Class-24: JPA Relationships

by Pial Kanti Samadder

What Are Entity Relationships in JPA?

In real-world applications, entities often depend on each other. For example, a user might create multiple posts, or an order might contain multiple products. JPA makes it easy to model such relationships using annotations.

Key types of relationships:

- **One-to-One:** A single entity is related to another single entity (e.g., a user and their profile).
- **One-to-Many / Many-to-One**: One entity relates to many others (e.g., a user and their posts).
- **Many-to-Many**: Multiple entities relate to multiple others (e.g., students and courses).Let's see each type in action with examples.

Entity Relationships Using Spring Data JPA

The four annotations used to represent entity relationships are:

1. `@OneToOne`
2. `@ManyToOne`
3. `@OneToMany`
4. `@ManyToMany`

Unidirectional vs. Bidirectional Relationships

Relationship Direction

Type	Meaning	Example
Unidirectional	Only **one entity knows** about the other.	`Customer` → Order
Bidirectional	**Both entities** know about each other.	`Customer` ←→ `Order`

“Aware” = The entity has a field referencing the other entity.

The Owner of the Relationship

In JPA entity relationships, there is always an "owner" of the relationship. Figuring out the owner of the relationship is different for unidirectional and bidirectional relationships.

In unidirectional relationships, the owning side is the side that is aware. Whereas **in bidirectional relationships, the entity that does not have a** `mappedBy` **annotation parameter is the owning side**.

Knowing which side owns the relationship is essential for two reasons:

1. Saving data through the child or owned entity can be hugely problematic.
2. The location of the join column in the database depends on which side owns the relationship.

More JPA Entity Relationship Options

1. `fetch`
   - This parameter dictates which strategy the JPA provider will use to fetch different instances from the database.
   - Two Loading options:
     - `FetchType.EAGER` : Loads related entities **immediately** with the parent.
       Example: Fetching a `Customer` also loads all its `Orders`.
     - `FetchType.LAZY` *(default for collections)* : Loads related entities **only when accessed**.
       Example: `customer.getOrders()` triggers a query.
2. `optional`
   - A **boolean that determines whether the relationship *must* include a related entity or can be null.** Similar to `@Column(nullable = false)` for a column's constraints.
   - Options:
     - `optional = true` *(default)* : Relationship **can be null**. (Child not required)
     - `optional = false` : Relationship **must exist**.
       Like `@Column(nullable = false)` — prevents saving parent without child. Attempting to save a parent information entry without a child will result in an exception.
3. `cascade`
   - Changes the way that various database changes (DELETE, UPDATE, etc.) propagate through relationships.
   - In other words, if two tables are related, and you delete the parent entity, does that delete command also propagate to the child entity? This depends on the cascade parameter.
   - The most common options are:
     - **CascadeType.ALL**: All operations will cascade to the child entity.
     - **CascadeType.PERSIST**: When persisting a parent entity, previously unpersisted child entities will be persisted as well.
     - `CascadeType.MERGE`: Updates child entities when parent is merged.
     - **CascadeType.REMOVE**: Deletion operations on the parent entity will also remove related child entities.

`@OneToOne` Relationship

A `@OneToOne` relationship is used when one entity is directly associated with exactly one other entity. This can be a straightforward **parent-child** relationship where each side has one associated entity. This is often used for **strongly related entities**.

Key Attributes:

- `mappedBy`: For a bidirectional `@OneToOne` relationship, the `mappedBy` attribute is used on the **inverse side** (the side that does not own the relationship).
- `cascade`: Defines how persistence operations should propagate. If set to `CascadeType.ALL`, changes to the parent entity will be applied to the related child entity.
- `fetch`: Defines the fetch strategy. For `@OneToOne`, **eager fetching** is the default (the associated entity is loaded immediately with the parent entity).

Example of `@OneToOne` Relationship

```
@Entity
public class Person {
    @Id
    private int id;

    @OneToOne(mappedBy = "person", cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    private Passport passport; // Each person has one passport

    // Getters and setters
}
```

```
@Entity
public class Passport {
    @Id
    private int id;

    @OneToOne
    @JoinColumn(name = "person_id") // Foreign key to Person
    private Person person; // Each passport belongs to one person

    // Getters and setters
}
```

Explanation:

- `mappedBy = "person"`: In this bidirectional `@OneToOne` relationship, the `Person` entity is the inverse side, and the `Passport` entity owns the relationship.
- `@JoinColumn(name = "person_id")`: The foreign key (`person_id`) linking the `Passport` to the `Person` is placed in the `Passport` entity.
- `cascade = CascadeType.ALL`: This means persistence operations will propagate from the `Person` entity to the `Passport` entity.

`@OneToMany` Relationship

A `@OneToMany` relationship represents a scenario where one entity can have multiple associated entities. This is typically a **parent-child** relationship where one parent entity (the **"one"** side) can have many child entities (the **"many"** side).

Key Attributes:

- `mappedBy`: Specifies the field in the child entity that owns the relationship. The `mappedBy` attribute is placed on the **inverse side** of the relationship, meaning on the **parent** entity. The foreign key linking the parent and child is stored in the **child entity**.
- `cascade`: Defines the persistence operations (such as `persist`, `remove`, etc.) that should propagate from the parent entity to the child entities.
- `fetch`: Defines the fetch strategy for the associated entities. The default fetch type for `@OneToMany` is **lazy loading**, meaning that the collection is loaded only when accessed.

Example of `@OneToMany` Relationship

```
@Entity
public class Person {
    @Id
    private int id;

    @OneToMany(mappedBy = "person", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    private List<Address> addresses; // One person can have many addresses

    // Getters and setters
}
```

```
@Entity
public class Address {
    @Id
    private int id;

    @ManyToOne
    @JoinColumn(name = "person_id") // Foreign key to Person
    private Person person; // Each address belongs to one person

    // Getters and setters
}
```

Explanation:

- `mappedBy = "person"` tells JPA that the `Person` entity is the inverse side of the relationship. The `Address` entity owns the relationship, which means the foreign key (`person_id`) linking the `Address` to the `Person` will be in the `Address` entity.
- `cascade = CascadeType.ALL` means that persistence operations on the `Person` entity (such as saving or deleting) will automatically propagate to the associated `Address` entities.
- `fetch = FetchType.LAZY` ensures that the `addresses` collection is fetched lazily (only when accessed), which is generally more efficient for large collections.

`@ManyToMany` Relationship

A `@ManyToMany` relationship is used when two entities can have multiple relationships with each other. In this case, both sides of the relationship are "many". This is common when entities are connected to one another, like a **student-course** relationship, where a student can enroll in multiple courses, and a course can have multiple students.

Key Attributes:

- `mappedBy`: The **inverse side** of the relationship uses `mappedBy` to specify the field that owns the relationship. It indicates that the relationship is managed from the other side (the owning side). The foreign keys for the relationship are stored in a **join table** rather than directly in the `Student` or `Course` tables.
- `cascade`: Defines the persistence operations (like `persist`, `remove`, etc.) that should propagate from the owning side to the inverse side.
- `fetch`: Defines the fetch strategy for the associated entities. By default, **lazy loading** is used in a many-to-many relationship.

Example of `@ManyToMany` Relationship

```
@Entity
public class Student {
    @Id
    private int id;

    @ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @JoinTable(
        name = "student_course", // Name of the join table
        joinColumns = @JoinColumn(name = "student_id"), // Foreign key referencing Student
        inverseJoinColumns = @JoinColumn(name = "course_id") // Foreign key referencing Course
    )
    private List<Course> courses;

    // Getters and setters
}
```

```
@Entity
public class Course {
    @Id
    private int id;

    @ManyToMany(mappedBy = "courses", fetch = FetchType.LAZY) // Inverse side
    private List<Student> students;

    // Getters and setters
}
```

Explanation:

- `@JoinTable(name = "student_course")`: Hibernate creates a join table named `student_course` to hold the relationship. This table will contain two foreign keys:
  - `student_id`: a foreign key referencing the `Student` entity.
  - `course_id`: a foreign key referencing the `Course` entity.
- `mappedBy = "courses"`: On the `Course` entity, `mappedBy` tells JPA that the relationship is managed from the `Student` side. It means the `Course` entity does not directly manage the foreign key but maps it back to the `Student` entity.
- **Foreign Keys**: The foreign keys are not stored in the `Student` or `Course` tables themselves, but in the **join table** (`student_course`).

JPA Entity Lifecycle Events

When working with JPA, there are several events that we can be notified of during an entity's lifecycle.

JPA specifies seven optional lifecycle events that are called:

- before persist is called for a new entity – *@PrePersist*
- after persist is called for a new entity – *@PostPersist*
- before an entity is removed – *@PreRemove*
- after an entity has been deleted – *@PostRemove*
- before the update operation – *@PreUpdate*
- after an entity is updated – *@PostUpdate*
- after an entity has been loaded – *@PostLoad*

There are two approaches for using the lifecycle event annotations:

1. annotating methods in the entity.
2. creating an *EntityListener* with annotated callback methods.

Annotating the *Entity*

Let’s start by using the callback annotations directly in our entity. In our example, we’re going to leave a log trail when `User` records are changed, so we’re going to add simple logging statements in our callback methods.

```
@Entity
public class User {
    private static Log log = LogFactory.getLog(User.class);

    @Id
    @GeneratedValue
    private int id;

    private String userName;
    private String firstName;
    private String lastName;

    // Standard getters/setter

    @PrePersist
    public void logNewUserAttempt() {
        log.info("Attempting to add new user with username: " + userName);
    }

    @PostPersist
    public void logNewUserAdded() {
        log.info("Added user '" + userName + "' with ID: " + id);
    }

    @PreRemove
    public void logUserRemovalAttempt() {
        log.info("Attempting to delete user: " + userName);
    }

    @PostRemove
    public void logUserRemoval() {
        log.info("Deleted user: " + userName);
    }

    @PreUpdate
    public void logUserUpdateAttempt() {
        log.info("Attempting to update user: " + userName);
    }

    @PostUpdate
    public void logUserUpdate() {
        log.info("Updated user: " + userName);
    }

    @PostLoad
    public void logUserLoad() {
        fullName = firstName + " " + lastName;
    }
}
```

Annotating an `EntityListener`

We're going to expand on our example now and use a separate *EntityListener* to handle our update callbacks.
We might favor this approach over placing the methods in our entity if we have some operation we want to apply to all of our entities.

Let's create our `UserEntityListener` to log all the activity on the *User* table:

```
public class UserEntityListener {
    private static Log log = LogFactory.getLog(UserEntityListener.class);

    @PrePersist
    @PreUpdate
    @PreRemove
    private void beforeAnyUpdate(User user) {
        if (user.getId() == 0) {
            log.info("[USER AUDIT] About to add a user");
        } else {
            log.info("[USER AUDIT] About to update/delete user: " + user.getId());
        }
    }

    @PostPersist
    @PostUpdate
    @PostRemove
    private void afterAnyUpdate(User user) {
        log.info("[USER AUDIT] add/update/delete complete for user: " + user.getId());
    }

    @PostLoad
    private void afterLoad(User user) {
        log.info("[USER AUDIT] user loaded from database: " + user.getId());
    }
}
```

Now, we need to go back to our *User* entity and add the *@EntityListener* annotation to the class:

```
@EntityListeners(UserEntityListener.class)
@Entity
public class User {
    //...
}
```

Running Sample Project

- **Clone the Repository**

If you haven't cloned the repository yet, run the following command (ensure `git` is installed):

```
git clone https://github.com/PialKanti/Ostad-SpringBoot-Course.git
```

Then switch to the correct branch for today's class (replace with the actual branch name, e.g., `class-24-jpa-relationship`):

```
git fetch
git switch class-24-jpa-relationship
```

Or,

If You Already Have the Repository Cloned, simply open your existing project folder and switch (or checkout) to the appropriate branch:

```
git fetch
git switch class-24-jpa-relationship
```

- **Set Up and Run PostgreSQL Database**

You can run PostgreSQL **either via Docker** or a **desktop installation**.

Option 1: Run via Docker

A `compose.yml` file is available in the root of the repository.
Run the following command from the project root:

```
docker compose up -d
```

This will start a PostgreSQL container automatically.

Or,

Option 2: Run via PostgreSQL Desktop (Manual Setup)

If you already have PostgreSQL installed locally:

1. Start your PostgreSQL server.
2. Create a new database named `crud_db` if not exists.

- **Open the Project in IntelliJ IDEA**

1. Open IntelliJ IDEA.
2. Click **File → Open** and select the `crud-sample` folder inside the repository.
3. Let IntelliJ import Maven/Gradle dependencies automatically.