1. 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>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:

```
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("Anusha", "Ramdasu", 1000);
Integer empID2 = ME.addEmployee("Deepthi", "Ettamsethi", 5000);
Integer empID3 = ME.addEmployee("Sujii", "Potnuri", 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();
}
}
}
```

Output:

Employee Table:

2. Write explain hibernate.cfg and hibernate.hbm file usage in ORM.

In Hibernate, both the `hibernate.cfg.xml` and `hibernate.hbm.xml` files play essential roles in configuring and defining the object-relational mapping (ORM) for the application.

1. `hibernate.cfg.xml` file:

The `hibernate.cfg.xml` file is the configuration file used to set up various properties and settings for the Hibernate framework. It provides information such as database connection details, dialect, cache configuration, transaction management, and other global settings required for the proper functioning of Hibernate. This file is typically placed in the classpath of the application.

Below are some common configuration properties found in the `hibernate.cfg.xml` file:

- `**hibernate.dialect**`: Specifies the SQL dialect of the database being used, which helps Hibernate generate appropriate SQL queries for different database systems.

- `**hibernate.connection.url**`: The URL to connect to the database.

- `**hibernate.connection.username**` and `**hibernate.connection.password**`: Database login credentials.

- `**hibernate.connection.driver_class**`: The JDBC driver class to be used.

- `**hibernate.show_sql**`: If set to true, it enables the printing of generated SQL queries in the console, helpful for debugging purposes.

- `**hibernate.hbm2ddl.auto**`: Specifies how Hibernate should handle the schema generation (create, update, validate, or none).

- **`hibernate.cache.provider_class`**: Specifies the caching provider to be used for caching entities and queries.

2. `hibernate.hbm.xml` file:

The `hibernate.hbm.xml` file is used to define the mapping between Java classes (entities) and database tables. It contains Hibernate mapping configurations for each entity class, specifying how the properties of the class map to the columns in the corresponding database table.

In this file, you define the following information for each entity:

- **Class-to-table mapping**: Specify the entity class and the corresponding database table.

- **Property-to-column mapping**: Define how each property of the class maps to the columns in the table.

- **Primary key mapping**: Specify the primary key property and its generation strategy (e.g., identity, sequence, table).

- Relationships: Define associations between entities using various mapping annotations or XML elements (e.g., one-to-one, one-to-many, many-to-one, many-to-many).

Here's a simplified example of a `hibernate.hbm.xml` file for the `Employee` entity:

xml

```
<!DOCTYPE hibernate-mapping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD//EN"
      "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
   <class name="com.example.Employee" table="employee">
      <id name="id" type="long">
         <column name="id" />
         <generator class="identity" />
      </id>
      <property name="name" column="name" />
      <property name="department" column="department" />
   </class>
</hibernate-mapping>
```

In this example, we define the mapping for the `Employee` entity class to the "employee" table. The `id` property is the primary key with an identity generation strategy, and the `name` and `department` properties are mapped to the respective columns in the table.

Both the `hibernate.cfg.xml` and `hibernate.hbm.xml` files work together to provide the necessary configuration and mapping information to Hibernate, allowing it to manage the persistence of Java objects to the underlying relational database.

3. Explain advantages of HQL and CACHING in hibernate.

Advantages of HQL (Hibernate Query Language):

1. Object-Oriented Querying: HQL allows developers to write database queries using object-oriented concepts instead of raw SQL. This makes the queries more natural and easier to read and maintain as they closely resemble the entity classes in the application.

2. Database Independence: HQL provides database independence, meaning you can write queries without worrying about the underlying database's specific SQL dialect. Hibernate takes care of translating the HQL queries to the appropriate SQL statements for the target database.

3. Portability: Since HQL queries are database-independent, applications using Hibernate can be easily ported to different databases without significant changes to the queries, reducing the development effort and time.

4. Type Safety: HQL is type-safe, meaning it uses entity class properties and associations directly, avoiding the risk of runtime errors due to incorrect SQL syntax or invalid column names.

5. Eager and Lazy Fetching Control: HQL allows developers to control the fetching behavior of associated entities, specifying whether they should be fetched eagerly or lazily. This helps to optimize the database access and avoid unnecessary data loading.

Advantages of Caching in Hibernate:

1. Improved Performance: Caching significantly improves application performance by reducing the number of database queries. When entities and query results are cached, subsequent requests for the same data can be served from memory instead of hitting the database, which is much faster.

2. Reduced Database Load: Caching reduces the load on the database server, especially in read-heavy applications. It minimizes the number of database round-trips required for fetching data, thus increasing the overall application's scalability.

3. Response Time Optimization: Cached data is readily available in memory, leading to faster response times for frequently accessed data. This results in a more responsive application, providing a better user experience.

4. Consistency and Isolation: Hibernate cache provides transactional cache management, ensuring that data consistency is maintained. When changes are made to cached entities within a transaction, they are not visible to other transactions until the changes are committed.

5. Query Result Caching: Hibernate allows caching the results of complex queries using query caching. This helps to speed up the execution of queries that are executed frequently, especially when the results don't change often.

6. Second-Level Cache: Hibernate supports a second-level cache that spans multiple sessions, making it more efficient for caching data across different users or sessions in a multi-user application.

While using caching in Hibernate offers numerous advantages in terms of performance and scalability, it's essential to use caching judiciously and consider cache eviction strategies to prevent stale data or excessive memory usage. Caching should be used in situations where the benefits outweigh the risks, and data consistency remains intact.

4. Describe session factory, session, transaction objects.

Session Factory:

The Session Factory is a core component of Hibernate, an object-relational mapping (ORM) framework in Java. It acts as a factory for creating Session objects, which are used to interact with the database. The Session Factory is typically created once during the application's startup and is thread-safe, meaning multiple threads can access it simultaneously without issues.

The Session Factory is responsible for configuring and managing various aspects of Hibernate, such as database connection properties, mapping metadata, caching, and transaction management. It builds upon the configuration provided by the application, mapping files, and annotations to create and manage a pool of database connections.

Session:

The Session represents a single unit of work with the database in Hibernate. It acts as an interface between the application and the database, allowing the application to perform CRUD (Create, Read,

Update, Delete) operations on the mapped objects (entities) without dealing directly with SQL queries.

Each thread in the application typically has its own Session object. A session is created from the Session Factory using the openSession() method and should be closed when the work is completed to release the associated resources.

The Session caches the loaded entities and manages their state throughout its lifecycle. It provides methods to save, update, delete, and query the database using the Hibernate Query Language (HQL) or Criteria API.

Transaction:

The Transaction object in Hibernate represents a single unit of work that is performed within a Session. It is used to define a boundary around a set of database operations, making sure they are executed atomically. In other words, either all the operations within a transaction are committed to the database, or none of them are.

The Transaction is created by the Session using the beginTransaction() method. It can be managed manually or automatically (using declarative transaction management). A typical transaction flow involves starting the transaction, performing database operations, and then either committing the transaction to make the changes permanent or rolling it back to discard the changes in case of failure.

By using transactions, Hibernate ensures data integrity and consistency while maintaining the ACID (Atomicity, Consistency, Isolation, Durability) properties of the database. It also helps to avoid potential data corruption or incomplete data updates in case of application failures or exceptions.

In summary, the Session Factory, Session, and Transaction objects are essential components of Hibernate, providing the means to interact with the database, manage entities, and ensure data integrity through transactional operations.