEmployees();

   /* Update employee's records */
  // ME.updateEmployee(empID1, 5000);

   /* Delete an employee from the database */
   //ME.deleteEmployee(empID2);

   /* List down new list of the employees */
   ME.listEmployees();
}

/* Method to CREATE an employee in the database */
public Integer addEmployee(String fname, String lname, int salary){
   Session session = factory.openSession();
   Transaction tx = null;
   Integer employeeID = null;

   try {
      tx = session.beginTransaction();
      Employee employee = new Employee();
      employee.setFirstName(fname);
      employee.setLastName(lname);
      employee.setSalary(salary);
      employeeID = (Integer) session.save(employee);
      tx.commit();
   } catch (HibernateException e) {
```

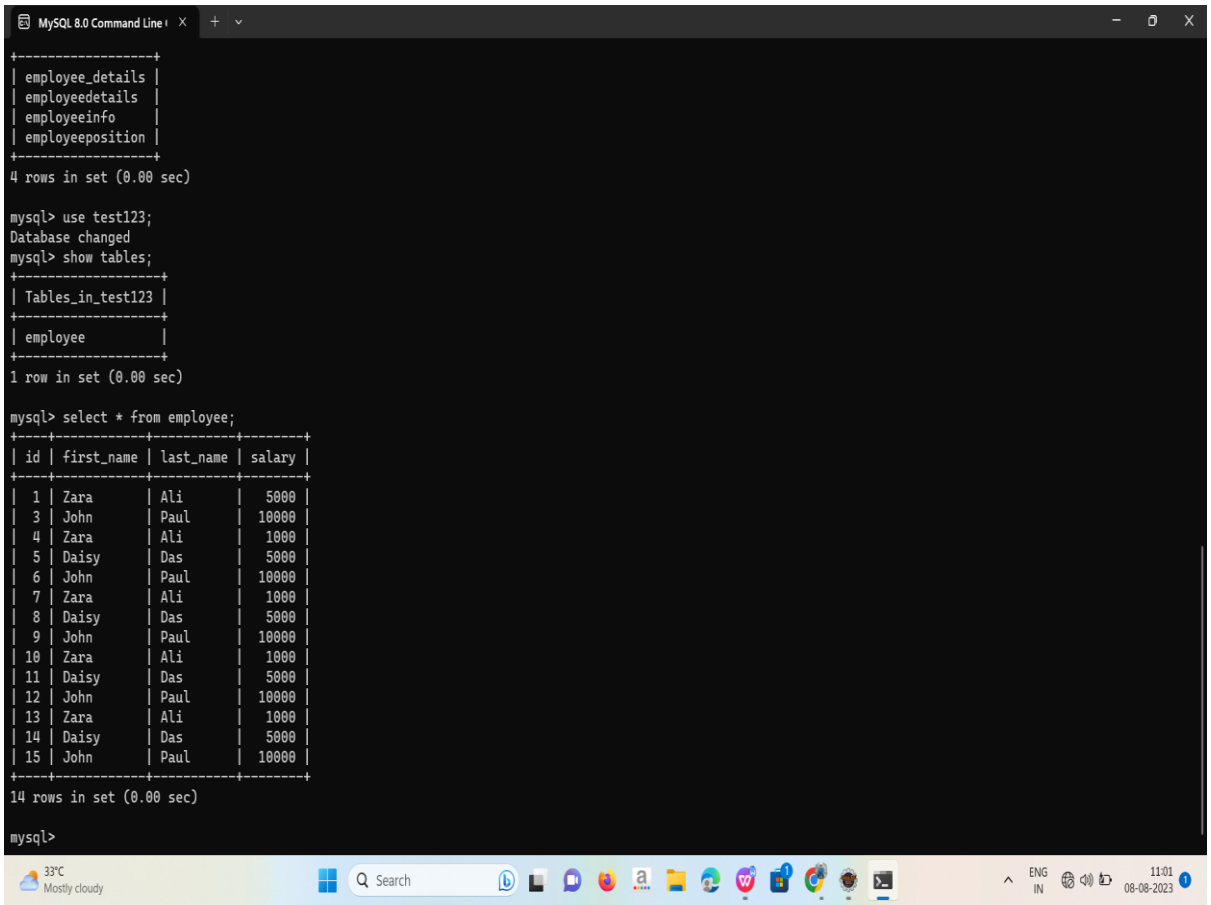
```
      if (tx!=null) tx.rollback();
      e.printStackTrace();
    } finally {
      session.close();
    }
    return employeeID;
  }

  /* Method to  READ all the employees */
  public void listEmployees( ){
    Session session = factory.openSession();
    Transaction tx = null;

    try {
      tx = session.beginTransaction();
      List employees = session.createQuery("FROM Employee").list();
      for (Iterator iterator = employees.iterator(); iterator.hasNext();){
        Employee employee = (Employee) iterator.next();
        System.out.print("First Name: " + employee.getFirstName());
        System.out.print("  Last Name: " + employee.getLastName());
        System.out.println("  Salary: " + employee.getSalary());
      }
      tx.commit();
    } catch (HibernateException e) {
      if (tx!=null) tx.rollback();
      e.printStackTrace();
    } finally {
```

```
      session.close();
    }
  }

  /* Method to UPDATE salary for an employee */
  public void updateEmployee(Integer EmployeeID, int salary ){
    Session session = factory.openSession();
    Transaction tx = null;

    try {
      tx = session.beginTransaction();
      Employee employee = (Employee)session.get(Employee.class,
EmployeeID);
      employee.setSalary( salary );
                session.update(employee);
      tx.commit();
    } catch (HibernateException e) {
      if (tx!=null) tx.rollback();
      e.printStackTrace();
    } finally {
      session.close();
    }
  }

  /* Method to DELETE an employee from the records */
  public void deleteEmployee(Integer EmployeeID){
    Session session = factory.openSession();
```

```
    Transaction tx = null;


    try {
      tx = session.beginTransaction();
      Employee employee = (Employee)session.get(Employee.class,
EmployeeID);
      session.delete(employee);
      tx.commit();
    } catch (HibernateException e) {
      if (tx!=null) tx.rollback();
      e.printStackTrace();
    } finally {
      session.close();
    }
  }
}
```



Employee Table in MYSQL:



Q2. write and explain hibernate.cfg and hibernate.hbm file usage in ORM?

In Object-Relational Mapping (ORM), Hibernate is a popular Java framework used to map Java objects to relational database tables and manage the interaction between them. Hibernate uses two configuration files, hibernate.cfg.xml and .hbm.xml files, to define the configuration and mapping details, respectively.

hibernate.cfg.xml:

This file is the main configuration file for Hibernate, and it contains various settings and properties needed to set up and configure the Hibernate environment. Some of the common properties found in this file are:

Database connection settings: It includes properties like database URL, username, password, and the JDBC driver class to connect to the database.

Dialect: Specifies the SQL dialect for the database being used. It helps Hibernate generate appropriate SQL queries for the specific database.

Caching settings: Defines cache-related properties to optimize performance.

Mapping file reference: This specifies the location of the .hbm.xml files that contain the object-to-table mapping definitions.

Connection pooling settings: Configures connection pooling to efficiently manage database connections.

Schema generation settings: Defines how Hibernate will handle database schema creation and updates.

Here's a simple example of a hibernate.cfg.xml file:

```
<hibernate-configuration>

<session-factory>

<!-- Database connection settings -->

<property
name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</proper
ty>

<property
name="hibernate.connection.url">jdbc:mysql://localhost:3306/mydb</pr
operty>

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

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

<!-- Dialect -->

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

<!-- Caching settings -->

<property
name="hibernate.cache.use_second_level_cache">true</property>

<property
name="hibernate.cache.region.factory_class">org.hibernate.cache.ehcache.
SingletonEhCacheRegionFactory</property>

<!-- Mapping file reference -->

<mapping resource="com/example/MyEntity.hbm.xml"/>

<!-- Connection pooling settings -->

<property name="hibernate.connection.pool_size">10</property>

<property
name="hibernate.connection.provider_class">org.hibernate.connection.C3
P0ConnectionProvider</property>
```

```
<!-- Schema generation settings -->
<property name="hibernate.hbm2ddl.auto">update</property>
</session-factory>
</hibernate-configuration>
```

".hbm.xml" files:

These files are used for defining the mapping between Java classes (entities) and database tables. Each .hbm.xml file corresponds to a single Java entity and contains XML mappings that describe how the properties of the Java class should be persisted into the database columns.

Here's an example of a .hbm.xml file:

```
<hibernate-mapping>
<class name="com.example.MyEntity" table="my_entity">
<id name="id" type="long">
<column name="id" />
<generator class="native" />
</id>
<property name="name" type="string">
<column name="name" />
</property>
<property name="age" type="integer">
<column name="age" />
</property>
</class>
</hibernate-mapping>
```

In this example, the .hbm.xml file maps the com.example.MyEntity Java class to a table named my_entity. The class has two properties: name and age, which are mapped to the respective columns in the database table.

To summarize, hibernate.cfg.xml is the main configuration file that sets up Hibernate's environment, and .hbm.xml files are used to define the mapping between Java classes and database tables, allowing Hibernate to perform the Object-Relational Mapping. Together, these files enable developers to interact with the database using Java objects, abstracting away the underlying SQL and database complexities.

Q3. Explain advantages of HQL and Cashing in hibernate?

Hibernate Query Language (HQL) and caching are two important features of Hibernate, a popular Object-Relational Mapping (ORM) framework for Java. Let's explore the advantages of both:

Advantages of Hibernate Query Language (HQL):

- Database Independence: HQL is a powerful query language that abstracts the underlying SQL database, making your code independent of the specific database vendor. You can write the queries in HQL, and Hibernate will translate them to the appropriate SQL statements for the target database.

- Object-Oriented Querying: HQL allows you to write queries using Java classes and properties instead of database tables and columns. This makes the queries more expressive and easier to understand, especially for developers familiar with object-oriented programming.

- Rich Query Capabilities: HQL supports various query features, such as filtering, sorting, joining, grouping, and aggregation, similar to SQL. It also supports subqueries and other advanced SQL features, providing powerful querying capabilities for complex use cases.

- Reusability and Maintainability: With HQL, you can define named queries at the entity level, allowing you to reuse them across different parts of your application. This improves code maintainability and reduces duplication.

- Type Safety: HQL is type-safe, which means the compiler can catch type-related errors at compile-time rather than runtime. This enhances code reliability and reduces the likelihood of SQL-related bugs.

- Integration with Criteria Queries: Hibernate provides Criteria API, which allows you to build type-safe queries programmatically using a more fluent and object-oriented approach. HQL and Criteria queries complement each other and offer different query styles to cater to different developer preferences.

Advantages of Caching in Hibernate:

- Improved Performance: Caching in Hibernate reduces the number of database trips, as data can be retrieved from the cache instead of going to the database. This results in faster response times and improved application performance, especially for frequently accessed data.

- Reduced Database Load: By caching frequently accessed data in memory, the load on the database server decreases. This can help improve the overall scalability of the application and reduce the risk of database bottlenecks.

- Network Latency Reduction: Caching minimizes the need for data transfer between the application and the database, which can be a significant source of latency. Cached data is readily available in memory, eliminating the need for network round-trips.

- Consistency and Predictability: Hibernate caching provides mechanisms to ensure that cached data is kept consistent with the underlying database. This maintains data integrity and prevents data staleness issues that can occur with aggressive caching strategies.

- Customizable Cache Strategies: Hibernate allows you to configure different cache strategies based on your application's requirements. You can choose from options like read-only, read-write, transactional, and more, to optimize cache behavior for specific entities or query results.

- Second-Level Cache: Hibernate supports both first-level and second-level caching. The first-level cache operates within the current session, while the second-level cache is shared across multiple sessions. The second-level cache helps in caching data that is accessed across different sessions, providing broader caching benefits.

In summary, HQL offers a powerful and flexible way to query the database using object-oriented concepts, while caching improves application performance by reducing database access and network latency. Properly utilizing HQL and caching in Hibernate can lead to more efficient and maintainable applications with enhanced performance.

Q4. Describe SessoinFactory, Session, Transaction objects?

In Hibernate, the SessionFactory, Session, and Transaction are three fundamental objects that play crucial roles in managing the interaction between the Java application and the database.

SessionFactory:

The SessionFactory is a heavyweight and thread-safe object in Hibernate. It is responsible for creating and managing Session objects. The SessionFactory is typically instantiated once when the application starts up and should be reused throughout the application's lifecycle. It represents the connection between your Java application and the database.

Advantages of using SessionFactory:

It caches the mapping metadata and the compiled query plans, improving performance and reducing the overhead of repeated parsing and optimization.

It manages connection pooling, ensuring that database connections are efficiently reused, resulting in better resource utilization and reduced connection overhead.

It is expensive to create, so using a single SessionFactory instance for the entire application is a recommended practice.

Session:

A Session is a lightweight object representing a single unit of work with the database. It is obtained from the SessionFactory and acts as a bridge between your Java application and the database. You use the Session to perform CRUD (Create, Read, Update, Delete) operations on entities mapped to database tables.

Advantages of using Session:

It provides an abstraction over the underlying JDBC connections, shielding the application from the complexities of dealing with raw SQL.

It implements the first-level cache (also known as the session cache or the Persistence Context). This cache holds the objects retrieved or persisted during a session, which helps reduce the number of database calls and improve performance.

It supports transactions, allowing you to group multiple database operations into a single transactional unit with commit and rollback capabilities.

Transaction:

The Transaction object represents a unit of work that is typically composed of one or more database operations. It is used to manage database transactions within the Session. Transactions ensure data consistency and integrity by providing a way to commit changes or roll back in case of errors or failures.

Advantages of using Transaction:

It allows you to group multiple database operations into a single logical transaction. If any operation within the transaction fails, you can roll back the entire transaction, ensuring data consistency.

It ensures that database changes made within a session are only persisted when the transaction is committed, preventing partial or inconsistent updates.

It provides isolation levels to control the visibility of data changes made by other concurrent transactions, offering options like read committed, repeatable read, and serializable.

ConnectionProvider:

It is a factory of JDBC connections. It abstracts the application from DriverManager or DataSource. It is optional.

TransactionFactory:

It is a factory of Transaction. It is optional.

In summary, the SessionFactory, Session, and Transaction are essential components in Hibernate that facilitate efficient communication with the database, manage caching, and ensure data integrity through transactional

behavior. Properly utilizing these objects is key to building robust and performant Hibernate applications.