S.N.R.Likhitha(AF0312909)

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

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>Sat</groupId>
  <artifactId>Manny</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>
```

Hibernate configuration:

```
<?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 Emp is the database name -->
<property name = "hibernate.connection.url">
jdbc:mysql://localhost/Emp
</property>
<property name = "hibernate.connection.username">
root
</property>
<property name = "hibernate.connection.password">
root
</property>
```

```
</session-factory>
</hibernate-configuration>
```

Create Entity class:

```
package Likhitha;

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;
}
}
```

Program:

package Likhitha;

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("Zara", "Ali", 1000);
    Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);
    Integer empID3 = ME.addEmployee("John", "Paul", 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:

```
First Name: Zara  Last Name: Ali  Salary: 5000
First Name: Daisy  Last Name: Das  Salary: 5000
First Name: John  Last Name: Paul  Salary: 10000
First Name: Zara  Last Name: Ali  Salary: 5000
First Name: Daisy  Last Name: Das  Salary: 5000
First Name: John  Last Name: Paul  Salary: 10000
First Name: Zara  Last Name: Ali  Salary: 1000
First Name: Daisy  Last Name: Das  Salary: 5000
First Name: John  Last Name: Paul  Salary: 10000
```

```
mysql> select *from employee;
+----+------------+-----------+--------+
| id | first_name | last_name | salary |
+----+------------+-----------+--------+
|  4 | Zara       | Ali       |   5000 |
|  5 | Daisy      | Das       |   5000 |
|  6 | John       | Paul      |  10000 |
|  7 | Zara       | Ali       |   5000 |
|  8 | Daisy      | Das       |   5000 |
|  9 | John       | Paul      |  10000 |
| 10 | Zara       | Ali       |   5000 |
| 11 | Daisy      | Das       |   5000 |
| 12 | John       | Paul      |  10000 |
+----+------------+-----------+--------+
9 rows in set (0.00 sec)
```

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

1.Hibernate Configuration:

As Hibernate can operate in different environments, it requires a wide range of configuration parameters. These configurations contain the mapping information that provides different functionalities to Java classes. Generally, we provide database related mappings in the configuration file. Hibernate facilitates to provide the configurations either in an XML file (like hibernate.cfg.xml) or properties file (like hibernate.properties).

In Hibernate, the **hibernate.cfg.xml** file is a crucial configuration file that is used to set up the Hibernate environment and manage various settings related to the Object-Relational Mapping (ORM) process. It contains all the necessary configurations required for Hibernate to interact with the database and handle entity mappings. Let's explore the usage and explanation of the **hibernate.cfg.xml** file:

1. Location and Name: The hibernate.cfg.xml file is typically placed in the classpath of your application (e.g., in the src or resources folder). The name of the file should be hibernate.cfg.xml to ensure that Hibernate can find and load it automatically.
2. XML Structure: The hibernate.cfg.xml file follows an XML structure and includes various elements to specify the configuration settings.
3. Session Factory: The most important element in the hibernate.cfg.xml file is the <session-factory> element. It acts as a factory for Hibernate Session objects, which are used to interact with the database. All the necessary configurations and properties are specified within this element.
4. Database Connection: You need to provide the database connection details to Hibernate, including the driver class, connection URL, username, and password. These properties allow Hibernate to establish a connection to the database. You use the following properties for this purpose:
    - hibernate.connection.driver_class: The fully qualified name of the database driver class.
    - hibernate.connection.url: The URL that specifies the database location and other connection parameters.
    - hibernate.connection.username: The username used to authenticate to the database.
    - hibernate.connection.password: The password for the database user.
5. Hibernate Dialect: The hibernate.dialect property is essential as it specifies the SQL dialect of the database being used. Different databases have slightly different SQL syntax, and Hibernate needs to know the appropriate dialect to generate compatible SQL queries.
6. Mapping Files: In the hibernate.cfg.xml file, you specify the mapping files (.hbm.xml files) for all the entity classes that Hibernate should manage. These mapping files contain information about how the Java classes are mapped to the database tables.

7. Other Configurations: Apart from the above, the hibernate.cfg.xml file allows you to specify various other configurations, such as caching settings, transaction management settings, logging configurations, etc. These configurations can help optimize the performance and behavior of your Hibernate application.

Once you have configured the hibernate.cfg.xml file with all the necessary properties and mapping information, you can use Hibernate's SessionFactory to obtain Session objects. These sessions are used to perform CRUD (Create, Read, Update, Delete) operations and other database interactions with your Java entities. The SessionFactory is created based on the configurations provided in the hibernate.cfg.xml file, making it the central configuration file for Hibernate ORM.

2. Hibernate.hbm:

These files define the mapping between Java classes (entities) and database tables. Each Java class that you want to persist in the database should have a corresponding *.hbm.xml file.

In Hibernate, the *.hbm.xml files are used to define the mappings between Java classes (entities) and database tables. These files play a crucial role in the Object-Relational Mapping (ORM) process by providing the necessary information to Hibernate about how Java objects should be persisted in the relational database and how to retrieve them back as objects.

Let's dive into the usage and explanation of the Hibernate *.hbm.xml files:

Defining Entity Mapping: The primary purpose of the *.hbm.xml file is to define the mapping between your Java entity classes and the corresponding database tables. Each Java entity class should have a corresponding *.hbm.xml file that describes how the properties of the entity map to the columns of the table.

XML Structure: The *.hbm.xml file has an XML structure and follows a specific DTD (Document Type Definition). The root element is <hibernate-mapping>, and under it, you define the mapping for each entity class using the <class> element.

Class Element: The <class> element is used to define the mapping for a specific Java entity class. It has attributes like name (fully qualified class name) and table (database table name). Within the <class> element, you define the mappings for the properties of the entity using <property> or <id> elements.

Property and Id Elements:

<property>: This element is used to map a simple property of the entity to a column in the database table. It has attributes like name (property name) and column (database column name).

<id>: This element is used to map the primary key (ID) of the entity. It has attributes like name (property name) and column (database column name). Additionally, you specify the type of the primary key and the <generator> to generate unique identifiers for new entities.

Relationship Mappings: Besides mapping properties, you can also define relationships between entities using elements like <one-to-one>, <one-to-many>, <many-to-one>, and <many-to-many>. These elements allow you to express associations and define the cardinality between entities.

Inheritance Mapping: Hibernate supports various inheritance strategies for entity inheritance. You can use <subclass>, <joined-subclass>, or <union-subclass> elements to define the inheritance mapping for your entities.

Collection Mappings: Hibernate also supports collections (e.g., Set, List, Map) and allows you to map them to database tables.

You can use elements like <set>, <list>, <map>, etc., to map these collections and their relationships with other entities.

3.Explain advantages of hql and caching in hibernate?

Hibernate Query Language (HQL)

Hibernate Query Language (HQL) is same as SQL (Structured Query Language) but it doesn't depends on the table of the database. Instead of table name, we use class name in HQL. So it is database independent query language.

Advantages of HQL (Hibernate Query Language):

1. Object-Oriented Querying: HQL is an object-oriented query language provided by Hibernate. It allows developers to write database queries using entity class and property names, making the queries more intuitive and closely resembling the Java code. This simplifies the querying process and reduces the need for writing complex native SQL queries.
2. Database Independence: One of the primary advantages of HQL is that it provides database independence. Developers can write HQL queries, and Hibernate takes care of translating these queries into appropriate SQL statements based on the underlying database dialect. This allows applications to be easily switched between different databases without changing the query syntax.
3. Readable and Maintainable Code: HQL queries are typically more readable and maintainable compared to native SQL queries. They abstract away the details of the

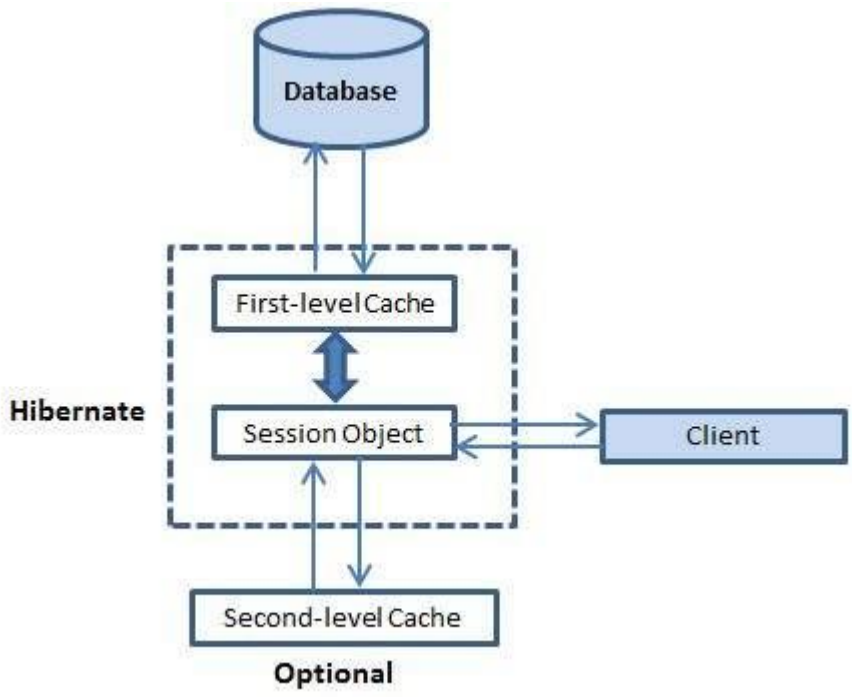
underlying database schema, making the code easier to understand, maintain, and modify.

4. Automatic Joins: HQL supports implicit joins, which means that developers don't need to explicitly specify joins between related entities. Hibernate automatically handles the necessary joins based on the entity associations defined in the entity mappings.
5. Dynamic Querying: HQL allows the creation of dynamic queries using parameters. This enables the development of flexible and reusable queries that can be customized at runtime based on different criteria or user inputs.
6. Support for Aggregations: HQL supports various aggregate functions, such as SUM, AVG, MAX, MIN, and COUNT. This simplifies the process of retrieving aggregate data from the database without the need to process the results in the application code.

Caching:

Caching is a mechanism to enhance the performance of a system. It is a buffer memorythat lies between the application and the database. Cache memory stores recently used data items in order to reduce the number of database hits as much as possible.

Caching is important to Hibernate as well. It utilizes a multilevel caching scheme as explained below −



Advantages of Caching in Hibernate:

1. Improved Performance: Caching in Hibernate helps reduce the number of database queries by storing frequently accessed data in memory. This results in faster response times and improved overall performance of the application.

2. Reduced Database Load: By caching frequently accessed data, Hibernate reduces the load on the underlying database server. This can be especially beneficial in applications with heavy read operations, as the database server is not overloaded with repetitive queries.
3. Minimized Network Traffic: Caching avoids unnecessary network traffic between the application server and the database, which is particularly advantageous when the application server and the database server are located on different machines or in different data centers.
4. Level of Control: Hibernate provides different levels of caching, such as first-level cache (session-level cache) and second-level cache (application-level cache). Developers can choose which entities or collections to cache and control the cache eviction strategies to suit their application's specific requirements.
5. Cache Coordination: In distributed environments, Hibernate supports cache coordination mechanisms such as distributed caches or clustering solutions. These mechanisms ensure that cached data remains consistent and up-to-date across multiple application instances.
6. Transactional Integrity: Hibernate ensures transactional integrity when using caching. It automatically invalidates cached data when an update or delete operation occurs, preventing stale or incorrect data from being served to the application.

The Hibernate architecture includes many objects such as persistent object, session factory, transaction factory, connection factory, session, transaction etc.Hibernate framework uses many objects such as session factory, session, transaction etc. alongwith existing Java API such as JDBC (Java Database Connectivity), JTA (Java Transaction API) and JNDI (Java Naming Directory Interface).

4. Describe session factory,Session,Transaction objects?

For creating the first hibernate application, we must know the elements of Hibernate architecture. They are as follows:

SessionFactory

The SessionFactory is a factory of session and client of ConnectionProvider. It holds second level cache (optional) of data. The org.hibernate.SessionFactory interface provides factory method to get the object of Session.

Session

The session object provides an interface between the application and data stored in the database. It is a short-lived object and wraps the JDBC connection. It is factory of Transaction, Query and Criteria. It holds a first-level cache (mandatory) of data. The org.hibernate.Session interface provides methods to insert, update and delete the object. It also provides factory methods for Transaction, Query and Criteria.

Transaction

The transaction object specifies the atomic unit of work. It is optional. The org.hibernate.Transaction interface provides methods for transaction management. The Session object represents a single unit of work and acts as a gateway to interact with the database. It is lightweight and not thread-safe, so it should be created, used, and closed within a single thread context. A Session typically corresponds to a database connection.

ConnectionProvider

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

TransactionFactory

It is a factory of Transaction.