Entity Mapping and Persistence in JPA

1. Introduction to JPA

The Java Persistence API (JPA) is a specification in Java that allows developers to map Java objects to relational database tables. This process is known as Object-Relational Mapping (ORM). JPA provides a powerful and flexible way to manage relational data in Java applications using annotations and XML-based configuration.

2. Entity Mapping

Entity Mapping refers to the process of defining how Java objects (entities) correspond to database tables. In JPA, an entity is a lightweight, persistent domain object typically mapped to a single database table.

2.1. Basic Entity Mapping

- Entity Class: A class annotated with @Entity is recognized as an entity class in JPA. This class will be mapped to a table in the database.

```
@Entity
public class Employee {
    @Id
    private int id;
    private String firstName;
    private String lastName;
}
```

In this example:

- The Employee class is mapped to a table named Employee (by default).

- The @Id annotation indicates the primary key of the table.

- Table Mapping: By default, the entity is mapped to a table with the same name as the class. However, you can specify a different table name using the @Table annotation.

```
@Entity
@Table(name = "employees")
public class Employee {
    @Id
    private int id;
    private String firstName;
    private String lastName;
}
```

2.2. Mapping Entity Fields to Columns

- Basic Fields: Each field in an entity is mapped to a column in the table by default. The field name is used as the column name unless specified otherwise using the @Column annotation.

```
@Column(name = "first_name")
private String firstName;
```

- Primary Keys: The @Id annotation marks the primary key. You can also use the @GeneratedValue annotation to indicate that the value of this field should be automatically generated by the database.

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
private int id;
```

- Composite Keys: When a table has a composite key (more than one column), you can use @EmbeddedId or @IdClass to map the composite key.

```
@EmbeddedId
private EmployeeId id;
```

2.3. Mapping Relationships

JPA allows the mapping of relationships between entities to represent associations between tables.

- One-to-One Relationship: This maps a one-to-one relationship between two entities using the @OneToOne annotation.

```
@OneToOne
@JoinColumn(name = "address_id")
private Address address;
```

- One-to-Many and Many-to-One Relationships: These annotations define relationships where one entity is related to many others (@OneToMany) and where many entities are related to one (@ManyToOne).

```
@OneToMany(mappedBy = "employee")
private List<Address> addresses;

@ManyToOne
@JoinColumn(name = "department_id")
```

```
private Department department;
```

- Many-to-Many Relationship: This maps a many-to-many relationship between two entities. The relationship is usually represented by a join table.

```
@ManyToMany
@JoinTable(
    name = "employee_project",
    joinColumns = @JoinColumn(name = "employee_id"),
    inverseJoinColumns = @JoinColumn(name = "project_id")
)
private List<Project> projects;
```

2.4. Inheritance Mapping

JPA supports inheritance mapping strategies to map inheritance hierarchies in Java to database tables.

- Single Table: All classes in the hierarchy are mapped to a single table.

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public class Employee {
    // common fields
}

@Entity
public class Manager extends Employee {
```

```
    // manager-specific fields
}
```

- Joined: Each class in the hierarchy is mapped to its own table, and they are joined using primary and foreign keys.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Employee {
    // common fields
}

@Entity
public class Manager extends Employee {
    // manager-specific fields
}
```

- Table per Class: Each class is mapped to its own table without joining.

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Employee {
    // common fields
}

@Entity
public class Manager extends Employee {
```

```
    // manager-specific fields
}
```

3. Persistence

Persistence in JPA refers to the lifecycle of an entity. The main component that manages this lifecycle is the EntityManager, which provides the necessary methods to interact with the persistence context.

3.1. Persistence Context

The persistence context is a set of managed entity instances that correspond to records in the database. It is managed by the EntityManager.

- Managed State: Entities in the persistence context are in the managed state. Any changes to these entities are tracked and synchronized with the database.

- Detached State: An entity that is no longer associated with a persistence context is in a detached state.

- Transient State: An entity that is newly created and not yet associated with the persistence context is in a transient state.

- Removed State: An entity that is scheduled for deletion is in the removed state.

3.2. EntityManager

The EntityManager API is used to interact with the persistence context. It provides methods to perform CRUD (Create, Read, Update, Delete) operations.

- Persisting Entities: The persist method is used to add an entity to the persistence context, which will then be inserted into the database.

```
Employee employee = new Employee();
entityManager.persist(employee);
```

- Finding Entities: The find method retrieves an entity by its primary key.

```
Employee employee = entityManager.find(Employee.class, 1);
```

- Merging Entities: The merge method is used to update an existing entity or save a detached entity.

```
Employee updatedEmployee = entityManager.merge(employee);
```

- Removing Entities: The remove method deletes an entity from the database.

```
entityManager.remove(employee);
```

3.3. Transactions

JPA requires all operations that modify the database to be executed within a transaction. The EntityTransaction API is used to manage transactions.

- Begin Transaction: Start a new transaction.

```
entityTransaction.begin();
```

- Commit Transaction: Commit the current transaction, persisting changes to the database.

```
entityTransaction.commit();
```

- Rollback Transaction: Roll back the current transaction, undoing any changes.

```
entityTransaction.rollback();
```

3.4. Querying with JPQL

JPA provides the Java Persistence Query Language (JPQL) for querying entities in the database. JPQL is similar to SQL but operates on the entity objects rather than directly on the database tables.

- Simple Query:

```
List<Employee> employees = entityManager.createQuery("SELECT e FROM Employee e", Employee.class).getResultList();
```

- Named Queries: You can define queries using the @NamedQuery annotation, which can be reused throughout the application.

```
@NamedQuery(name = "Employee.findByName", query = "SELECT e FROM Employee e WHERE e.name = :name")
```

- Criteria API: JPA also provides a type-safe way to build queries dynamically using the Criteria API.

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Employee> cq = cb.createQuery(Employee.class);
Root<Employee> employee = cq.from(Employee.class);
```

```
cq.select(employee).where(cb.equal(employee.get("name"), "John"));
List<Employee> employees = entityManager.createQuery(cq).getResultList();
```

4. Summary

Entity mapping and persistence in JPA are crucial for building robust, database-driven Java applications. By mapping Java objects to relational database tables and managing their lifecycle, developers can focus on the business logic while JPA handles the underlying database interactions. The combination of JPA's powerful annotations, EntityManager, and query capabilities allows for flexible and efficient management of persistent data.