Acceso a Datos

UT4. ASOCIACIONES ENTRE ENTIDADES

1. Introducción

- Normalmente, en todos los sistemas de información, las entidades del modelo de dominio suelen estar asociadas entre ellas, de forma que suele ser tan importante (o más) registrar la información asociada a la asociación como a la entidad.

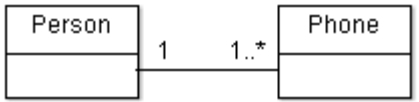
- A nivel de base de datos, estas asociaciones suelen implementarse mediante **claves externas**. Una *foreign key* no es más que la presencia de la clave primaria de una tabla en otra tabla (puede ser reflexiva, y estar en la misma tabla), a la cual apunta.

- JPA nos permite modelar estas asociaciones, tanto de forma unidireccional como de forma bidireccional. Veamos poco a poco todas ellas.

2. Asociación Many-To-One (Muchos a uno)

- Se trata de la asociación más sencilla y común de todas. Este tipo de asociaciones es conocida en algunos contextos como una relación padre/hijo, donde el lado muchos es el *hijo* y el lado uno es el *padre* **–NO CONFUNDIR CON JERARQUÍAS Y HERENCIA–.**

- La versión más sencilla de esta asociación es la **unidireccional**, en la que solo representamos la asociación en el lado muchos.



2. Asociación Many-To-One (Muchos a uno)

La anotación *@ManyToOne* nos permite indicar que una columna representa una asociación muchos-a-uno con otra entidad. El tipo de dato será el de la clase del lado uno.

Complementariamente, podemos añadir las anotaciones *@JoinColumn*, que nos permite indicar el nombre de la columna que hará las funciones de clave externa, así como *@ForeignKey*, con la que podemos indicar el nombre de la restricción que se creará a nivel de base de datos (muy útil para depurar errores).

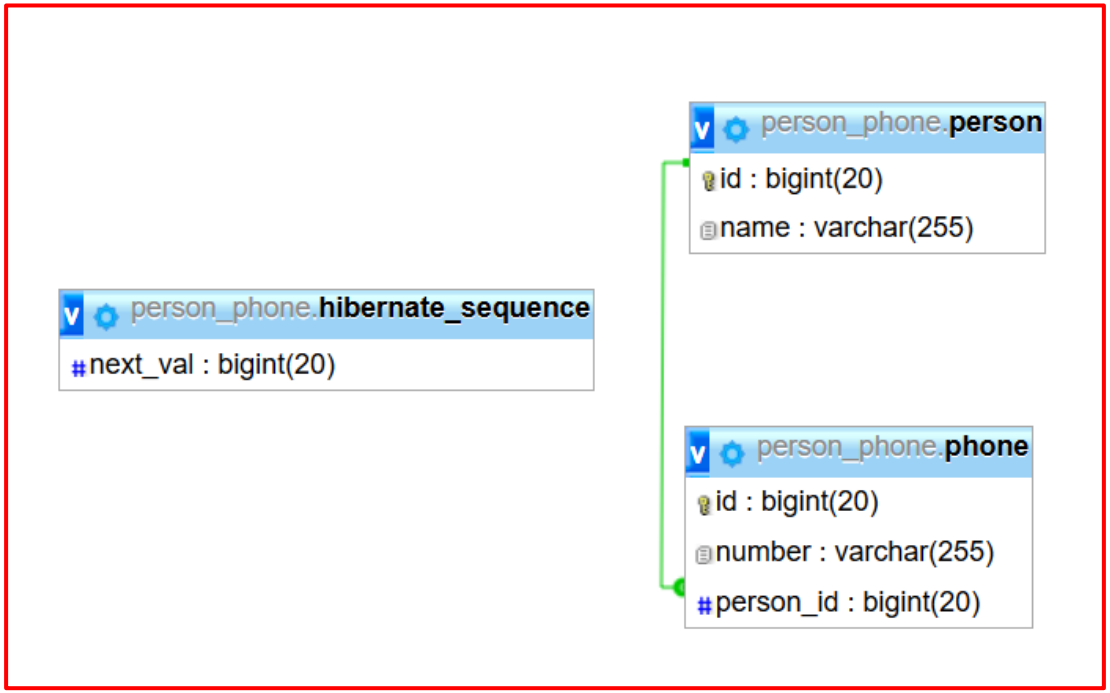
```
@ManyToOne
@JoinColumn(name = "person_id",
            foreignKey = @ForeignKey(name="PERSON_ID_FK"))
private Person person;
```

2. Asociación Many-To-One (Muchos a uno)

```java
@Entity
public class Person {
    @Id
    @GeneratedValue
    private long id;
    private String name;

    public Person() {
    }

    public Person(String name) {
        this.name = name;
    }

    public long getId() {
        return id;
    }
    //resto de métodos...
}
```

```java
@Entity
public class Phone {
    @Id
    @GeneratedValue
    private long id;
    private String number;
    @ManyToOne
    @JoinColumn(name = "person_id",
                foreignKey = @ForeignKey(name = "PERSON_ID_FK"))
    private Person person;

    public Phone() {
    }

    public Phone(String number) {
        this.number = number;
    }

    //resto de métodos...
}
```

2. Asociación Many-To-One (Muchos a uno)



2.1. Propiedades de @JoinColumn

Ahora pasemos a analizar las propiedades que nos brinda @JoinColumn :

- name: Indica el nombre con el que se deberá de crear la columna dentro de la tabla.
- foreignKey: Le indica a JPA si debe de crear el Foreign Key, esta propiedad recibe uno de los siguientes valores CONSTRAINT , NO_CONSTRAINT, PROVIDER_DEFAULT definidos en la enumeración javax.persistence.ForeignKey .
- referencedColumnName: Se utiliza para indicar sobre que columna se realizará el Join de la otra tabla. Por lo general no se suele utilizar, pues JPA asume que la columna es el ID de la Entidad objetivo.
- unique: Crea un constraints en la tabla para impedir valores duplicados (default false).
- nullable: Crea un constraints en la tabla para impedir valores nulos (default true).

2.1. Propiedades de @JoinColumn

- insertable: Le indica a JPA si este valor deberá guardarse en la operación de inserción (default true). **Si la relación es bidireccional será necesario ponerlo a false.**
- updatable: Le indica a JPA si el valor deberá actualizarse durante el proceso de actualización (default true). **Si la relación es bidireccional será necesario ponerlo a false.**
- columnDefinition: Esta propiedad se utiliza para indicar la instrucción SQL que se deberá utilizar para crear la columna en la base de datos. Esta nos ayuda a definir exactamente como se creará la columna sin depender de la configuración de JPA.
- table: Le indicamos sobre que tabla deberá realizar el JOIN, normalmente no es utilizada, pues JPA asume la tabla por medio de la entidad objetivo.

3. Asociación One-To-Many

La asociación **Uno-a-muchos** nos permite enlazar, en una asociación padre/hijo, el lado *padre* con todos sus hijos. Para ello, en la clase debemos colocar una colección de elementos hijos.

Si la asociación *@OneToMany* no tiene la correspondiente asociación *@ManyToOne*, decimos que es **unidireccional**. En caso de que sí exista, decimos que es **bidireccional**.

3.1. One-To-Many unidireccional

Para representar esta asociación, añadimos en la clase padre un listado de elementos del tipo hijo.

```java
@Entity
public class Person {
    @Id
    @GeneratedValue
    private long id;
    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Phone> phones = new ArrayList<>();
    private String name;

    public Person() {
    }

    public Person(String name) {
        this.name = name;
    }

    //Resto de métodos...
}
```

```java
@Entity
public class Phone {
    @Id
    @GeneratedValue
    private long id;
    private String number;

    public Phone() {
    }
    // Resto de métodos

}
```

3.1. One-To-Many unidireccional

Hibernate creará una tabla para cada entidad, y otra tabla para asociar ambas, añadiendo una restricción de unicidad al identificador del lado muchos. La anotación que define la asociación es *@OneToMany*. Vamos a repasar algunos elementos del código:

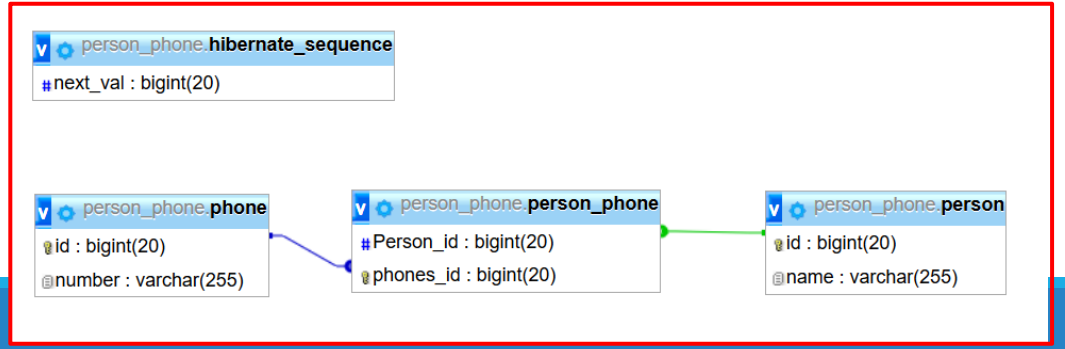
```
@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
private List<Phone> phones = new ArrayList<>();
```

- La lista es inicializada al instanciar el objeto, para poder almacenar los elementos hijos.
- La opción *cascade = CascadeType.ALL* indica que las operaciones (formalmente llamadas transiciones entre estados) sobre el elemento padre se actualizarán hacia los hijos.
- La opción *orphanRemoval = true* indica que una entidad Phone será borrada cuando se elimine su asociación con la instancia de Person.

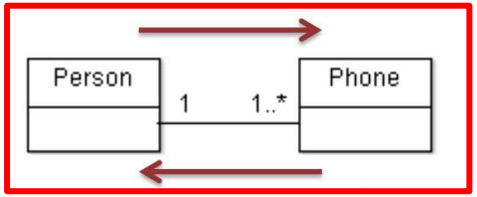
3.1. One-To-Many unidireccional

Para gestionar esta asociación *unidireccional*, debemos trabajar siempre con la lista de elementos que posea la entidad padre.

```
Person person = new Person("Pepe");
Phone phone1 = new Phone("954000000");
Phone phone2 = new Phone("600000000");

person.getPhones().add(phone1);
person.getPhones().add(phone2);

em.persist(person);
```



3.2. One-To-Many bidireccional



La asociación *@OneToMany* bidireccional necesita de una asociación *@ManyToOne* en el lado hijo. Aunque en los modelos estas asociaciones se representan bidireccionalmente, esta está representada solamente por una clave externa.

Toda asociación bidireccional debe tener un lado **propietario** (lado hijo). El otro lado vendrá referenciado mediante el atributo *mappedBy*.

```java
@Entity
public class Person {

    @Id
    @GeneratedValue
    private long id;

    @OneToMany(mappedBy = "person", cascade = Cascac
    private List<Phone> phones = new ArrayList<>();
```

```java
@Entity
public class Phone {

    @Id
    @GeneratedValue
    private long id;

    private String number;

    @ManyToOne
    private Person person;
```

3.2. One-To-Many bidireccional

```java
@Entity
public class Phone {
    @Id
    @GeneratedValue
    private long id;
    private String number;
    @ManyToOne
    private Person person;

    public Phone() {
    }
    //Resto de métodos
}
```

```java
@Entity
public class Person {
    @Id
    @GeneratedValue
    private long id;
    @OneToMany(mappedBy = "person", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Phone> phones = new ArrayList<>();
    private String name;

    public Person() {
    }

    public Person(String name) {
        this.name = name;
    }

    public long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public List<Phone> getPhones() {
        return phones;
    }

    public void addPhone(Phone phone) {
        phones.add(phone);
        phone.setPerson(this);
    }

    public void removePhone(Phone phone) {
        phones.remove(phone);
        phone.setPerson(null);
    }
}
```

3.2. One-To-Many bidireccional

Podemos destacar aquí la lógica de estos métodos:

```
public void addPhone(Phone phone) {
    phones.add(phone);
    phone.setPerson(this);
}
```

El primero, realiza la asociación de un teléfono a una persona: se añade el teléfono a la lista de teléfonos de la persona, y se asigna la persona como propietaria del teléfono.

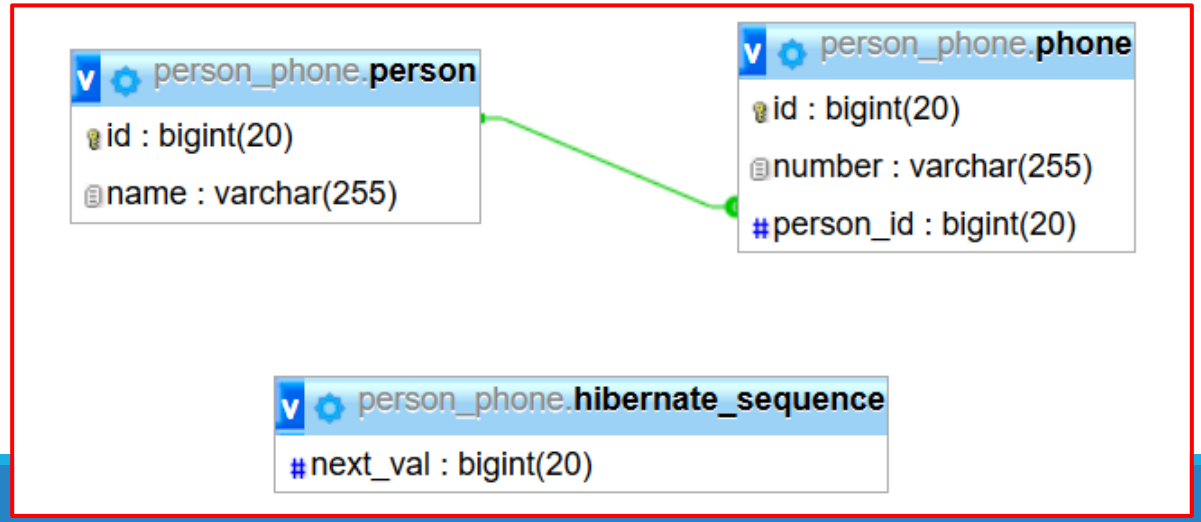
3.2. One-To-Many bidireccional

```
public void removePhone(Phone phone) {
     phones.remove(phone);
     phone.setPerson(null);
}
```

El segundo, realiza la operación inversa, sacando el teléfono de la lista de teléfonos de la persona, y asignando como nulo la persona que es dueña del teléfono.

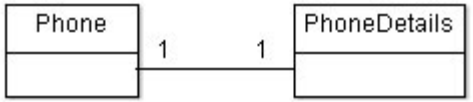
3.2. One-To-Many bidireccional

La asociación bidireccional es más eficiente gestionando el estado de persistencia de la asociación. La eliminación de un elemento solo requiere de una sentencia UPDATE (poniendo a NULL la clave externa); además, si el ciclo de vida de la clase hija está enmarcado dentro del de su clase padre (es decir, que no puede existir sin ella) podemos anotar la asociación con el atributo *orphanRemoval*, de forma que al desasociar un hijo, automáticamente se eliminará la fila asociada.



4. Asociación One-To-One

Las asociaciones Uno-a-Uno también pueden ser unidireccionales o bidireccionales.



4.1. One-To-One unidireccional

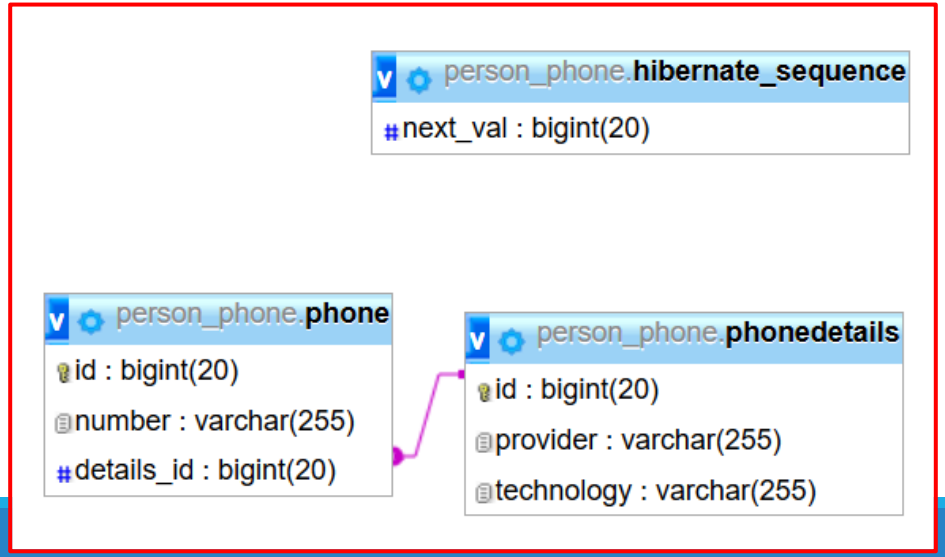
En una asociación uno-a-uno, solamente una instancia de una clase se asocia con una instancia de otra. Al usar el esquema *unidireccional*, tenemos que decidir un lado como **propietario**.

```java
@Entity
public class Phone {
    @Id
    @GeneratedValue
    private long id;
    private String number;
    @OneToOne
    @JoinColumn(name = "details_id")
    private PhoneDetails details;
    public Phone() {
    }
    //Resto de métodos

}
```

```java
@Entity
public class PhoneDetails {
    @Id
    @GeneratedValue
    private Long id;
    private String provider;
    private String technology;

    public PhoneDetails() {
    }

    public PhoneDetails(String provider, String technology) {
        this.provider = provider;
        this.technology = technology;
    }

    // Resto de métodos

}
```

4.1. One-To-One unidireccional

En apariencia, el tratamiento de esta asociación es como una *@ManyToOne* (desde el punto de vista relacional, estamos usando una clave externa). Sin embargo, no tendría sentido entender la entidad *PhoneDetails* como padre, y *Phone* como hija (no podemos tener unos detalles de teléfono sin un teléfono). Para solucionar esto, podemos usar una asociación bidireccional.



4.2. One-To-One bidireccional

Para añadir el tratamiento bidireccional, añadimos un reflejo de la asociación en la clase *PhoneDetails*.

```java
@Entity
public class PhoneDetails {
    @Id
    @GeneratedValue
    private Long id;

    private String provider;
    private String technology;

    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "phone_id")
    private Phone phone;

    public PhoneDetails() {
    }

    // Resto de métodos

}
```

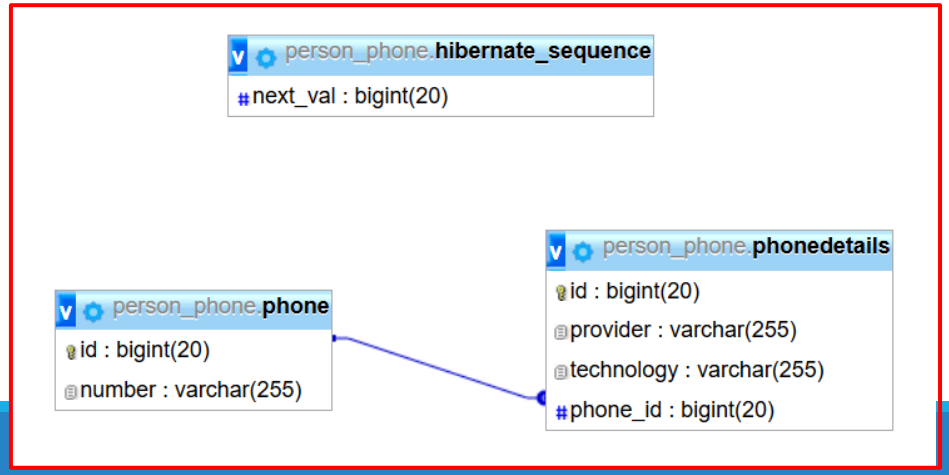
```java
@Entity
public class Phone {
    @Id
    @GeneratedValue
    private long id;
    private String number;
    @OneToOne(mappedBy = "phone",
            cascade = CascadeType.ALL,
            orphanRemoval = true,
            fetch = FetchType.LAZY)
    private PhoneDetails details;

    public Phone() {
    }

    // Resto de métodos
    public PhoneDetails getDetails() {
        return details;
    }

    public void addDetails(PhoneDetails details) {
        details.setPhone(this);
        this.details = details;
    }

    public void removeDetails() {
        if (details != null) {
            details.setPhone(null);
            this.details = null;
        }
    }

}
```

4.2. One-To-One bidireccional

Los métodos “addDetails” y “removeDetails” gestionan la lógica de la asociación.

De esa forma, el ciclo de vida de uso sería como sigue:

```
Phone phone = new Phone("954000000");
PhoneDetails details = new PhoneDetails("Movistar", "Fijo");
phone.addDetails(details);
em.persist(phone);
```



¿En qué casos sería más beneficioso tener una relación bidireccional y cuándo una unidireccional?

La asociación bidireccional tiene sentido si vamos a realizar un uso frecuente de ella. Si además, el uso es muy frecuente, y el volumen de datos no es muy grande, incluso nos podemos plantear la utilización de un FETCH de tipo EAGER (explicado más adelante).

Si el volumen de datos (número de filas en bases de datos, o de instancias de objetos) es muy grande, o los objetos tienen muchas relaciones con otros objetos, el FETCH de tipo EAGER no es nada recomendable, y entrarían en juego las consultas, que rellenarían de datos esas asociaciones bidireccionales en el momento justo en que vayamos a necesitarlos.

Si no vamos a utilizar nunca la asociación en ambas direcciones, y solo la vamos a usar en una (habitualmente, en el lado @ManyToOne), no merece la pena establecer la asociación en la dirección contraria.

Dudas y preguntas

Práctica

A partir de la relación Empleados → Departamentos, créate las asociaciones One-To-Many unidireccional y bidireccional en proyectos distintos y comprueba qué tablas se crean en cada caso.