**Proof of Concept: Hibernate Architecture Implementation**

* **Introduction**

This Proof of Concept (POC) aims to demonstrate Hibernate's architecture and its key components in a Java-based application. Hibernate is a powerful Object-Relational Mapping (ORM) framework that simplifies database interactions by eliminating the need for manual SQL queries and JDBC operations. Through this POC, we will showcase how Hibernate manages database transactions efficiently using ORM principles.

**1. Hibernate**

Hibernate is an **ORM (Object-Relational Mapping)** framework for Java that simplifies database interactions by mapping Java objects to database tables. It abstracts the complexities of JDBC, allowing developers to perform database operations without writing extensive SQL queries.

**2. Hibernate Architecture**

The Hibernate architecture consists of several components:

**a. Configuration**

* It loads the configuration settings from hibernate.cfg.xml or hibernate.properties file.
* Contains database connection details and Hibernate properties.

**b. SessionFactory**

* It is a heavyweight object created once per application and provides Session objects.
* It is built using the configuration file.

**c. Session**

* A lightweight object created for every transaction.
* It is used to interact with the database, fetch data, and perform CRUD operations.

**d. Transaction**

* Manages database transactions (commit, rollback).
* Hibernate uses JDBC transactions or JTA for transaction management.

**e. Query (HQL/SQL)**

* Hibernate Query Language (HQL) is used to fetch and manipulate data in an object-oriented manner.
* Native SQL queries can also be executed.

**f. Criteria API**

* Provides an alternative to HQL for programmatic query construction.

**g. Hibernate Annotations & Entity Manager**

* Hibernate supports JPA (Java Persistence API) annotations for defining entity mappings.
* **Technologies Used**

1. Java (JDK 17) – Primary programming language
2. Hibernate ORM (6.3.1.Final) – ORM framework for database interactions
3. MySQL Database (or H2 for testing) – Relational database management
4. IntelliJ IDEA & Eclipse IDE – IDEs for Java development
5. Maven – Dependency and build management
6. SLF4J & Simple Logger – Logging framework

**Step 1: Setting Up the Project**

1. Create a Maven Project
   1. Open IntelliJ IDEA or Eclipse.
   2. Create a new Maven project.
   3. Define groupId as com.hibernate and artifactId as hibernate-architecture-implementation.

**Project Structure**

hibernate-poc/

├── src/main/java/com/hibernate/

│ ├── entity/User.java

│ ├── util/HibernateUtil.java

│ ├── dao/UserDAO.java

│ ├── HibernateApplication.java

├── src/main/resources/

│ ├── hibernate.cfg.xml

├── pom.xml

1. **Add Dependencies in pom.xml**

The pom.xml (Project Object Model) file is the configuration file for Maven, which is used to manage dependencies, build configurations, and project metadata in Java applications.

**example :-**

<?xml version="1.0" encoding="UTF-8"?>  
<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 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
  
 <groupId>org.hibernate</groupId>  
 <artifactId>hibernate-architecture-implementation</artifactId>  
 <version>1.0-SNAPSHOT</version>  
  
 <properties>  
 <maven.compiler.source>17</maven.compiler.source>  
 <maven.compiler.target>17</maven.compiler.target>  
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 </properties>  
  
 <dependencies>  
 <!-- Hibernate Core -->  
 <dependency>  
 <groupId>org.hibernate</groupId>  
 <artifactId>hibernate-core</artifactId>  
 <version>6.3.1.Final</version>  
 </dependency>  
  
 <!-- MySQL Driver -->  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>8.0.33</version>  
 </dependency>  
  
 <!-- Jakarta Persistence API -->  
 <dependency>  
 <groupId>jakarta.persistence</groupId>  
 <artifactId>jakarta.persistence-api</artifactId>  
 <version>3.1.0</version>  
 </dependency>  
  
 <!-- SLF4J API & Simple Logger (for logging) -->  
 <dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-api</artifactId>  
 <version>2.0.7</version>  
 </dependency>  
 <dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-simple</artifactId>  
 <version>2.0.7</version>  
 </dependency>  
 </dependencies>  
  
  
</project>

**Step 2: Configuring Hibernate**

The hibernate.cfg.xml file is a crucial configuration file in Hibernate, used to define database connection settings, Hibernate properties, and entity mappings. It is typically placed inside src/main/resources/ in a Spring Boot or Java-based Hibernate application.

**example :-**

Create hibernate.cfg.xml inside src/main/resources/.

<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  
 "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">  
  
<hibernate-configuration>  
 <session-factory>  
 <!-- Database Connection -->  
 <property name="hibernate.connection.driver\_class">com.mysql.cj.jdbc.Driver</property>  
 <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/hibernate\_db</property>  
 <property name="hibernate.connection.username">root</property>  
 <property name="hibernate.connection.password">password</property>  
  
 <!-- Hibernate Settings -->  
 <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>  
 <property name="hibernate.show\_sql">true</property>  
<!-- <property name="hibernate.hbm2ddl.auto">create</property>-->  
 <property name="hibernate.hbm2ddl.auto">update</property>  
  
 <!-- Entity Mapping -->  
 <mapping class="com.hibernate.entity.User"/>  
 </session-factory>  
</hibernate-configuration>

**Step 3: Defining the Entity Class**

The User.java class inside the com.hibernate.entity package is an entity class that represents the users table in the database. Hibernate uses JPA annotations to map this Java class to a relational database table.

**example :-**

Create User.java inside com.hibernate.entity package.

package com.hibernate.entity;  
  
import jakarta.persistence.\*;

@Entity  
@Table(name = "users")  
public class User {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Long id;  
 @Column(nullable = false)  
 private String name;  
 @Column(unique = true)  
 private String email;  
  
 public User() {}  
  
 public User(String name, String email) {  
 this.name = name;  
 this.email = email;  
 }  
  
 public Long getId() {  
 return id;  
 }  
  
 public void setId(Long id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public String getEmail() {  
 return email;  
 }  
  
 public void setEmail(String email) {  
 this.email = email;  
 }  
  
 // Override toString() method to print user-friendly output  
 @Override  
 public String toString() {  
 return "User{id=" + id + ", name='" + name + "', email='" + email + "'}";  
 }  
}

**Step 4: Creating Hibernate Utility Class**

The HibernateUtil.java class inside the com.hibernate.util package is a utility class responsible for managing the Hibernate SessionFactory. It provides a singleton implementation of SessionFactory, ensuring that only one instance of it is created and used throughout the application.

**example :-**

Create HibernateUtil.java inside com.hibernate.util package.

package com.hibernate.util;  
  
import org.hibernate.SessionFactory;  
import org.hibernate.cfg.Configuration;  
  
public class HibernateUtil {  
 private static final SessionFactory *sessionFactory* = *buildSessionFactory*();  
  
 private static SessionFactory buildSessionFactory() {  
 try {  
 return new Configuration().configure("hibernate.cfg.xml").buildSessionFactory();  
 } catch (Throwable ex) {  
 throw new ExceptionInInitializerError("SessionFactory creation failed: " + ex);  
 }  
 }  
  
 public static SessionFactory getSessionFactory() {  
 return *sessionFactory*;  
 }  
}

**Step 5: Implementing Data Access Object (DAO)**

The UserDAO.java class inside the com.hibernate.dao package is responsible for managing database operations for the User entity. It provides methods for CRUD operations (Create, Read, Update, Delete) using Hibernate ORM.

**example :-**

Create UserDAO.java inside com.hibernate.dao package.

package com.hibernate.dao;  
  
import com.hibernate.entity.User;  
import com.hibernate.util.HibernateUtil;  
import jakarta.persistence.criteria.CriteriaBuilder;  
import jakarta.persistence.criteria.CriteriaQuery;  
import jakarta.persistence.criteria.Root;  
import org.hibernate.Session;  
import org.hibernate.Transaction;  
import org.hibernate.query.Query;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
  
import java.util.List;  
  
  
public class UserDAO {  
 private static final Logger *logger* = LoggerFactory.*getLogger*(UserDAO.class);  
  
 public void saveUser(User user) {  
 Transaction transaction = null;  
 try (Session session = HibernateUtil.*getSessionFactory*().openSession()) {  
 transaction = session.beginTransaction();  
  
 Query<User> query = session.createQuery("FROM User WHERE email = :email", User.class);  
 query.setParameter("email", user.getEmail());  
 User existingUser = query.uniqueResult();  
  
 if (existingUser == null) {  
 session.persist(user);  
 transaction.commit();  
 *logger*.info("User saved: {}", user);  
 } else {  
 *logger*.warn("User with email {} already exists!", user.getEmail());  
 }  
  
 } catch (Exception e) {  
 if (transaction != null) transaction.rollback();  
 *logger*.error("Error saving user", e);  
 }  
 }  
  
  
 public List<User> getUsersUsingHQL() {  
 try (Session session = HibernateUtil.*getSessionFactory*().openSession()) {  
 List<User> users = session.createQuery("FROM User", User.class).list();  
 if (users.isEmpty()) {  
 *logger*.warn("No users found in the database.");  
 } else {  
 *logger*.info("Users : {}", users);  
 }  
 return users;  
 }  
 }  
  
  
 public List<User> getUsersUsingCriteria(String nameFilter) {  
 try (Session session = HibernateUtil.*getSessionFactory*().openSession()) {  
 CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();  
 CriteriaQuery<User> criteriaQuery = criteriaBuilder.createQuery(User.class);  
 Root<User> root = criteriaQuery.from(User.class);  
  
 // Apply filtering condition dynamically  
 criteriaQuery.select(root)  
 .where(criteriaBuilder.like(root.get("name"), "%" + nameFilter + "%"));  
  
 List<User> result = session.createQuery(criteriaQuery).getResultList();  
  
 // Log message if no results are found  
 if (result.isEmpty()) {  
 *logger*.warn("No users found with name containing: {}", nameFilter);  
 } else {  
 *logger*.info("Users: {}", result);  
 }  
  
 return result;  
 }  
 }  
  
  
  
 public void updateUser(Long id, String newName, String newEmail) {  
 Transaction transaction = null;  
 try (Session session = HibernateUtil.*getSessionFactory*().openSession()) {  
 transaction = session.beginTransaction();  
  
 User user = session.get(User.class, id);  
 if (user != null) {  
 // Check if the new email is already used by another user  
 Query<User> query = session.createQuery("FROM User WHERE email = :email AND id <> :id", User.class);  
 query.setParameter("email", newEmail);  
 query.setParameter("id", id);  
 User existingUser = query.uniqueResult();  
  
 if (existingUser == null) {  
 user.setName(newName);  
 user.setEmail(newEmail);  
 session.merge(user);  
 transaction.commit();  
 *logger*.info("User updated successfully: {}", user);  
 } else {  
 *logger*.warn("Email '{}' is already taken by another user!", newEmail);  
 transaction.rollback(); // Fix: Rollback transaction if email exists  
 }  
 } else {  
 *logger*.warn("User with ID {} not found!", id);  
 transaction.rollback();  
 }  
 } catch (Exception e) {  
 if (transaction != null) transaction.rollback();  
 *logger*.error("Error updating user with ID {}: {}", id, e.getMessage(), e);  
 }  
 }  
  
 public void deleteUser(Long id) {  
 Transaction transaction = null;  
 try (Session session = HibernateUtil.*getSessionFactory*().openSession()) {  
 transaction = session.beginTransaction();  
 User user = session.get(User.class, id);  
  
 if (user != null) {  
 session.remove(user);  
 transaction.commit();  
 *logger*.info("User with ID {} removed from the database.", id);  
 } else {  
 *logger*.warn("User with ID {} not found. No deletion performed.", id);  
 transaction.rollback(); // Fix: Rollback transaction if user not found  
 }  
 } catch (Exception e) {  
 if (transaction != null) transaction.rollback();  
 *logger*.error("Error deleting user with ID {}: {}", id, e.getMessage(), e);  
 }  
 }  
}

**Step 6: Running the Application**

The HibernateApplication.java class serves as the entry point for executing Hibernate operations in your project. It initializes the UserDAO class and performs various CRUD operations using Hibernate ORM.

**example :-**

Create HibernateApplication.java in com.hibernate package.

package com.hibernate;  
  
import com.hibernate.dao.UserDAO;  
import com.hibernate.entity.User;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
  
import java.util.List;  
  
public class HibernateApplication {  
 private static final Logger *logger* = LoggerFactory.*getLogger*(HibernateApplication.class);  
  
 public static void main(String[] args) {  
 UserDAO userDAO = new UserDAO();  
  
 /\* //Insert Users  
 userDAO.saveUser(new User("Arjun Hambarde", "arjunhambarde1208@gmail.com"));  
 userDAO.saveUser(new User("Akshay Hamde", "akshay1234@gmail.com"));  
 \*/  
 /\*// Update User  
 userDAO.updateUser(1L, "Pranav Patwekar", "pranav1245@gmail.com");  
\*/  
  
 /\* // Delete User  
 userDAO.deleteUser(1L);  
\*/  
  
 // Retrieve Users Using HQL  
 List<User> hqlUsers = userDAO.getUsersUsingHQL();  
  
 /\*// Retrieve Users Using Criteria API  
 String searchName = "Arjun Hambarde"; // Dynamic search parameter  
 List<User> criteriaUsers = userDAO.getUsersUsingCriteria(searchName);  
 \*/  
 }  
}

* **Conclusion**

This Proof of Concept successfully demonstrates Hibernate's architecture and its key components in a Java-based application. By leveraging Hibernate ORM, the application eliminates the need for manual SQL queries and JDBC operations, simplifying database interactions. The implementation showcases essential Hibernate features, including SessionFactory for efficient session management, Session for database connectivity, Transaction management for data integrity, and HQL & Criteria API for flexible querying. Additionally, automatic schema generation and proper entity mapping highlight Hibernate’s capabilities in handling object-relational mapping. This POC serves as a foundational reference for integrating Hibernate into Java applications, ensuring scalable and maintainable database interactions.

**THE END**