Mapeo de relaciones en Hibernate / JPA

1. Introducción

En el tema de CRUD trabajábamos con entidades independientes (Persona, Producto, etc.).
En una aplicación real, las entidades **se relacionan**:

- Un cliente tiene **muchos** pedidos.
- Un grupo tiene **un único** profesor tutor.
- Un alumno está en **muchas** asignaturas, y una asignatura tiene **muchos** alumnos.

En una base de datos relacional las relaciones se representan con:

- **Claves foráneas (FK)** en las tablas.
- En algunos casos, **tablas intermedias**.

En Hibernate/JPA las relaciones se mapean con **anotaciones** sobre los atributos de nuestras clases:

- *@ManyToOne*
- *@OneToMany*
- *@OneToOne*
- *@ManyToMany*

2. Tipos de relaciones

2.1. ManyToOne / OneToMany (N – 1 / 1 – N)

Caso típico:

- Muchos pedidos pertenecen a un cliente.
- Un cliente tiene muchos pedidos.

En BD:

En PEDIDO hay una columna cliente_id que es FK a CLIENTE(id).

2.2. OneToOne (1 – 1)

Ejemplos:

- Cada grupo tiene un único profesor tutor.
- Cada persona tiene un único DNI.

Normalmente se resuelve con **una FK con restricción UNIQUE** en una de las dos tablas.

2.3. ManyToMany (N – N)

Ejemplos:

- Un alumno está en muchas asignaturas.
- Una asignatura tiene muchos alumnos.

En BD se resuelve con una **tabla intermedia**:

ALUMNO ----- ALUMNO_ASIGNATURA ---- ASIGNATURA

3. ManyToOne / OneToMany: ejemplo Cliente–Pedido

3.1. Modelo

- Un **Cliente** tiene muchos **Pedido**.
- Un **Pedido** pertenece a un solo **Cliente**.

Base de datos

- Tabla CLIENTE con id, nombre, email, …
- Tabla PEDIDO con id, fecha, importe_total, cliente_id (FK).

3.2. Lado "muchos → uno" (@ManyToOne)

En la entidad PEDIDO:

```
@Entity
public class Pedido {

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

    private String descripcion;

    @ManyToOne
    @JoinColumn(name = "cliente_id")  // columna FK en PEDIDO
    private Cliente cliente;

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

- *@ManyToOne* indica que muchos pedidos apuntan a un cliente.
- *@JoinColumn(name = "cliente_id") indica* el nombre de la columna FK en la tabla PEDIDO.

Este **lado es el propietario** de la relación (owning side).

3.3. Lado "uno → muchos" (@OneToMany)

En la entidad CLIENTE:

```
@Entity
public class Cliente {

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

    private String nombre;

    @OneToMany(mappedBy = "cliente")
    private List<Pedido> pedidos = new ArrayList<>();

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

- *mappedBy = "cliente"* hace referencia al nombre del atributo en Pedido que tiene la FK.
- Este lado es el lado inverso. No crea nuevas columnas, solo refleja la relación.

3.4. Mantener las dos entidades en Java

Para que la relación esté correcta en memoria y ambos lados estén sincronizados:

```
Cliente c = new Cliente();
c.setNombre("Ana");

Pedido p1 = new Pedido();
p1.setDescripcion("Portátil");
// relacionar ambos lados
p1.setCliente(c);
c.getPedidos().add(p1);
```

4. OneToOne: ejemplo Grupo–Profesor tutor

4.1. Modelo

- Un **Grupo** tiene **un solo** profesor tutor.
- Un **Profesor** puede ser tutor de **como máximo un** grupo.

Base de datos

Una opción sencilla:

- Tabla GRUPO con idGrupo, nombre, tutor_id (FK a PROFESOR).
- Tabla PROFESOR con idProfe, nombre, ...

GRUPO.tutor_id es FK y se puede marcar como UNIQUE para asegurar la 1-1.

4.2. Lado propietario

En la entidad GRUPO

```
@Entity
@Table(name = "Grupo")
public class Grupo {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "idGrupo")
    private Long idGrupo;

    private String nombre;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "tutor_id", referencedColumnName = "idProfe")
    private Profesor tutor;

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

- *@JoinColumn* crea la columna tutor_id en GRUPO.
- Este lado es el **propietario** de la relación.

4.3. Lado inverso con mappedBy

En la entidad `PROFESOR`:

```java
@Entity
@Table(name = "Profesor")
public class Profesor {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "idProfe")
    private Long idProfe;

    private String nombre;

    @OneToOne(mappedBy = "tutor")
    private Grupo grupoQueTutora;

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

Importante:

- *mappedBy = "tutor"* debe coincidir con el **nombre del atributo** en Grupo que tiene *@OneToOne* + *@JoinColumn*.
- No es el nombre de la columna (tutor_id), ni “elProfe”: es el nombre del **campo Java**.

5. ManyToMany: ejemplo Alumno–Asignatura

5.1. Modelo

- Un **Alumno** puede cursar muchas **Asignatura**.
- Una **Asignatura** puede tener muchos **Alumno**.

Base de datos

- Tabla ALUMNO (id, nombre, ...)
- Tabla ASIGNATURA (id, nombre, ...)
- Tabla intermedia ALUMNO_ASIGNATURA con:
    - alumno_id (FK a ALUMNO)

- asignatura_id (FK a ASIGNATURA)
- clave primaria compuesta (alumno_id, asignatura_id)

5.2. Lado propietario

Elegimos que el propietario sea ALUMNO.

```
@Entity
public class Alumno {

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

    private String nombre;

    @ManyToMany
    @JoinTable(
        name = "alumno_asignatura",
        joinColumns = @JoinColumn(name = "alumno_id"),
        inverseJoinColumns = @JoinColumn(name = "asignatura_id")
    )
    private List<Asignatura> asignaturas = new ArrayList<>();

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

- *@ManyToMany* indica la relación N-N.
- *@JoinTable* define la tabla intermedia y las dos columnas de unión.

Este lado es el **propietario**.

5.3. Lado inverso

```
@Entity
public class Asignatura {

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

    private String nombre;

    @ManyToMany(mappedBy = "asignaturas")
    private List<Alumno> alumnos = new ArrayList<>();

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

- *mappedBy = "asignaturas"* indica que la tabla intermedia se define en Alumno.

5.4. Uso en Java

```
Alumno a = new Alumno();
Asignatura mps = new Asignatura();

a.getAsignaturas().add(mps);
mps.getAlumnos().add(a);

session.save(a);    // gracias a cascade y lado propietario, se guarda la relación
```

6. Cascadas (cascade)

El atributo cascade permite que ciertas operaciones se propaguen automáticamente a las entidades relacionadas.

Ejemplo:

```
@OneToMany(mappedBy = "cliente", cascade = CascadeType.AL
private List<Pedido> pedidos;
```

Tipos de cascada más comunes:

- *CascadeType.PERSIST* → al guardar el padre (save), se guardan también los hijos nuevos.
- *CascadeType.MERGE* → al hacer update/merge se sincronizan los cambios.
- *CascadeType.REMOVE* → al borrar el padre, se borran también los hijos.
- *CascadeType.ALL* → aplica todas las anteriores.

Ejemplo práctico:

```
Cliente c = new Cliente();
Pedido p = new Pedido();

p.setCliente(c);
c.getPedidos().add(p);

session.save(c);  // si hay cascade PERSIST/ALL, también guarda p

// SIN CASCADA TENDRÍAMOS QUE HACER:

session.save(c);
session.save(p);
```

7. Fetch: LAZY vs EAGER

El atributo fetch indica **cuándo** se cargan las relaciones:

```
@OneToMany(mappedBy = "cliente", fetch = FetchType.LAZY)
private List<Pedido> pedidos;
```

Valores posibles:

- *FetchType.LAZY* → **carga perezosa**:
    - La colección (o relación) se carga **cuando se accede a ella por primera vez**.
    - Hibernate crea un "proxy" que dispara la consulta cuando llamamos a getPedidos(), .size(), etc.
- *FetchType.EAGER* → **carga ansiosa**:
    - La relación se carga **siempre** junto con la entidad principal.

Reglas por defecto de JPA:

- Colecciones *(@OneToMany, @ManyToMany)* → por defecto **LAZY**.
- Relaciones a una sola entidad (*@ManyToOne, @OneToOne*) → por defecto **EAGER**.

8. Errores típicos y cómo evitarlos

1. **mappedBy mal puesto**
    - Usar el nombre de la columna (tutor_id) en vez del nombre del atributo (tutor).
    - Resultado: Hibernate intenta crear columnas/tablas extra o no mapea bien la relación.
    - Solución: mappedBy **siempre** es el nombre del atributo Java del otro lado.
2. **No actualizar los dos lados de la relación**
    - Ej.: poner pedido.setCliente(c) pero no añadir pedido a c.getPedidos().
    - A veces funciona (porque manda el lado propietario), pero en memoria los datos quedan inconsistentes.
    - Buen hábito: en métodos de utilidad, hacer ambas cosas a la vez.
3. **Olvidar las cascadas necesarias**
    - Crear un cliente con pedidos, guardar solo el cliente y esperar que se guarden los pedidos.

- Solución: configurar cascade o guardar explícitamente los hijos.

4. **LazyInitializationException**
    - Acceder a colecciones LAZY fuera de la sesión(LAZY exige que la sesión siga abierta).
    - Solución: o mantener la sesión abierta, o usar JOIN FETCH y cargar antes de cerrar.
5. **ManyToMany sin pensar la tabla intermedia**
    - Cambiar la relación a @ManyToMany sin revisar cómo queda la base de datos.
    - Recordar siempre que en BD aparece una **tabla nueva** con las dos FKs.