**JPA using Hibernate:**

**Introduction, Entities, Managing Entities, Querying Entities, Entity Relationships.**

**(Model Question Paper –I)**

**5. a. Define JPA. Explain the properties of an Entities objects. Design and implement a simple entity program for student class.**

**JPA Definition:**

**JPA (Java Persistence API)** is a specification for accessing, persisting, and managing data between Java objects and relational databases. It provides a standardized way to work with data persistence in Java applications, offering an object-oriented abstraction over traditional database interactions.

**Properties of an Entity Object:**

In the context of JPA, an **Entity** represents a persistent data entity that is typically mapped to a database table. Here are the key properties of an Entity object:

1. **@Entity Annotation**: An entity class in JPA is annotated with @Entity. This annotation marks the class as a persistent entity, indicating that instances of this class will be stored in the database.
2. **Persistent Identity**: Each entity object has a unique identity that is represented by its primary key. The @Id annotation specifies the primary key of the entity.
3. **Attributes Mapping**: Entity attributes are mapped to database columns using annotations such as @Column. This allows JPA to map Java fields to corresponding database columns.
4. **Relationship Mapping**: Entities can have relationships with other entities (One-to-One, One-to-Many, Many-to-One, Many-to-Many). These relationships are defined using annotations like @OneToOne, @OneToMany, @ManyToOne, @ManyToMany.
5. **Lifecycle Callbacks**: JPA provides lifecycle callbacks (@PrePersist, @PostPersist, @PreUpdate, @PostUpdate, @PreRemove, @PostRemove, @PostLoad) that allow developers to hook into entity lifecycle events and perform actions.
6. **Entity Inheritance**: JPA supports entity inheritance strategies (@Inheritance) where entities can inherit properties and behaviors from a superclass.

**Example: Design and Implement a Simple Entity Program for Student Class:**

import javax.persistence.\*;

@Entity

@Table(name = "students")

public class Student {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    @Column(name = "first\_name")

    private String firstName;

    @Column(name = "last\_name")

    private String lastName;

    private int age;

    // Constructors, getters, and setters

    public Student() {

    }

    public Student(String firstName, String lastName, int age) {

        this.firstName = firstName;

        this.lastName = lastName;

        this.age = age;

    }

    // Getters and setters

    public Long getId() {

        return id;

    }

    public void setId(Long id) {

        this.id = id;

    }

    public String getFirstName() {

        return firstName;

    }

    public void setFirstName(String firstName) {

        this.firstName = firstName;

    }

    public String getLastName() {

        return lastName;

    }

    public void setLastName(String lastName) {

        this.lastName = lastName;

    }

    public int getAge() {

        return age;

    }

    public void setAge(int age) {

        this.age = age;

    }

    @Override

    public String toString() {

        return "Student{" +

                "id=" + id +

                ", firstName='" + firstName + '\'' +

                ", lastName='" + lastName + '\'' +

                ", age=" + age +

                '}';

    }

}

**5.b. Explain One-to-One Entity Relation mapping both in unidirectional and bidirectional way with an example.**

In JPA, **One-to-One** relationship mapping refers to a relationship where one entity (source) is associated with exactly one instance of another entity (target), and vice versa. Let's explore both unidirectional and bidirectional mappings with an example:

#### Unidirectional One-to-One Mapping Example:

In a unidirectional mapping, only one side of the relationship knows about the other.

import javax.persistence.\*;

@Entity

@Table(name = "addresses")

public class Address {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    private String street;

    private String city;

    // Getters and Setters, Constructors

    // Constructors

    public Address() {

    }

    public Address(String street, String city) {

        this.street = street;

        this.city = city;

    }

    // Getters and Setters

    public Long getId() {

        return id;

    }

    public void setId(Long id) {

        this.id = id;

    }

    public String getStreet() {

        return street;

    }

    public void setStreet(String street) {

        this.street = street;

    }

    public String getCity() {

        return city;

    }

    public void setCity(String city) {

        this.city = city;

    }

    // toString() method (optional for debugging)

    @Override

    public String toString() {

        return "Address{" +

                "id=" + id +

                ", street='" + street + '\'' +

                ", city='" + city + '\'' +

                '}';

    }

}

**Student Entity:**

import javax.persistence.\*;

@Entity

@Table(name = "students")

public class Student {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    private String firstName;

    private String lastName;

    @OneToOne

    @JoinColumn(name = "address\_id")

    private Address address;

    // Constructors

    public Student() {

    }

    public Student(String firstName, String lastName) {

        this.firstName = firstName;

        this.lastName = lastName;

    }

    // Getters and Setters

    public Long getId() {

        return id;

    }

    public void setId(Long id) {

        this.id = id;

    }

    public String getFirstName() {

        return firstName;

    }

    public void setFirstName(String firstName) {

        this.firstName = firstName;

    }

    public String getLastName() {

        return lastName;

    }

    public void setLastName(String lastName) {

        this.lastName = lastName;

    }

    public Address getAddress() {

        return address;

    }

    public void setAddress(Address address) {

        this.address = address;

    }

    // toString() method (optional for debugging)

    @Override

    public String toString() {

        return "Student{" +

                "id=" + id +

                ", firstName='" + firstName + '\'' +

                ", lastName='" + lastName + '\'' +

                ", address=" + address +

                '}';

    }

}

* **Address Entity**: Represents the Address entity with @Entity and @Table annotations, and an id, street, and city attributes.
* **Student Entity**: Represents the Student entity with @Entity and @Table annotations, and an id, firstName, lastName, and address attribute.
* **@OneToOne**: Defines a unidirectional one-to-one relationship from Student to Address. The address field in Student entity maps to the address\_id foreign key column in the students table.
* **@JoinColumn**: Specifies the name of the foreign key column (address\_id) in the students table that references the id column in the addresses table.

#### Bidirectional One-to-One Mapping Example:

In a bidirectional mapping, both sides of the relationship know about each other.

* Address Entity: Includes an additional @OneToOne mapped by student field, establishing a bidirectional relationship with Student.
* Student Entity: Continues to have a unidirectional @OneToOne relationship to Address.

In this example:

* Unidirectional: Only Student knows about Address.
* Bidirectional: Both Student and Address know about each other

import javax.persistence.\*;

@Entity

@Table(name = "addresses")

public class Address {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    private String street;

    private String city;

    @OneToOne(mappedBy = "address")

    private Student student;

    // Constructors, Getters and Setters

    // Constructors

    public Address() {

    }

    public Address(String street, String city) {

        this.street = street;

        this.city = city;

    }

    // Getters and Setters

    public Long getId() {

        return id;

    }

    public void setId(Long id) {

        this.id = id;

    }

    public String getStreet() {

        return street;

    }

    public void setStreet(String street) {

        this.street = street;

    }

    public String getCity() {

        return city;

    }

    public void setCity(String city) {

        this.city = city;

    }

    public Student getStudent() {

        return student;

    }

    public void setStudent(Student student) {

        this.student = student;

    }

    // toString() method (optional for debugging)

    @Override

    public String toString() {

        return "Address{" +

                "id=" + id +

                ", street='" + street + '\'' +

                ", city='" + city + '\'' +

                '}';

    }

}

**Student Entity (Updated)**

import javax.persistence.\*;

@Entity

@Table(name = "students")

public class Student {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    private String firstName;

    private String lastName;

    @OneToOne

    @JoinColumn(name = "address\_id")

    private Address address;

    // Constructors, Getters and Setters

    // Constructors

    public Student() {

    }

    public Student(String firstName, String lastName) {

        this.firstName = firstName;

        this.lastName = lastName;

    }

    // Getters and Setters

    public Long getId() {

        return id;

    }

    public void setId(Long id) {

        this.id = id;

    }

    public String getFirstName() {

        return firstName;

    }

    public void setFirstName(String firstName) {

        this.firstName = firstName;

    }

    public String getLastName() {

        return lastName;

    }

    public void setLastName(String lastName) {

        this.lastName = lastName;

    }

    public Address getAddress() {

        return address;

    }

    public void setAddress(Address address) {

        this.address = address;

        if (address != null) {

            address.setStudent(this);

        }

    }

    // toString() method (optional for debugging)

    @Override

    public String toString() {

        return "Student{" +

                "id=" + id +

                ", firstName='" + firstName + '\'' +

                ", lastName='" + lastName + '\'' +

                ", address=" + address +

                '}';

    }

}

**Entity Relationship:**

Entity relationships in JPA are crucial for modeling real-world data and interactions between different entities. JPA provides several annotations to define these relationships, including @OneToOne, @OneToMany, @ManyToOne, and @ManyToMany. Each of these annotations can be used to specify the nature of the relationship between entities.

**Types of Entity Relationships**

* 1. One-to-One Relationship
  2. One-to-Many Relationship
  3. Many-to-One Relationship
  4. Many-to-Many Relationship

### 1. One-to-One Relationship

A one-to-one relationship is where each instance of an entity is associated with exactly one instance of another entity.

@Entity

public class User {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    private String name;

    @OneToOne(cascade = CascadeType.ALL)

    @JoinColumn(name = "address\_id", referencedColumnName = "id")

    private Address address;

    // Getters and Setters

}

@Entity

public class Address {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    private String street;

    private String city;

    @OneToOne(mappedBy = "address")

    private User user;

    // Getters and Setters

}

### 2. One-to-Many Relationship

A one-to-many relationship is where each instance of an entity is associated with multiple instances of another entity.

@Entity

public class Department {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    private String name;

    @OneToMany(mappedBy = "department", cascade = CascadeType.ALL)

    private List<Employee> employees;

    // Getters and Setters

}

@Entity

public class Employee {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    private String name;

    @ManyToOne

    @JoinColumn(name = "department\_id")

    private Department department;

    // Getters and Setters

}

### 3. Many-to-One Relationship

A many-to-one relationship is the inverse of a one-to-many relationship, where multiple instances of one entity are associated with one instance of another entity.

#### Example: Employee and Department (inversed)

@Entity

public class Employee {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    private String name;

    @ManyToOne

    @JoinColumn(name = "department\_id")

    private Department department;

    // Getters and Setters

}

@Entity

public class Department {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    private String name;

    @OneToMany(mappedBy = "department", cascade = CascadeType.ALL)

    private List<Employee> employees;

    // Getters and Setters

}

### 4. Many-to-Many Relationship

A many-to-many relationship is where multiple instances of one entity are associated with multiple instances of another entity, often requiring a join table to manage the relationship.

#### Example: Student and Course

@Entity

public class Student {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    private String name;

    @ManyToMany

    @JoinTable(

        name = "student\_course",

        joinColumns = @JoinColumn(name = "student\_id"),

        inverseJoinColumns = @JoinColumn(name = "course\_id")

    )

    private Set<Course> courses;

    // Getters and Setters

}

@Entity

public class Course {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    private String name;

    @ManyToMany(mappedBy = "courses")

    private Set<Student> students;

    // Getters and Setters

}

**Explain the steps for creating the Hibernate application:**

Creating a Hibernate application involves several steps to set up the project, configure Hibernate, define entity classes, manage transactions, and perform database operations. Below are the general steps for creating a Hibernate application:

**Step 1: Set Up Your Development Environment**

1. **Setup Java Development Kit (JDK)**: Ensure you have JDK installed on your system. Hibernate typically works with Java SE 8 or later versions.
2. **Setup Build Tool**: Use a build tool like Maven or Gradle to manage dependencies and build your project. Here, we'll use Maven as an example.
   * **Maven**: Download and install Maven from [Apache Maven](https://maven.apache.org/download.cgi) and configure it according to your environment.

**Step 2: Create a Maven Project**

1. **Create Maven Project**: Use Maven to create a new project structure. You can do this via Maven archetype or manually create the directory structure.

bash

Copy code

mvn archetype:generate -DgroupId=com.example -DartifactId=hibernate-demo -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false

This command creates a basic Maven project with the groupId com.example and artifactId hibernate-demo.

1. **Project Structure**: After creating the project, you will have a directory structure similar to this:

less

Copy code

hibernate-demo

├── src

│ ├── main

│ │ ├── java // Java source files

│ │ └── resources // Resource files

│ └── test // Test source files

└── pom.xml // Maven project configuration file

**Step 3: Configure Hibernate**

1. **Add Hibernate Dependencies**: Open pom.xml and add the necessary dependencies for Hibernate, JDBC driver (e.g., MySQL, PostgreSQL), and connection pool (e.g., HikariCP).

xml

Copy code

<dependencies>

<!-- Hibernate ORM -->

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>5.6.4.Final</version>

</dependency>

<!-- Database Driver and Connection Pool -->

<dependency>

<groupId>mysql</groupId>

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

<version>8.0.28</version>

</dependency>

<dependency>

<groupId>com.zaxxer</groupId>

<artifactId>HikariCP</artifactId>

<version>5.0.2</version>

</dependency>

</dependencies>

1. **Hibernate Configuration File**: Create a Hibernate configuration file (hibernate.cfg.xml) under src/main/resources.

xml

Copy code

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

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

<hibernate-configuration>

<session-factory>

<!-- Database connection settings -->

<property name="hibernate.connection.driver\_class">com.mysql.cj.jdbc.Driver</property>

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

<property name="hibernate.connection.username">your\_username</property>

<property name="hibernate.connection.password">your\_password</property>

<!-- JDBC connection pool settings -->

<property name="hibernate.connection.provider\_class">org.hibernate.hikaricp.internal.HikariCPConnectionProvider</property>

<property name="hibernate.hikari.dataSourceClassName">com.mysql.cj.jdbc.MysqlDataSource</property>

<property name="hibernate.hikari.dataSource.url">jdbc:mysql://localhost:3306/your\_database</property>

<property name="hibernate.hikari.dataSource.user">your\_username</property>

<property name="hibernate.hikari.dataSource.password">your\_password</property>

<!-- Hibernate dialect -->

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

<!-- Enable Hibernate's automatic session context management -->

<property name="hibernate.current\_session\_context\_class">thread</property>

<!-- Echo all executed SQL to stdout -->

<property name="hibernate.show\_sql">true</property>

<property name="hibernate.format\_sql">true</property>

<!-- Drop and re-create the database schema on startup -->

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

<!-- Mapping files -->

<!-- Add your entity classes here -->

<mapping class="com.example.model.User"/>

</session-factory>

</hibernate-configuration>

* + Adjust hibernate.connection.url, username, password, and other properties according to your database setup.

**Step 4: Define Entity Classes**

1. **Create Entity Classes**: Define your entity classes under src/main/java.

java

Copy code

package com.example.model;

import javax.persistence.\*;

@Entity

@Table(name = "users")

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

// Getters and Setters

// Constructors

// Optionally, override toString(), equals(), and hashCode() methods

}

* + Annotate your entity classes with @Entity and @Table(name = "table\_name").
  + Use @Id for primary key, @GeneratedValue for auto-generation strategy, and other JPA annotations as needed (@Column, @ManyToOne, @OneToMany, etc.).

**Step 5: Perform Database Operations**

1. **Use Hibernate APIs**: Use Hibernate's Session or EntityManager API to perform CRUD operations on entities.

java

Copy code

package com.example;

import com.example.model.User;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

public class MainApp {

public static void main(String[] args) {

// Create session factory (once per application)

SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();

// Create session

Session session = sessionFactory.openSession();

// Begin transaction

Transaction transaction = session.beginTransaction();

// Perform database operations

User user = new User();

user.setName("John Doe");

session.save(user); // Insert

User retrievedUser = session.get(User.class, user.getId()); // Select

retrievedUser.setName("Jane Doe");

session.update(retrievedUser); // Update

session.delete(retrievedUser); // Delete

// Commit transaction

transaction.commit();

// Close session

session.close();

// Close session factory (on application shutdown)

sessionFactory.close();

}

}

* + **Session vs EntityManager**: Hibernate provides both Session (traditional Hibernate API) and EntityManager (JPA API). Choose one based on your application's requirements and design.

**Step 6: Run and Test Your Application**

1. **Run Your Application**: Compile and run your application using your IDE or Maven command-line tools.

bash

Copy code

mvn clean install

1. **Verify Database Operations**: Check your database tables to ensure that CRUD operations are reflected correctly.

**Additional Steps (Optional)**

1. **Logging and Error Handling**: Implement logging and error handling mechanisms to monitor and manage database operations effectively.
2. **Advanced Configurations**: Explore advanced Hibernate configurations for caching, performance tuning, transaction management, etc.
3. **Integration with Spring Framework**: Integrate Hibernate with the Spring framework for enhanced dependency injection and declarative transaction management.