Entity Mapping and Persistence with JPA

Sandra Kumi

REVIEWER Mr. Thomas

Entity Mapping and Persistence with JPA

1. Introduction

Data persistence is a fundamental aspect of application development, especially in enterprise applications where data needs to be stored and retrieved efficiently. Java Persistence API (JPA) is a standard specification that provides a powerful and flexible way to map Java objects to relational database tables. This document serves as a comprehensive guide to understanding entity mapping and persistence using JPA, focusing on essential concepts, annotations, and best practices.

---

2. Understanding JPA and ORM

What is JPA?

The Java Persistence API (JPA) is a specification for managing relational data in Java applications. It defines a set of standards and guidelines for mapping Java objects to database tables, allowing developers to focus on their domain model rather than on database-specific code.

What is ORM?

Object-Relational Mapping (ORM) is a technique that allows developers to convert data between incompatible systems (like a relational database and an object-oriented programming language). ORM frameworks, such as Hibernate (which implements JPA), automate the mapping between Java objects and database tables, enabling developers to work with high-level object-oriented constructs rather than writing complex SQL queries.

Why Use JPA?

JPA provides several advantages, including:

- **Portability: JPA is a standard specification, meaning your code is portable across different ORM providers.**
- **Ease of Use: JPA simplifies data persistence by abstracting complex database interactions.**
- **Integration: JPA integrates seamlessly with other Java EE technologies, providing a unified development experience.**

---

3. Entity Basics

Defining an Entity

An entity in JPA represents a table in a relational database. It is a lightweight, persistent domain object that is managed by the JPA entity manager. Entities typically represent business objects like Customer, Product, or Order.

The @Entity Annotation

To define a class as an entity, annotate it with @Entity. This tells JPA to map the class to a corresponding database table.

```
import jakarta.persistence.Entity;

import jakarta.persistence.Table;


@Entity

@Table(name = "customers")

public class Customer {

    // fields, getters, setters

}
```

In this example, the Customer class is mapped to the customers table in the database.

---

4. Primary Keys and Identity

The @Id Annotation

Every entity must have a primary key, which uniquely identifies each record in the database. The @Id annotation is used to specify the primary key field in the entity.

```
import jakarta.persistence.Id;


@Entity

public class Customer {

    @Id

    private Long id;

    // other fields, getters, setters

}
```

Generation Strategies

JPA provides several strategies for generating primary key values:

- **AUTO: JPA provider chooses the generation strategy.**
- **IDENTITY: The database generates the primary key.**
- **SEQUENCE: JPA uses a database sequence.**
- **TABLE: A table is used to generate unique identifiers.**

```
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;

@Entity
public class Customer {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   // other fields, getters, setters
}
```

---

5. Field Mapping

Basic Field Mapping

Fields in an entity are automatically mapped to columns in the corresponding database table. You can customize the mapping using the @Column annotation.

```
import jakarta.persistence.Column;

@Entity
public class Customer {
   @Id
```

```
    private Long id;

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

    @Column(name = "last_name")
    private String lastName;
    // getters, setters
}
```

Transient Fields

Sometimes, you might have fields in your entity that you don't want to persist. You can use the @Transient annotation to mark these fields.

```
import jakarta.persistence.Transient;

@Entity
public class Customer {
    @Id
    private Long id;

    private String firstName;
    private String lastName;

    @Transient
    private int age; // not persisted
    // getters, setters
}
```

Embeddable Types

You can group related fields into a value type using @Embeddable and @Embedded annotations. This allows you to reuse the group across multiple entities.

```
import jakarta.persistence.Embeddable;
import jakarta.persistence.Embedded;

@Embeddable
public class Address {
    private String street;
    private String city;
    private String zipCode;
    // getters, setters
}

@Entity
public class Customer {
    @Id
    private Long id;

    @Embedded
    private Address address;
    // getters, setters
}
```

---

6. Entity Relationships

One-to-One Relationships

A one-to-one relationship is mapped using the @OneToOne annotation. This means that one entity is associated with exactly one instance of another entity.

```
import jakarta.persistence.OneToOne;

@Entity
public class Customer {
    @Id
    private Long id;

    @OneToOne
    private Address address;
    // getters, setters
}
```

One-to-Many Relationships

A one-to-many relationship is where one entity is associated with multiple instances of another entity. This is mapped using the @OneToMany annotation.

```
import jakarta.persistence.OneToMany;

@Entity
public class Customer {
    @Id
    private Long id;

    @OneToMany(mappedBy = "customer")
    private List<Order> orders;
    // getters, setters
}
```

Many-to-One Relationships

A many-to-one relationship is the opposite of one-to-many, where multiple entities are associated with a single instance of another entity. This is mapped using the @ManyToOne annotation.

```
import jakarta.persistence.ManyToOne;


@Entity
public class Order {
   @Id
   private Long id;


   @ManyToOne
   private Customer customer;
   // getters, setters
}
```

Many-to-Many Relationships

A many-to-many relationship is where multiple instances of one entity are associated with multiple instances of another entity. This is mapped using the @ManyToMany annotation.

```
import jakarta.persistence.ManyToMany;


@Entity
public class Course {
   @Id
   private Long id;


   @ManyToMany
   private List<Student> students;
```

```
    // getters, setters
}
```

---

7. Inheritance Mapping

Single Table Inheritance

In single table inheritance, all classes in the hierarchy are mapped to a single table. This is the default strategy and is configured using the @Inheritance annotation.

```
import jakarta.persistence.Inheritance;
import jakarta.persistence.InheritanceType;

@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class Person {
    @Id
    private Long id;
    // other fields, getters, setters
}

@Entity
public class Employee extends Person {
    private String department;
    // getters, setters
}

@Entity
public class Customer extends Person {
    private String membershipLevel;
```

```
    // getters, setters
}
```

Table per Class

In the table per class strategy, each class in the hierarchy has its own table. This can be specified using InheritanceType.TABLE_PER_CLASS.

```
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Person {
    @Id
    private Long id;
    // other fields, getters, setters
}
```

Joined Table Strategy

In the joined table strategy, each class in the hierarchy has its own table, but they are joined through foreign keys. This is specified using InheritanceType.JOINED.

```
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Person {
    @Id
    private Long id;
    // other fields, getters, setters
}
```

---

8. Entity Lifecycle

Entity States

An entity can be in one of the following states:

- **Transient: The entity is new and has not been persisted yet.**
- **Persistent: The entity is managed by the EntityManager.**
- **Detached: The entity was persisted but is no longer managed by the EntityManager.**

- **Removed: The entity has been marked for deletion.**

Lifecycle Callbacks

JPA provides lifecycle callback annotations that allow you to intercept specific lifecycle events of an entity.

- **@PrePersist: Executed before the entity is persisted.**
- **@PostPersist: Executed after the entity is persisted.**
- **@PreRemove: Executed before the entity is removed.**
- **@PostRemove: Executed after the entity is removed.**
- **@PreUpdate: Executed before the entity is updated.**
- **@PostUpdate: Executed after the entity is updated.**

```
import jakarta.persistence.PrePersist;
import jakarta.persistence.PostPersist;

@Entity
public class Customer {
   @Id
   private Long id;

   @PrePersist
   public void prePersist() {
      System.out.println("Before persisting the entity");
   }

   @PostPersist
   public void postPersist() {
      System.out.println("After persisting the entity");
   }
}
```

9. EntityManager and Persistence Context

What is the EntityManager?

The EntityManager is the primary interface used to interact with the persistence context. It manages the lifecycle of entities and provides an API for performing CRUD operations.

Persistence Context

The persistence context is a set of managed entity instances that correspond to rows in a database. It ensures that for any particular database row, there is only one managed entity instance within a particular persistence context.

CRUD Operations with EntityManager

The EntityManager provides methods for creating, reading, updating, and deleting entities.

- **Persisting an Entity:**

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

Customer customer = new Customer();
customer.setFirstName("John");
customer.setLastName("Doe");

em.persist(customer);
em.getTransaction().commit();
```

- **Finding an Entity:**

```
Customer customer = em.find(Customer.class, 1L);
```

- **Updating an Entity:**

```
Customer customer = em.find(Customer.class, 1L);
```

customer.setLastName("Smith");

em.merge(customer);

- **Deleting an Entity:**

Customer customer = em.find(Customer.class, 1L);

em.remove(customer);

---

10. Querying Entities

JPQL (Java Persistence Query Language)

JPQL is a query language that operates on entity objects rather than directly on database tables. It is similar to SQL but is tailored for JPA entities.

java

Copy code

TypedQuery<Customer> query = em.createQuery("SELECT c FROM Customer c WHERE c.lastName = :lastName", Customer.class);

query.setParameter("lastName", "Doe");

List<Customer> results = query.getResultList();

Named Queries

Named queries are predefined queries that are defined using the @NamedQuery annotation on the entity class.

@NamedQuery(name = "Customer.findByLastName", query = "SELECT c FROM Customer c WHERE c.lastName = :lastName")

@Entity

public class Customer {

@Id

private Long id;

// other fields, getters, setters

}

Native SQL Queries

JPA also allows you to execute native SQL queries using the createNativeQuery method.

java

Copy code

```
Query query = em.createNativeQuery("SELECT * FROM customers WHERE last_name = ?", Customer.class);
query.setParameter(1, "Doe");
List<Customer> results = query.getResultList();
```

---

11. Optimistic and Pessimistic Locking

Versioning with @Version

Optimistic locking is used to prevent lost updates in concurrent transactions. It is implemented using the @Version annotation.

```
import jakarta.persistence.Version;

@Entity
public class Customer {
    @Id
    private Long id;

    @Version
    private int version;
    // other fields, getters, setters
}
```

Locking Strategies

- **Optimistic Locking: Relies on versioning to ensure data integrity. It allows multiple transactions to complete as long as they don't conflict.**

- **Pessimistic Locking: Locks the database row immediately, preventing other transactions from updating it until the lock is released.**

em.lock(customer, LockModeType.PESSIMISTIC_WRITE);

---

12. Conclusion

Entity mapping and persistence are critical components in building robust and scalable Java applications. JPA provides a standardized approach to managing relational data, enabling developers to focus on the business logic rather than database intricacies. By understanding and utilizing JPA's rich feature set, including entity relationships, inheritance, lifecycle management, and querying, developers can create efficient and maintainable applications.

---

13. References

- **Books:**
  - **"Pro JPA 2 in Java EE 8: An In-Depth Guide to Java Persistence APIs" by Mike Keith and Merrick Schincariol**
  - **"Java Persistence with Hibernate" by Christian Bauer and Gavin King**
- **Online Resources:**
  - **Java Persistence API (JPA) Documentation**
  - **JPA Tutorial by Baeldung**
  - **Vlad Mihalcea's Blog on Hibernate and JPA**