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 Entity Objects

Entity objects in JPA represent tables in a relational database. Key properties of entity objects include:

1. Entity Class: A Java class representing a table in a database.
2. Primary Key: Every entity must have a primary key which uniquely identifies each instance of the entity. This is typically annotated with @Id.
3. Column Mappings: Fields in the entity class are mapped to columns in the table using annotations like @Column.
4. Relationships: Entities can have relationships with other entities, such as OneToMany, ManyToOne, OneToOne, and ManyToMany.
5. Lifecycle Callbacks: Methods in entity classes can be annotated to receive callbacks for entity lifecycle events like @PrePersist, @PostPersist, @PreRemove, @PostRemove, etc.
6. Annotations: JPA uses annotations to provide metadata about the entity class. Common annotations include @Entity, @Table, @Column, @Id, @GeneratedValue, etc.

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

Simple Entity Program for Student Class

Here's a simple example of a Student entity class in a JPA-based application:

Dependencies (Gradle)

Ensure you have the following dependencies in your build.gradle file:

groovy

Copy code

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    implementation 'com.h2database:h2' // For an in-memory database
}
```

Entity Class

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table(name = "students")
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private int age;
    private String email;

    // Constructors
    public Student() {}

    public Student(String name, int age, String email) {
        this.name = name;
```

```
    this.age = age;
    this.email = email;
}

// Getters and Setters
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 int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

public String getEmail() {
    return email;
}
```

```
    public void setEmail(String email) {
        this.email = email;
    }
}
```

Repository Interface

```
import org.springframework.data.jpa.repository.JpaRepository;
public interface StudentRepository extends JpaRepository<Student, Long> {
}
```

Main Application Class

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class StudentApplication implements CommandLineRunner {
    @Autowired
    private StudentRepository studentRepository;
    public static void main(String[] args) {
        SpringApplication.run(StudentApplication.class, args);
    }
    @Override
    public void run(String... args) throws Exception {
        // Save a few students
        studentRepository.save(new Student("John Doe", 20, "john.doe@example.com"));
        studentRepository.save(new Student("Jane Doe", 22, "jane.doe@example.com"));
        // Fetch all students
        studentRepository.findAll().forEach(student -> {
            System.out.println(student.getName());
        });
    }
}
```

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() {
```

```java
    }

    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

```java
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
```

```java
    // 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)

```java
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.

```java
@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.

```java
@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 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`.

2. **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>
```

2. **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</prop
erty>
        <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.Hika
riCPConnectionProvider</property>
        <property
name="hibernate.hikari.dataSourceClassName">com.mysql.cj.jdbc.MysqlDataSource</
property>
        <property
name="hibernate.hikari.dataSource.url">jdbc:mysql://localhost:3306/your_databas
e</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
   ```

2. **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.

Entity relationships and querying entities are fundamental concepts in object-relational mapping (ORM) frameworks such as Hibernate or JPA (Java Persistence API). Let's explore these concepts with a program example using JPA.

i. Entity Relationship

Entity relationships define how entities (tables in a database) relate to each other. The most common types of relationships are:

1. One-to-One: Each instance of an entity relates to one instance of another entity.

2. One-to-Many: One instance of an entity relates to multiple instances of another entity.
3. Many-to-One: Multiple instances of an entity relate to one instance of another entity.
4. Many-to-Many: Multiple instances of an entity relate to multiple instances of another entity.

ii. Querying Entities

Querying entities involves retrieving data from the database using the ORM framework. JPA provides a powerful query language called JPQL (Java Persistence Query Language) for this purpose.

Example Program

Let's create a simple program to demonstrate these concepts. We will have two entities: Author and Book, with a one-to-many relationship where one author can write multiple books.

1. Define Entities

Author.java

```
import javax.persistence.*;
import java.util.HashSet;
import java.util.Set;
@Entity
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    @OneToMany(mappedBy = "author", cascade = CascadeType.ALL, orphanRemoval = true)
    private Set<Book> books = new HashSet<>();
    // Constructors, getters, and setters
    public Author() {}
    public Author(String name) {
        this.name = name;
    }
    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 Set<Book> getBooks() {  return books;}

    public void setBooks(Set<Book> books) {  this.books = books;   }

    public void addBook(Book book) {
        books.add(book);
        book.setAuthor(this);
    }

    public void removeBook(Book book) {
        books.remove(book);
        book.setAuthor(null);
    }
}
```

Book.java

```
import javax.persistence.*;
@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    @ManyToOne
    @JoinColumn(name = "author_id")
    private Author author;
    // Constructors, getters, and setters
    public Book() {}
    public Book(String title) {   this.title = title;  }
    public Long getId() {return id;  }
    public void setId(Long id) {this.id = id;}
    public String getTitle() {  return title;}
    public void setTitle(String title) {  this.title = title;  }
    public Author getAuthor() {return author;  }
```

```
    public void setAuthor(Author author) {    this.author = author;    }
```

2. Configure Persistence

Create a persistence.xml file to configure the JPA persistence unit.

persistence.xml

xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="2.1">
    <persistence-unit name="examplePU">
        <class>Author</class>
        <class>Book</class>
        <properties>
            <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/exampledb"/>
            <property name="javax.persistence.jdbc.user" value="root"/>
            <property name="javax.persistence.jdbc.password" value="password"/>
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5Dialect"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
            <property name="hibernate.show_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

3. Main Application

Main.java

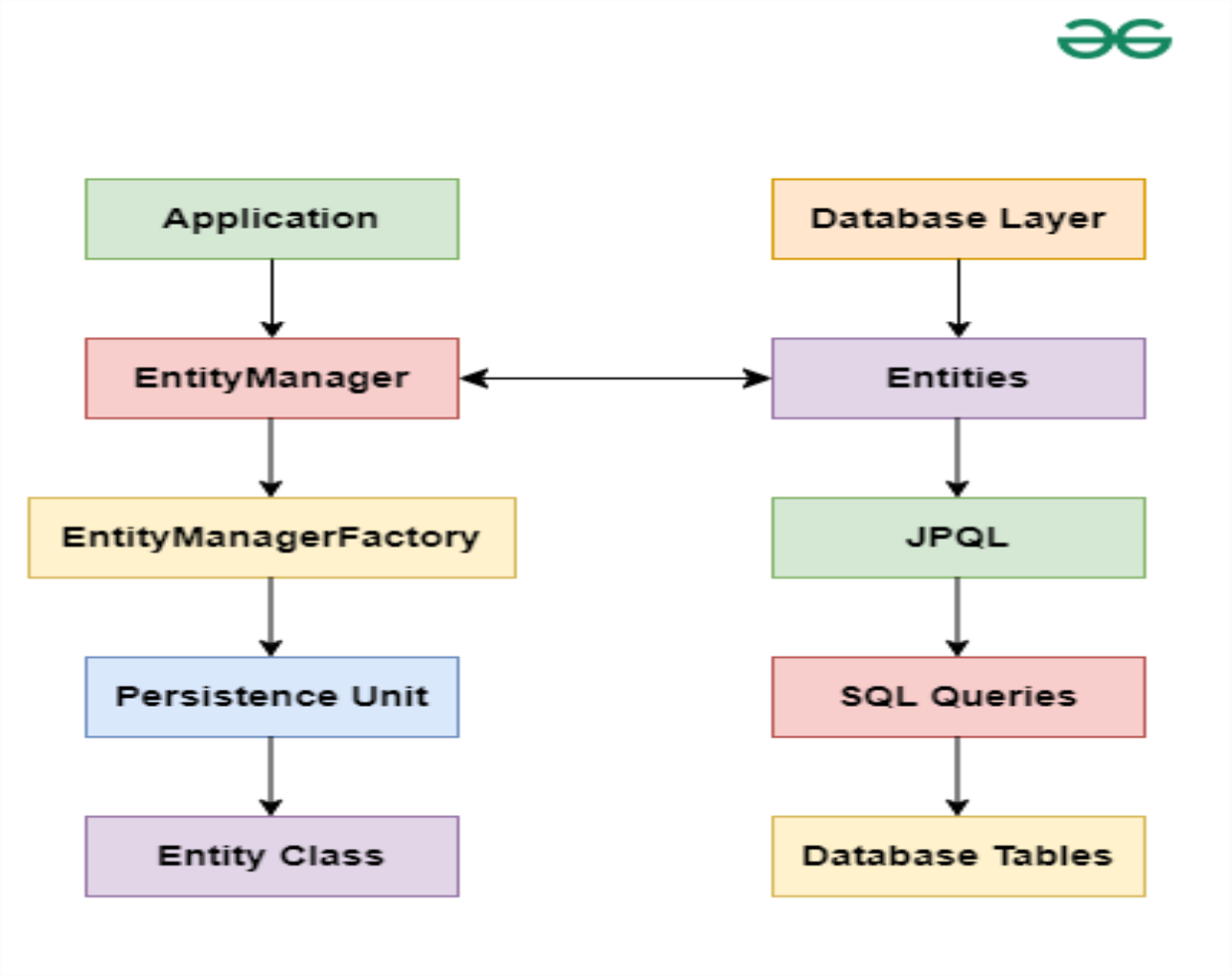
```
import javax.persistence.*;
import java.util.List;
public class Main {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("examplePU");
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();
        Author author = new Author("J.K. Rowling");
        Book book1 = new Book("Harry Potter and the Philosopher's Stone");
        Book book2 = new Book("Harry Potter and the Chamber of Secrets");
```

```
        author.addBook(book1);
        author.addBook(book2);
        em.persist(author);
        em.getTransaction().commit();
        // Querying Entities
        TypedQuery<Author> query = em.createQuery("SELECT a FROM Author a WHERE a.name =
:name", Author.class);
        query.setParameter("name", "J.K. Rowling");
        List<Author> authors = query.getResultList();
        for (Author a : authors) {
            System.out.println("Author: " + a.getName());
            for (Book b : a.getBooks()) {System.out.println("Book: " + b.getTitle());}}
        em.close();
        emf.close();}}
```

Explanation

1. Entity Relationship: The Author entity has a one-to-many relationship with the Book entity. This is defined using the @OneToMany and @ManyToOne annotations.
2. Querying Entities: We use a JPQL query to retrieve authors by name and print their books.

Explain JPA Architecture and how it is related to ORM.



PA architecture consists of several layers and components that work together to provide ORM functionality:

1. Entity Classes: The Java classes annotated with JPA annotations that define the mapping between the Java objects and the database tables.
2. Annotations: Used to define the metadata for ORM, such as @Entity, @Table, @Id, @Column, @OneToMany, @ManyToOne, etc.
3. Persistence Unit: Defined in persistence.xml file, it contains configuration details like data source, provider, and entity classes.
4. EntityManager and EntityManagerFactory: The core components that manage the entities. EntityManager handles the operations, and EntityManagerFactory is responsible for creating instances of EntityManager.
5. Persistence Context: Managed by EntityManager, it holds the entity instances. There are two types of contexts:
    - Transaction-scoped: Tied to a specific transaction.
    - Extended-scoped: Can span across multiple transactions.
6. JPA Provider: The implementation of the JPA specification, such as Hibernate, EclipseLink, or OpenJPA. The provider translates the JPA calls into database-specific operations.

7. JPQL and Criteria API: JPQL allows querying entities using a SQL-like syntax, while Criteria API provides a programmatic way to create queries.

How JPA is Related to ORM

ORM is a technique for mapping between the objects in the application and the relational database tables. JPA provides a standardized way to perform ORM in Java applications. Here's how JPA facilitates ORM:

1. Mapping: JPA uses annotations to map Java classes to database tables and Java fields to table columns. This abstraction allows developers to work with Java objects instead of SQL queries.
2. CRUD Operations: JPA provides methods for create, read, update, and delete operations on entities, managing the underlying SQL statements.
3. Transaction Management: JPA integrates with Java Transaction API (JTA) to manage transactions. The EntityManager handles transactions, ensuring data consistency and integrity.
4. Caching: JPA supports caching mechanisms to improve performance by reducing database access.
5. Relationships: JPA manages relationships between entities, supporting one-to-one, one-to-many, many-to-one, and many-to-many relationships.
6. Querying: JPA offers JPQL and the Criteria API for querying entities. JPQL is similar to SQL but operates on the entity model, while the Criteria API allows building type-safe queries programmatically.

Explain the purpose of EntityManager in JPA.

The EntityManager is a core component in the Java Persistence API (JPA) that manages the lifecycle of entities, handles CRUD operations, and provides query capabilities. Here's an in-depth look at the purpose and functions of the EntityManager:

Purpose of EntityManager in JPA

1. Managing Entity Lifecycle:
    - Persisting Entities: The EntityManager is responsible for saving new entities to the database.
    - Retrieving Entities: It provides methods to find and query entities from the database.
    - Updating Entities: The EntityManager tracks changes to entities and synchronizes those changes with the database.
    - Removing Entities: It allows for the deletion of entities from the database.
2. Transaction Management:
    - The EntityManager works with the Java Transaction API (JTA) to manage transactions. It ensures that operations are executed within a transaction context, providing consistency and atomicity.
3. Query Execution:
    - The EntityManager provides APIs for executing JPQL (Java Persistence Query Language) and native SQL queries to retrieve entities and perform bulk operations.

4. Entity Caching:
    - It maintains a persistence context (first-level cache) to manage entities' states and ensure that each entity instance is unique within the context. This improves performance by minimizing database access.

Demonstrate the concept of crud operations on data in JPA.

```
import javax.persistence.*;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Create EntityManagerFactory
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("examplePU");

        // Create EntityManager
        EntityManager em = emf.createEntityManager();

        // Perform CRUD operations
        createAuthor(em);
        readAuthor(em);
        updateAuthor(em);
        deleteAuthor(em);

        // Close EntityManager and EntityManagerFactory
        em.close();
        emf.close();
    }

    private static void createAuthor(EntityManager em) {
        EntityTransaction transaction = em.getTransaction();
        transaction.begin();

        Author author = new Author("J.K. Rowling");
        Book book1 = new Book("Harry Potter and the Philosopher's Stone");
        Book book2 = new Book("Harry Potter and the Chamber of Secrets");
```

```
        author.addBook(book1);
        author.addBook(book2);

        em.persist(author);

        transaction.commit();
        System.out.println("Author and books created successfully.");
    }

    private static void readAuthor(EntityManager em) {
        TypedQuery<Author> query = em.createQuery("SELECT a FROM Author a WHERE a.name =
:name", Author.class);
        query.setParameter("name", "J.K. Rowling");
        List<Author> authors = query.getResultList();

        for (Author a : authors) {
            System.out.println("Author: " + a.getName());
            for (Book b : a.getBooks()) {
                System.out.println("Book: " + b.getTitle());
            }
        }
    }

    private static void updateAuthor(EntityManager em) {
        EntityTransaction transaction = em.getTransaction();
        transaction.begin();

        TypedQuery<Author> query = em.createQuery("SELECT a FROM Author a WHERE a.name =
:name", Author.class);
        query.setParameter("name", "J.K. Rowling");
        Author author = query.getSingleResult();

        if (author != null) {
```

```
            author.setName("Joanne Rowling");
            em.merge(author);
        }

        transaction.commit();
        System.out.println("Author updated successfully.");
    }

    private static void deleteAuthor(EntityManager em) {
        EntityTransaction transaction = em.getTransaction();
        transaction.begin();

        TypedQuery<Author> query = em.createQuery("SELECT a FROM Author a WHERE a.name =
:name", Author.class);
        query.setParameter("name", "Joanne Rowling");
        Author author = query.getSingleResult();

        if (author != null) {
            em.remove(author);
        }

        transaction.commit();
        System.out.println("Author deleted successfully.");
    }
}
```