

LAB9(B.Bhagyasri)

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

(POM.XML) File:

```
<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>dummy</groupId>

    <artifactId>demoo</artifactId>

    <version>0.0.1-SNAPSHOT</version>

    <dependencies>

        <!-- Hibernate 4.3.6 Final -->

    <dependency>

        <groupId>org.hibernate</groupId>

        <artifactId>hibernate-core</artifactId>

        <version>5.4.29.Final</version>

    </dependency>

    <!-- Mysql Connector -->

    <dependency>

        <groupId>mysql</groupId>

        <artifactId>mysql-connector-java</artifactId>

        <version>8.0.28</version>

    </dependency>

    </dependencies>

</project>
```

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.MySQL5Dialect</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>
<property name="show_sql">true</property>
<property name = "hbm2ddl.auto">update</property>
<!-- Set the current session context -->
<property name="current_session_context_class">thread</property>
</session-factory>
</hibernate-configuration>

```

Program:

```

package CRUD;

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;
}
}
```

Implementation:

```
package CRUD;

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.SessionFactory;
import org.hibernate.cfg.Configuration;
@SuppressWarnings("deprecation")
public class ManageEmployee {
    private static SessionFactory factory;
    public static void main(String[] args) {

        try {
            factory = new Configuration().
                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("Bhagya", "sri", 1000);
        Integer empID2 = ME.addEmployee("Sri", "Mona", 5000);
```

```

Integer empID3 = ME.addEmployee("Nandu", "kaka", 10000);

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

/* Update employee's records */
ME.updateEmployee(empID1, 6000);

/* 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:

Hibernate: select next_val as id_val from hibernate_sequence for update

Hibernate: update hibernate_sequence set next_val= ? where next_val=?

Hibernate: insert into EMPLOYEE (first_name, last_name, salary, id) values (?, ?, ?, ?)

Hibernate: select next_val as id_val from hibernate_sequence for update

Hibernate: update hibernate_sequence set next_val= ? where next_val=?

Hibernate: insert into EMPLOYEE (first_name, last_name, salary, id) values (?, ?, ?, ?)

Hibernate: select next_val as id_val from hibernate_sequence for update

Hibernate: update hibernate_sequence set next_val= ? where next_val=?

Hibernate: insert into EMPLOYEE (first_name, last_name, salary, id) values (?, ?, ?, ?)

Hibernate: select employee0_.id as id1_0_, employee0_.first_name as first_na2_0_,
employee0_.last_name as last_nam3_0_, employee0_.salary as salary4_0_ from EMPLOYEE
employee0_

Hibernate: delete from EMPLOYEE where id=?

Hibernate: select employee0_.id as id1_0_, employee0_.first_name as first_na2_0_,
employee0_.last_name as last_nam3_0_, employee0_.salary as salary4_0_ from EMPLOYEE
employee0_

First Name: Zara Last Name: Ali Salary: 5000

First Name: Zara Last Name: Ali Salary: 1000

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

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: Zara Last Name: Ali Salary: 6000

First Name: Daisy Last Name: Das Salary: 5000

First Name: John Last Name: Paul Salary: 10000

First Name: Zara Last Name: Ali Salary: 6000

First Name: Daisy Last Name: Das Salary: 5000

First Name: John Last Name: Paul Salary: 10000

First Name: Zara Last Name: Ali Salary: 6000

First Name: Daisy Last Name: Das Salary: 5000

First Name: John Last Name: Paul Salary: 10000

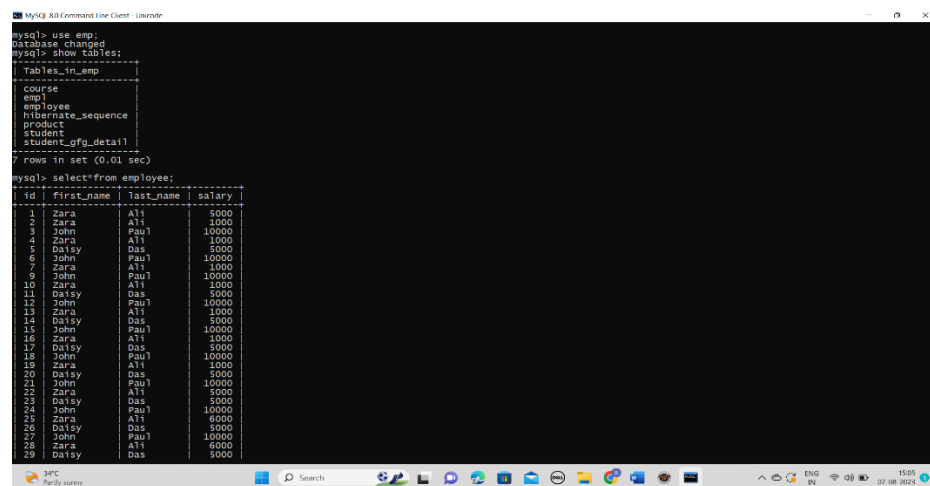
First Name: Zara Last Name: Ali Salary: 6000

First Name: Daisy Last Name: Das Salary: 5000

First Name: John Last Name: Paul Salary: 10000

First Name: Bhagya Last Name: sri Salary: 6000

Table:



```
mysql> use emp;
Database changed
mysql> show tables;
+-----+
| Tables_in_emp |
+-----+
| course        |
| emp           |
| employee      |
| hibernate_sequence |
| product       |
| student       |
| student_gfg_detail |
+-----+
7 rows in set (0.01 sec)

mysql> select * from employee;
```

id	first_name	last_name	salary
1	Zara	Ali	5000
2	Zara	Ali	1000
3	John	Paul	10000
4	Zara	Ali	1000
5	Daisy	Das	5000
6	John	Paul	10000
7	Zara	Ali	1000
9	John	Paul	10000
10	Zara	Ali	1000
11	Daisy	Das	5000
12	John	Paul	10000
13	Zara	Ali	1000
14	Daisy	Das	5000
15	John	Paul	10000
16	Zara	Ali	1000
17	Daisy	Das	5000
18	John	Paul	10000
19	Zara	Ali	1000
20	Daisy	Das	5000
21	John	Paul	10000
22	Zara	Ali	5000
23	Daisy	Das	5000
24	John	Paul	10000
25	Zara	Ali	6000
26	Daisy	Das	5000
27	John	Paul	10000
28	Zara	Ali	6000
29	Daisy	Das	5000

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

In Object-Relational Mapping (ORM) with Hibernate, two key configuration files are used to set up the framework's runtime environment and define the mappings between Java objects and database tables: **hibernate.cfg.xml** and ***.hbm.xml** files.

1. hibernate.cfg.xml:

The **hibernate.cfg.xml** file is the main configuration file for Hibernate. It contains various properties and settings that define how Hibernate interacts with the database and manages the persistence of Java objects. This file is usually placed in the classpath of the application and is read by Hibernate when the session factory is created. Here are some of the important properties that can be specified in the **hibernate.cfg.xml** file:

- **Connection Properties:** This includes the database connection details such as the database URL, username, password, and driver class. Hibernate uses this information to establish a connection to the underlying database.

- **Database Dialect:** The database dialect specifies the type of SQL that Hibernate should generate based on the target database. Each database has its own SQL variations, and the dialect ensures that the appropriate SQL is generated for the specific database being used.
- **Mapping Resources:** The **hibernate.cfg.xml** file lists the mapping resources or classes that define the relationship between Java objects and database tables. These mappings tell Hibernate how to persist objects and retrieve data from the database.
- **Caching Configuration:** Hibernate provides various caching strategies to improve performance. The **hibernate.cfg.xml** file allows you to configure different cache providers and cache settings to optimize the caching behavior.
- **Transaction Management:** Hibernate supports both programmatic and declarative transaction management. In the configuration file, you can specify the transaction manager class, which helps manage transactions when interacting with the database.
- **Data Source Configuration:** Instead of specifying connection properties, Hibernate can also be configured to use a data source. This allows for more advanced connection pool management and is particularly useful in enterprise applications.
- **Schema Generation and Validation:** Hibernate can automatically create database tables based on the mapped entity classes or validate the existing schema against the entity mappings. These options can be configured in the **hibernate.cfg.xml** file.

```
<!DOCTYPE hibernate-configuration PUBLIC
```

```
"-//Hibernate/Hibernate Configuration DTD//EN"
```

```
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
```

```
<hibernate-configuration>
```

```
<session-factory>
```

```
<!-- Database Connection Settings -->
```

```
<propertyname="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
```

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

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

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

```
<!-- Database Dialect -->
```

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

```
<!-- Mapping Resources -->
```

```

        <mapping resource="com/example/Entity1.hbm.xml"/>

        <mapping resource="com/example/Entity2.hbm.xml"/>

        <!-- Caching Configuration -->

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

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

        <!-- Transaction Management -->

        <property
name="hibernate.transaction.factory_class">org.hibernate.transaction.JDBCTransactionFact
ory</property>

        <!-- Schema Generation -->

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

    </session-factory>

</hibernate-configuration>

```

The **hibernate.cfg.xml** file acts as a central configuration file for Hibernate, and it is essential to provide the correct settings to ensure proper interaction with the database and efficient object persistence.

2.hibernate.hbm.xml:

The **hibernate.hbm.xml** files (also known as HBM mapping files) are used to define the mappings between Java classes (entities) and database tables. Each entity in the application typically has its own corresponding ***.hbm.xml** file. In these files, you define how the attributes of Java classes are mapped to the columns in database tables, the relationships between different entities, and any other mapping details.

Here's an example of a simple **User** entity and its corresponding **user.hbm.xml** file:

Java Entity Class (User.java):

javaCopy code

```

public class User {

    private Long id;

    private String username;

    private String email;

```

```
// Getters and Setters
}

user.hbm.xml:

xmlCopy code

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

<hibernate-mapping>

    <class name="com.example.User" table="user_table">

        <id name="id" column="user_id">
            <generator class="increment"/>
        </id>

        <property name="username" column="user_name"/>
        <property name="email" column="user_email"/>

    </class>

</hibernate-mapping>
```

In this example, the **user.hbm.xml** file defines the mapping for the **User** entity. It specifies the table name as "user_table" and maps the **id** property to the "user_id" column, the **username** property to the "user_name" column, and the **email** property to the "user_email" column in the database.

During runtime, Hibernate reads the **hibernate.hbm.xml** files and uses them to create the necessary SQL queries and manage the mapping between Java objects and database tables. This way, developers can work with objects in their code and let Hibernate handle the underlying database operations, making ORM development more efficient and less error-prone.

3.explain advantages of hql and caching in hibernate?

Hibernate Query Language (HQL) and caching are two powerful features of Hibernate that provide significant advantages for developing efficient and high-performance applications.

Advantages of HQL:

1. **Database Independence:** HQL abstracts the underlying SQL syntax, allowing developers to write queries using an object-oriented syntax that is independent of the database. This means that the same HQL query can be used with different databases without modification, making the application more portable and easier to maintain.

2. **Object-Oriented Querying:** HQL enables developers to perform object-oriented querying directly on the entity classes and their properties rather than dealing with raw SQL. This makes the code more readable and natural, especially when working with complex object graphs and relationships.
3. **Entity Fetching and Associations:** HQL provides powerful mechanisms to fetch associated entities and their collections. With HQL, you can easily navigate relationships between entities and fetch only the required data, which helps optimize the performance by reducing the number of database queries.
4. **Aggregate Functions:** HQL supports various aggregate functions like SUM, AVG, MAX, MIN, etc., which allow developers to perform calculations on the data directly in the query. This eliminates the need to fetch data into memory and perform calculations in the application code.
5. **Named Queries:** HQL allows developers to define named queries in mapping files or annotations. Named queries can be reused throughout the application, promoting code reusability and maintainability.
6. **Dynamic Querying:** HQL supports dynamic querying by using parameters in queries. This enables developers to build dynamic queries based on runtime conditions and user input, leading to more flexible and adaptable applications.
7. **Query Pagination:** HQL provides built-in support for pagination, allowing developers to fetch a limited set of results at a time. This is particularly useful when dealing with large result sets, as it improves performance and reduces memory usage.

Advantages of Caching in Hibernate:

1. **Performance Improvement:** Caching in Hibernate can significantly improve application performance by reducing the number of database queries. Frequently accessed data is stored in the cache, so subsequent requests for the same data can be served from the cache instead of querying the database, resulting in faster response times.
2. **Reduced Database Load:** Caching reduces the load on the database server since some data is already available in the cache. This can help prevent the database from becoming a performance bottleneck, especially in high-traffic applications.
3. **Better Scalability:** Caching enhances application scalability by reducing the need for repeated database access. It allows the application to handle more concurrent users without overburdening the database.
4. **Consistency and Isolation:** Hibernate provides caching strategies that ensure data consistency and isolation. When using appropriate cache settings, the cached data is automatically updated or invalidated when changes occur in the database, maintaining data integrity.
5. **Optimized Read Operations:** Caching is particularly effective for read-heavy applications. It enables Hibernate to retrieve data quickly from the cache, eliminating the need for repeated database reads, and thus reducing response times.

6. **Second-Level Caching:** Hibernate offers second-level caching, which is shared across sessions. This provides caching benefits across different users and sessions, resulting in a more efficient use of system resources.

It's important to use caching judiciously and consider the nature of the data and the application's requirements. While caching can greatly enhance performance, it should be applied carefully to avoid potential data consistency issues and excessive memory usage. Hibernate provides various caching strategies and configurations to tailor caching to specific application needs.

4. Describe session factory, Session, Transaction objects?

In Hibernate, the Session Factory, Session, and Transaction objects are fundamental components that play crucial roles in managing the interaction between Java objects and the underlying relational database. Understanding these objects is essential for effectively using Hibernate for data persistence.

1. Session Factory:

The Session Factory is a heavyweight, thread-safe, and immutable object in Hibernate. It represents a single shared instance of a Hibernate configuration and serves as a factory for creating Session objects. The Session Factory is typically built during the application's startup process and should be kept alive throughout the application's lifecycle.

Key characteristics and functions of the Session Factory:

- **Configuration:** The Session Factory encapsulates the configuration settings defined in the **hibernate.cfg.xml** file. It holds information about the database connection, entity mappings, caching, and other essential settings.
- **Thread-Safe:** The Session Factory is designed to be thread-safe, allowing multiple threads to access it concurrently. This ensures that multiple sessions can be created and used simultaneously by different parts of the application.
- **Optimization:** The Session Factory caches compiled mappings and queries, leading to improved performance as the overhead of parsing and processing mappings is minimized.
- **Session Creation:** When an application needs to interact with the database, it requests a Session from the Session Factory. The Session represents a single unit of work with the database.

2. Session:

The Session object represents a single unit of work with the database. It serves as the primary interface for performing database operations and managing the persistence of Java objects (entities). Each Session corresponds to a database connection and acts as a cache for loaded entities, ensuring that each entity is associated with a unique Session.

Key characteristics and functions of the Session:

- **Persistence Operations:** The Session provides methods to perform CRUD (Create, Read, Update, Delete) operations on entities. It allows developers to save, update, delete, and retrieve data from the database using object-oriented APIs.

- **Caching:** The Session manages an internal first-level cache (also known as the Session cache) that stores the loaded entities during a transaction. This cache reduces the number of database queries and improves performance.
- **Lazy Loading:** The Session enables lazy loading of associations, meaning that associated entities are loaded from the database only when accessed for the first time. This helps avoid unnecessary data retrieval and improves efficiency.
- **Transaction Management:** The Session can be used to manage transactions. It supports methods to begin, commit, and rollback transactions, ensuring that data modifications are performed atomically.

3. Transaction:

The Transaction object represents a single database transaction. It allows developers to group multiple database operations into a single unit of work, ensuring that either all operations succeed (commit) or none of them take effect (rollback).

Key characteristics and functions of the Transaction:

- **Atomicity:** A Transaction ensures atomicity by providing a mechanism to commit or rollback the changes made during the transaction. If any part of the transaction fails, the whole transaction is rolled back, ensuring data consistency.
- **Isolation:** Transactions provide isolation between concurrent database operations, preventing interference between multiple transactions.
- **Concurrency Control:** Transactions manage database locks to prevent conflicts between concurrent operations, ensuring data integrity and consistency.
- **Boundary Management:** The Transaction object allows developers to define the boundaries of a transaction by starting and ending it explicitly. This provides fine-grained control over the persistence operations.

In summary, the Session Factory is responsible for creating and managing Session objects, each representing a single unit of work with the database. Transactions ensure data integrity and isolation during database operations, allowing developers to handle multiple changes as an atomic operation. Understanding these objects is vital for effectively leveraging Hibernate's capabilities in managing object persistence and database interactions in ORM applications.