Entity Mapping and Persistence in JPA

1. Introduction

Overview of ORM (Object-Relational Mapping)

Object-Relational Mapping (ORM) is a programming technique used to convert data between incompatible systems—in particular, between object-oriented programming languages and relational databases.

Popular ORM frameworks include:

- **Hibernate:** A widely used ORM framework for Java that implements the JPA (Java Persistence API) specification.
- **JPA (Java Persistence API):** A Java specification for accessing, persisting, and managing data between Java objects and a relational database.

Importance of Entity Mapping

Entity mapping is the process of defining how an application's domain objects (entities) map to database tables. With ORM, each entity typically corresponds to a table in the database, and fields in the entity correspond to columns in the table. ORM simplifies database interactions, allowing developers to focus on the object model without worrying about the underlying database structure.

Benefits of Using ORM:

- **Reduced Boilerplate Code:** Developers write less SQL code and focus more on business logic.
- **Database Portability:** ORM frameworks abstract database specifics, making it easier to switch databases.
- **Seamless Object-to-Table Mapping:** Simplifies data management by allowing developers to work directly with objects instead of SQL queries.

2. Understanding Entities

What is an Entity?

An entity is a lightweight, persistent domain object in an ORM framework that typically maps to a single table in the database. Each entity represents a row in the table, and the entity's fields correspond to the table's columns.

Annotations in JPA (Example: @Entity)

In JPA, the `@Entity` annotation marks a class as an entity that can be mapped to a database table.

Example:

```
@Entity
public class Employee {
    @Id
    private String emNo;
    private String firstName;
}
```

In this example, the `Employee` class is an entity that will be mapped to a table named `Employee` in the database.

3. Primary Keys and Annotations

Defining Primary Keys with @Id

The `@Id` annotation is used to specify the primary key of an entity. The primary key is a unique identifier for each entity instance.

Example:

```
@Id
private String emNo;
```

Here, `emNo` is the primary key for the `Employee` entity.

Generating Primary Keys

Primary keys can be generated automatically using the `@GeneratedValue` annotation, which defines strategies for key generation.

Example:

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```

In this example, `id` is automatically generated using the `AUTO` strategy, which is typically database-dependent.

4. Field Mapping

Basic Field Mapping

Fields in an entity class map directly to columns in the corresponding database table.

Example:

```
private String firstName;
```

Here, the `firstName` field in the `Employee` entity maps to a column named `firstName` in the `Employee` table.

Column Annotations (@Column)

The `@Column` annotation allows customization of the column to which a field maps. You can specify properties like column name, whether the column is nullable, the column's length, etc.

Example:

```
@Column(name = "FIRST_NAME", nullable = false, length = 50)
private String firstName;
```

This maps the `firstName` field to a column named `FIRST_NAME` that is non-nullable and has a maximum length of 50 characters.

5. Relationships Between Entities

One-to-One, One-to-Many, Many-to-One, and Many-to-Many

Entities often have relationships with one another, which can be mapped using specific annotations in JPA:

- **@OneToOne:** Maps a one-to-one relationship between two entities.
- **@OneToMany:** Maps a one-to-many relationship where one entity is related to multiple instances of another entity.
- **@ManyToOne:** Maps a many-to-one relationship where many instances of an entity are related to a single instance of another entity.
- **@ManyToMany:** Maps a many-to-many relationship where multiple instances of one entity are related to multiple instances of another entity.

Mapping Relationships

Relationships are mapped using annotations that define the nature of the relationship and how it is mapped to the database.

Example: One-to-Many Relationship

```
@Entity
public class Department {
    @Id
    private String depCode;

    @OneToMany(mappedBy = "department")
    private Set<Employee> employees;

    // Getters and Setters
}
```

In this example, the `Department` entity has a one-to-many relationship with the `Employee` entity, meaning each department can have multiple employees.

Example: Many-to-One Relationship

```
@Entity
public class Employee {
    @Id
    private String emNo;

    @ManyToOne
    @JoinColumn(name = "DEP_CODE")
    private Department department
```

Here, the `Employee` entity has a many-to-one relationship with the `Department` entity, meaning multiple employees can belong to one department.

6. Inheritance Mapping

Single Table Strategy (@Inheritance and @DiscriminatorColumn)

Inheritance in object-oriented programming can be mapped to database tables using different strategies. The single table strategy maps all classes in the inheritance hierarchy to a single table in the database.

Example:

java

Copy code

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "EMP_TYPE")
public class Employee {
    @Id
    private String emNo;
    // Common fields for all employees
}

@Entity
public class Doctor extends Employee {
    private String specialty;
    // Fields specific to doctors
}

@Entity
public class Nurse extends Employee {
    private String rotation;
    // Fields specific to nurses
```

```
}
```

In this example, all employees, doctors, and nurses are stored in a single table, with a `EMP_TYPE` column differentiating between them.

Joined Strategy

The joined strategy creates separate tables for each class in the inheritance hierarchy, with common fields stored in a base table.

Example:

java

Copy code

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Employee {
    @Id
    private String emNo;
    // Common fields for all employees
}

@Entity
public class Doctor extends Employee {
    private String specialty;
    // Fields specific to doctors
}

@Entity
public class Nurse extends Employee {
```

```
    private String rotation;
    // Fields specific to nurses
}
```

In this example, `Employee`, `Doctor`, and `Nurse` each have their own tables, with `Doctor` and `Nurse` tables containing only the fields specific to those entities.

7. Persistence Context and Entity Lifecycle

Entity States (New, Managed, Detached, Removed)

Entities can exist in different states within a persistence context, which represents the environment in which entities are managed:

- **New:** An entity that has been created but not yet persisted to the database.
- **Managed:** An entity that is currently being managed by the persistence context (e.g., after being persisted).
- **Detached:** An entity that was once managed by the persistence context but is no longer managed.
- **Removed:** An entity that has been marked for deletion but not yet removed from the database.

Persistence Context

The persistence context is a cache where entities are managed during a transaction. The `EntityManager` interface is responsible for managing the persistence context, ensuring that entities are in sync with the database.

Example:

java

Copy code

```
EntityManager em = entityManagerFactory.createEntityManager();
em.getTransaction().begin();

Employee employee = em.find(Employee.class, "E001"); // Managed state
employee.setFirstName("John");
em.getTransaction().commit();
em.close(); // Employee is now in the detached state
```