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

Employee pom.xml:

```
<projectxmlns="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>EMpDB</groupId>
<artifactId>Emp_DB_Maven</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>8.0.18</version>
</dependency>
</dependencies>
</project>
```

Employee Entity:

```
packageEmployee.Manage;
importjavax.persistence.*;
@Entity
@Table(name = "EMPLOYEE")
publicclass Employee {
@Id@GeneratedValue
@Column(name = "id")
privateintid;
@Column(name = "first_name")
private String firstName;
@Column(name = "last_name")
private String lastName;
@Column(name = "salary")
privateintsalary;
publicEmployee() {}
publicintgetId() {
returnid;
}
publicvoidsetId( intid ) {
this.id = id;
}
public String getFirstName() {
returnfirstName;
}
publicvoidsetFirstName( Stringfirst_name ) {
this.firstName = first_name;
}
public String getLastName() {
returnlastName;
}
publicvoidsetLastName( Stringlast_name ) {
this.lastName = last_name;
}
publicintgetSalary() {
returnsalary;
}
publicvoidsetSalary( intsalary ) {
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("Manisha", "Rani", 10000);
Integer empID2 = ME.addEmployee("Anvesh", "Sachin", 15000);
Integer empID3 = ME.addEmployee("pooja", "Sharma", 20000);

/* List down all the employees */
ME.listEmployees();

/* Update employee's records */
ME.updateEmployee(empID1,
25000);

/* 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();
}
}
}
```

Q2. write and explain hibernate.cfg and hibernate.hbmfile 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</p
roperty>

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

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

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

<!-- Dialect -->

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

<!-- Caching settings -->

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

<property
name="hibernate.cache.region.factory_class">org.hibernate.cache.ehc
ache.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.C3P0ConnectionProvider</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):

Advantages of Hibernate Query Language (HQL):

Hibernate Query Language (HQL) is a powerful and expressive query language provided by Hibernate that allows developers to perform database operations using object-oriented syntax. HQL offers several advantages:

1.Object-Oriented Querying: HQL uses object-oriented syntax that closely resembles SQL but operates on persistent objects and their properties.

2. Database Portability: HQL abstracts away the underlying database-specific SQL syntax. This means that you can write HQL queries without worrying about database-specific differences, making your application more portable across different database systems.

3. Reuse of Business Logic: With HQL, you can define named queries in your mapping files or annotations.

4. Entity-Based Queries: HQL works directly with entity classes, allowing you to express complex relationships and joins between entities in a natural and readable manner.

5. Eager and Lazy Loading Control: HQL queries can control the fetching strategy of related objects, allowing you to load related objects eagerly or lazily based on your application's requirements.

6. Dynamic Query Building: HQL supports dynamic query building, where you can construct queries at runtime based on user input or application logic.

Advantages of Caching in Hibernate:

Caching in Hibernate involves storing frequently accessed data in memory to reduce the number of database queries, thereby improving application performance. Here are the advantages of caching in Hibernate:

1.Improved Performance: Caching reduces the number of database round-trips by storing frequently accessed data in memory.

2. Reduced Database Load: With caching, the application can retrieve data directly from the cache instead of querying the database.

3. Network Latency Reduction: Database queries involve network communication, which can introduce latency.

4. Enhanced Scalability: By reducing the load on the database server, caching enhances the scalability of your application

5. Consistent Data: Hibernate's caching mechanisms help maintain data consistency.

6. Query Result Caching: Hibernate supports caching the results of queries using the second-level cache.

7. Read-Only Data Optimization: Caching read-only data that doesn't change frequently can result in substantial performance improvements.

8. Local and Distributed Caching: Hibernate supports different caching strategies, including local caching (within a JVM) and distributed caching (across multiple JVMs), allowing you to choose the approach that suits your application architecture.

In summary, HQL simplifies querying and interaction with your database, while caching enhances performance by reducing database interactions .

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.

Let’s know each in detail

In Hibernate, the SessionFactory, Session, and Transaction objects are fundamental components that play crucial roles in managing the interactions between your Java application and the database. Let's delve into the details of each:

1. SessionFactory:

The SessionFactory is a heavyweight object in Hibernate that is responsible for creating and managing Session instances. It is typically instantiated once and shared throughout the application's lifecycle. The SessionFactory is thread-safe and designed to be a long-lived object.

characteristics of the Session Factory:

- Database Connection Management
- Configuration and Metadata
- Session Creation:
- The Session Factory
- Caching Management
- Immutable

2. Session:

A Session in Hibernate represents a single unit of work and serves as an interface to interact with the database. Each interaction with the database should occur within the context of a Session. A Session is lightweight and not thread-safe, meaning that each thread should have its own Session instance.

Characteristics of Session are :

- Database Operations
- Identity Map
- Caching
- Transaction Management
- Lazy Loading
- Detached Objects

3. Transaction:

A Transaction in Hibernate represents a single unit of work within a database session. It ensures that a series of database operations either complete successfully or are rolled back in case of an error or exception.

Responsibilities of the Transaction are:

- ACID Properties
- Data Integrity
- Transactional Boundaries
- Isolation Levels

In summary, the Session Factory is responsible for creating and managing sessions, the Session provides the primary interface for database interactions within a unit of work, and the Transaction ensures the integrity and consistency of these interactions. Understanding the roles and responsibilities of these objects is essential for effectively utilizing Hibernate's ORM capabilities.