**Working with multiple entities and relationship with jpa**

**🔄 What is Cascade in JPA (Java Persistence API)?**

**Cascade** is a way to automatically apply certain operations (like persist, remove, merge, etc.) on **related entities** when an operation is performed on the **parent entity**.

**🧠 Why Use Cascade?**

Imagine two entities:

* SocialUser (parent)
* Post (child)

If a SocialUser has many Posts, and you **delete a user**, you might also want to **automatically delete** all their posts — this is where **cascade** comes in.

**✅ Example: One-to-Many with Cascade**

java

CopyEdit

@Entity

public class SocialUser {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String username;

@OneToMany(mappedBy = "user", cascade = CascadeType.ALL)

private List<Post> posts = new ArrayList<>();

}

In this example:

* @OneToMany defines the relationship.
* cascade = CascadeType.ALL means:

When you **save**, **update**, or **delete** a SocialUser, those operations will also be applied to the Posts.

**🧾 Common CascadeType Options:**

| **CascadeType** | **Meaning** |
| --- | --- |
| ALL | Applies all cascade operations (persist, merge, remove, etc.) |
| PERSIST | When parent is saved, child is also saved |
| MERGE | When parent is updated, child is updated too |
| REMOVE | When parent is deleted, child is deleted |
| DETACH | Detaches child when parent is detached |
| REFRESH | Refreshes child entities when parent is refreshed |

**🛑 Without Cascade**

If you don’t use cascade, you’ll have to manually save or delete child entities. For example:

java

CopyEdit

entityManager.persist(user);

entityManager.persist(post1);

entityManager.persist(post2);

With cascade, this becomes:

java

CopyEdit

entityManager.persist(user); // posts will also be saved automatically

Great question! In JPA (Java Persistence API), **fetch** type (like LAZY or EAGER) defines **when** related data (like a list of posts for a user) is loaded from the database.

**🔍 fetch in JPA Relationships**

Used with annotations like @OneToMany, @ManyToOne, etc.

**🔹 FetchType.LAZY (Default for most collections)**

👉 Loads the related entity **only when you access it** (on-demand / delayed).

**Example:**

java

CopyEdit

@OneToMany(mappedBy = "user", fetch = FetchType.LAZY)

private List<Post> posts;

* When you fetch a SocialUser, **posts are not loaded immediately.**
* If you do user.getPosts(), then JPA will **fire a query** to load them.

✅ Efficient if you **don’t always need** the related data.

**🔸 FetchType.EAGER**

👉 Loads the related entity **immediately** along with the parent entity.

**Example:**

java

CopyEdit

@OneToMany(mappedBy = "user", fetch = FetchType.EAGER)

private List<Post> posts;

* When you fetch a SocialUser, it **immediately loads all posts**.
* One query or a join is fired to get both user and posts.

⚠️ Not memory efficient if the child list is **large** or **rarely used**.

**📊 Comparison Table**

| **Feature** | **LAZY** | **EAGER** |
| --- | --- | --- |
| Loading | On access (when needed) | Immediately |
| Performance | Better (less data fetched) | Slower if data is large |
| Default for List | Yes | No |
| Use When | Child data is optional | Child data is always needed |

**🔧 Real Example:**

java

CopyEdit

@Entity

public class SocialUser {

@Id

private Long id;

@OneToMany(mappedBy = "user", fetch = FetchType.LAZY)

private List<Post> posts;

}

Now:

java

CopyEdit

SocialUser user = userRepo.findById(1L).get();

// posts not loaded yet

List<Post> posts = user.getPosts();

// Now JPA loads posts from DB (lazy loading)

**🧠 When to Use What?**

* Use **LAZY**: When you want better performance and don’t always need the related data.
* Use **EAGER**: Only if you're sure the data is **always required** (e.g., user profile with roles).